UNIVERSITÀ DEGLI STUDI DI MILANO-BICOCCA

Dipartimento di Informatica Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica
Scuola di Scienze

Methods of Scientific Calculation
Project 1 - Comparison of Sparse Linear Systems solvers via Cholesky Decomposition

Candidati
Stranieri Francesco
Matr. 816551

Porto Francesco
Matr. 816042

Manan Mohammad Ali
Matr. 817205

Anno Accademico 2019–2020

Abstract

Linear Systems involving sparse, symmetric and positive defined matrices occur in many practical applications, such as in computer vision, fluid dynamics or electromagnetism.

We suppose our company needs to solve such systems via the Cholesky method, and has asked us to find the best programming environment for this task: should a reliable paid environment such as MATLAB be preferred, or is it worth investigating into free open-source alternatives? Also, does the choice of operating system matters?

Contents

1 Introduction

In this report we analyze and compare four different implementations of Sparse Linear Systems solvers that make use of the Cholesky Decomposition for sparse, symmetric and positive defined matrices from the SuiteSparse Matrix Collection [1]. We used one closed-source environment, **MATLAB**, and three different open-source environments, **C++**, **Python** and **R**. For each environment, we will take into consideration its *execution time*, *memory usage* and *relative error*. We will also make comparisons between different operating systems, (**Windows** and **Linux**), in order to understand if the choice matters. The aim of this study is to decide whether MATLAB is worth paying for or free and open-source solutions are good enough.

2 Theorical Background

In the next few subsections we will provide a brief introduction to a key concepts that will be discussed upon in the report.

2.1 Sparse Matrices

A matrix is said to be *sparse* when only a small portion of its entries are different than 0. In order to maximize efficiency, only values that differ from 0 are stored. These values are stored as a set of **triplets**:

$$(i, j, a_{i,j})$$

where i is the row index, j is the column index, and $a_{i,j}$ is the value of the matrix in row i and column j.

Linear systems involving sparse matrices are often computationally expensive to solve, due to the **fill-in problem**: in order to solve the system via traditional solving methods (such as *Gaussian elimination*) one would need to allocate space for the whole matrix.

2.2 Cholesky Decomposition

Definition. *Given a matrix $A \in \mathbb{R}^{n \times n}$, A is said to be:*

1. ***symmetric**, if $A = A^t$.*
2. ***positive defined**, if $\mathbf{y}^t A \mathbf{y} > 0, \forall \mathbf{y} \in \mathbb{R}^n \backslash \{\mathbf{0}\}$.*

Theorem (Cholesky). *Let $A \in \mathbb{R}^{n \times n}$ be a symmetric and positive defined matrix. Then, there exists an upper triangular matrix $R \in \mathbb{R}^{n \times n}$ such that:*

$$A = R^t R$$

If all elements in the main diagonal of R are positive, then R is unique.

We will now provide an overview of how to compute the matrix R. If we consider R as:

$$R := \left[\begin{array}{c|c} \rho & \mathbf{r}_1^t \\ \hline 0 & R_* \end{array}\right]$$

We can now obtain the transpose R^t, and multiply it by R:

$$R^t R = \begin{bmatrix} \rho & \mathbf{0} \\ \mathbf{r}_1 & R_*^t \end{bmatrix} \begin{bmatrix} \rho & \mathbf{r}_1^t \\ \mathbf{0} & R_* \end{bmatrix} = \begin{bmatrix} \rho^2 & \rho\mathbf{r}_1^t \\ \rho\mathbf{r}_1 & \mathbf{r}_1\mathbf{r}_1^t + R_*^t R \end{bmatrix}$$

And finally, if we make $A = R^t R$ we obtain the following linear system:

$$\begin{bmatrix} \alpha & \mathbf{a}_1^t \\ \mathbf{a}_1 & A_{1,1} \end{bmatrix} = \begin{bmatrix} \rho^2 & \rho\mathbf{r}_1^t \\ \rho\mathbf{r}_1 & \mathbf{r}\mathbf{r}^t + R_*^t R_* \end{bmatrix} \Rightarrow \begin{cases} \alpha = \rho^2 \\ \mathbf{a}_1 = \rho\mathbf{r}_1 \\ \mathbf{a}_1^t = \rho\mathbf{r}_1^t \\ A_{1,1} = \mathbf{r}_1\mathbf{r}^t + R_*^t R_* \end{cases}$$

By solving this linear system, we obtain the first column (and the first row since it's the transpose) of the matrix R. We then iterate through all the columns of A in order to obtain the remaining columns (and rows) of R.

2.3 Relative Error

Given an approximate solution x_h for a certain linear system, and the exact solution $\tilde{\mathbf{x}}$, we define its relative error as:

$$e_{rel} := \frac{||\mathbf{x}_h - \tilde{\mathbf{x}}||}{||\tilde{\mathbf{x}}||} \tag{1}$$

where $\|\cdot\|$ is the norm between vectors. The relative error gives us an idea of how close the exact and approximate solutions are to each other, normalized according to the exact solution.

3 Implementation Details

Our implementation is **fully automatized** and **multi-platform**, since it takes as input the path with the matrices in either *MAT* or *MTX* format, and returns a result.txt file, which contains, for each matrix:

- **DateTime**, day and time when execution started.
- **Platform**, the operating system used.
- **Language**, the programming environment used.
- **MatrixName**, the name of the matrix.
- **MatrixSize**, the size of the matrix.
- **NonZero**, the number of non-zero entries.
- **RelativeError**, the relative error, as shown in Equation 1.
- **ExecutionTime(ms)**, the overall time required for decomposing the matrix and solving the linear system associated, in milliseconds.
- **MemoryAllocated(KB)**, the overall memory required for decomposing the matrix and solving the linear system associated, in kilobytes.

We have also created another script, called DisplayResults.py, that parses the result.txt file, creates a Pandas DataFrame for storing the various results, and finally draws the plots that will be shown later.

3.1 Pseudocode

The Algorithm 1 which has been followed in all our programs is provided in form of pseudocode. Obviously, each environment has its own quirks when executing certain operations (these will be discussed later), but we feel this representation can help get a high-level idea of how our programs work.

In the next few sections we will provide, for each programming environment, an usage description in terms of the main functions for loading the matrices, computing their Cholesky decomposition and solving their associated linear systems. We will also discuss their maintenance status, with regards to documentation quality (in terms of completeness and overall simplicity) and license type. An image providing a quick recap on different license types can be found in Appendix A.7.

Algorithm 1 MatrixSolver

```
1:  procedure matrixSolver(matrixFilename)
2:      ▷ Obtain matrix information
3:      $matrixName \leftarrow name\,of\,the\,matrix$
4:      $matrix \leftarrow load(matrixFilename)$
5:      $matrixSize \leftarrow size\,of\,the\,matrix$
6:      $matrixNnz \leftarrow nonzero\,of\,the\,matrix$
7:      ▷ Set-up the linear system
8:      $xEs \leftarrow ones(matrixSize)$
9:      $b \leftarrow matrix * xEs$
10:     ▷ Solve the linear system, keeping track of time and memory
11:     $StartTime \leftarrow timer.start()$
12:     $StartMemory \leftarrow memory\_profiler.start()$
13:         $R \leftarrow cholesky(matrix)$
14:         $x \leftarrow solve(R, b)$
15:     $EndTime \leftarrow timer.stop()$
16:     $EndMemory \leftarrow memory\_profiler.stop()$
17:     ▷ Compute stats
18:     $executionTime \leftarrow EndTime - StartTime$
19:     $memoryAllocated \leftarrow EndMemory - StartMemory$
20:     $relativeError \leftarrow \frac{|x-xEs|}{|xEs|}$
21:     ▷ Return result, also taking errors into consideration
22:     if Error then
23:         $executionTime \leftarrow 0$
24:         $memoryAllocated \leftarrow 0$
25:         $relativeError \leftarrow 0$
26:     return matrixName, matrixSize, matrixNnz, executionTime, memoryAllocated, relativeError
```

4 MATLAB

MATLAB [2] is a multi-paradigm numerical computing environment and proprietary programming language developed by MathWorks. MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages.

4.1 Usage

The full code is available in Appendix A.1.

Loading the matrix

We first load the matrix by using load(filename) function, where filename is a *MAT* file. MATLAB detects automatically if the loaded matrix is symmetric and positive defined.

Cholesky decomposition and finding a solution

In order to compute the Cholesky decomposition of a given sparse matrix A, we use the chol(A) function. This function factorizes symmetric positive definite matrix A into an upper triangular R that satisfies $A = R^t R$. We can then solve the associated linear problem by using Matlab's built-in **backslash operator** in the following way `x = R(R'\b)`.

Difficulties Experienced

No particular difficulties have been experienced, at least from an implementation point of view.

4.2 Maintenance Status

The version we used is R2020a. MATLAB receives multiple updates every few months (for example in 2019 a total of 8 updates were released), and a full-on new version is released each year.

Documentation

MATLAB's documentation [3] is plentiful, while also being very reliable and in-depth. Many examples are also provided, which should allow even an inexperienced user to work with this environment.

License

MATLAB is not free. It can be bought for 800$ per year, or 2000$ for a perpetual license. However, students and researchers can get it for free through their university.

5 Eigen (C++)

Eigen [4] is a C++ pure template library (defined in the headers) for linear algebra: matrices, vectors, numerical solvers, and related algorithms. Eigen is versatile, fast, reliable, elegant and with a good compiler support.

5.1 Usage

The full code is available in Appendix A.2.

Loading the matrix and creating the linear system

We first load the folder containing the *MTX* files via loadMarket(A, filename), where A is the sparse matrix that will contain the values of the matrix itself. We have to explicitly define the matrix as an object of type SparseMatrix¡double¿.

Cholesky decomposition and finding a solution

We first create the Cholesky decomposition of the sparse matrix A via $SimplicialLLT{<}SpMat{>}solver(A)$. From the documentation we know that this decomposition is particularly stable on positive definite matrices. In order to reduce the fill-in, a symmetric permutation P is applied prior to the factorization itself. The solve() function computes the solution of the linear systems.

Difficulties Experienced

Eigen's load function is no longer being supported (it is necessary to include an unsupported header). It is also bugged on symmetric matrices in *MTX* format, such as the ones downloaded from the SuiteSparse Matrix Collection website, because only the upper triangular matrix is properly loaded. In order to bypass this issue we had to create a MATLAB script (MatrixConversion.m) that performs the conversion from *MAT* to *MTX* format, while also storing both the upper triangular matrix and lower triangular matrix. This however doubles the amount of storage required to store each matrix, while also making the overall read process slower, since each value has to be read twice. The script's code is available in Appendix A.6.

As an alternative, we also tried the Armadillo [5] library, that supports the MAT file format natively. However it only supports the Cholesky decomposition for dense matrices.

5.2 Maintenance Status

Eigen gets constantly updated by its community, but the latest stable version is Eigen 3.3.7, which released back in December 2018.

Documentation

Eigen's main documentation is organized into chapters covering all of its different use cases (including sparse matrices). They are themselves composed of user manual pages describing the different features in a difficult but comprehensive way.

License

Starting from the 3.1.1 version, Eigen is licensed under the MPL2 license; previous versions were licensed under the LGPL3+. It is also important to notice that Eigen relies on third party software (such as SimplicialCholesky) that is licensed under the LGPL, but can be explicitly disabled or replaced by other equivalent libraries.

6 Matrix (R)

Matrix [6] provides a rich hierarchy of matrix classes, including triangular, symmetric, and diagonal matrices, both dense and sparse. The Matrix package consists of basically two parts: Matrix's own C code and the included libraries (CHOLMOD, COLAMD, CSparse and SPQR) from the SuiteSparse collection.

6.1 Usage

The full code is available in Appendix A.3.

Loading the matrix and creating the linear system

In order to obtain the matrix, we use the readMat(fileName) function, since Matrix does not support reading from *MAT* files directly. We treat the matrix as a dsCMatrix. A dsCMatrix represents a symmetric, sparse numeric matrix in a compressed, column-oriented format.

Cholesky decomposition and finding a solution

We can compute the Cholesky decomposition of a matrix A via R¡-Cholesky(A), and then find the approximate solution X via the x¡-solve(R,b) function. The method is based on functions from the CHOLMOD library.

Difficulties Experienced

RStudio's latest version has serious issues on Unix devices (mostly due to issues related to GPU drivers), so downgrading to an older version was necessary. However, running the scripts via R's command line utility RScript did not cause any issue.

6.2 Maintenance Status

Matrix's latest stable version 1.2-18 was released in November 2019. Development is also active.

Documentation

Matrix's documentation [7] is a bit more minimalist when compared to other environments; nonetheless it is very clear and thorough.

License

Matrix is released under GPL-3, the GNU GENERAL PUBLIC LICENCE Version 3.

7 Scikits-Sparse (Python)

Scikit-sparse [8] is a companion to the scipy.sparse library (which does not support the Cholesky decomposition routine) for sparse matrix manipulation in Python. The intent of scikit-sparse is to wrap GPL'ed code, such as SuiteSparse software [9]. Currently it is available only a wrapper for the CHOLMOD routines for sparse Cholesky decomposition, which we used.

7.1 Usage

The full code is available in Appendix A.4.

Loading the matrix and creating the linear system

We first load the matrix by using scipy.io.loadmat(filename) function. This function is capable of loading matrices in the *MAT* format. The matrix is stored as a csc_matrix.

Cholesky decomposition and finding a solution

In order to obtain the Cholesky decomposition, we use R = cholesky((A)). According to the documentation, this function performs a preliminary fill-reducing permutation before computing the Cholesky decomposition for a sparse matrix A. This function returns a factor which can then be used for obtaining the approximate solution x via x = R(b).

Difficulties Experienced

This library has huge compatibility issues on Windows, and requires multiple troubleshooting steps to get working. The library cannot be installed in conventional ways, such as via the PIP dependency manager or via Anaconda, due to several missing library (including, but not limited, to *cholmod.h*). We had to follow multiple guides [10][11] in order to first compile the required libraries, and then to locally install the library. We do discourage the use of this library on Windows, while it can be installed relatively painlessly on Linux via Anaconda. We think this may be due to Windows 10's latest updates, or due to the fact that the library has not been updated in many years.

7.2 Maintenance Status

Unfortunately, this library is no longer being actively developed. The last update was in 2017.

Documentation

The documentation [12] is very limited when compared to other environments. However, this makes it much easier to read and understand.

License

The wrapper code is released under a BSD license, but the internal libraries (e.g., CHOLMOD) used are released under the less permissive GNU GPL or LGPL.

8 Results Analysis

In this section we will present and analyze the results obtained by our scripts. For evaluating the performance of each environment, we will take into account the following metrics: *relative error*, *memory allocated* and *execution time*. The code used for drawing the plots is available in Appendix A.5.

We utilized the matrices in Table 1, where the columns represent the name of the matrix, its size and the number of NonZero.

Name	**Size**	**NonZero**
ex15	6.867	98.671
cfd1	70.656	1.825.580
shallow_water1	81.920	327.680
cfd2	123.440	3.085.406
parabolic_fem	525.825	3.674.625
apache2	715.176	4.817.870
G3_circuit	1.585.478	7.660.826
StocF-1465	1.465.137	21.005.389
Flan_1565	1.564.794	114.165.372

Table 1: Matrices analysed in our project.

Note that the number of NonZeros in *cfd1* is greater than the number of NonZeros in *shallow_water_1*, even though it's smaller in size.

8.1 Computer Specifications

The computer used for running all scripts has the following specifications:

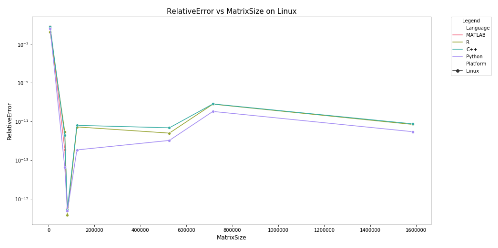
- **Device:** Desktop Computer
- **CPU:** 3,5 GHz Intel Core i5-4690K
- **RAM:** 8 GB 3700 MHz DDR3
- **Storage:** HDD 7200 rpm, 2 TB

8.2 Analysis on Operating Systems

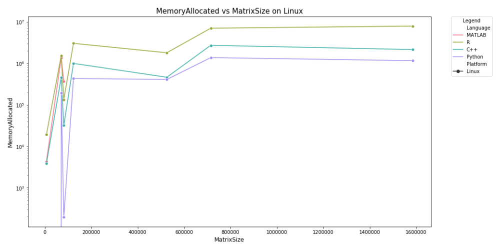
We have run our programs on two different operating systems, **Windows 10 2004** and **Ubuntu 20.04 LTS**.

8.2.1 Linux

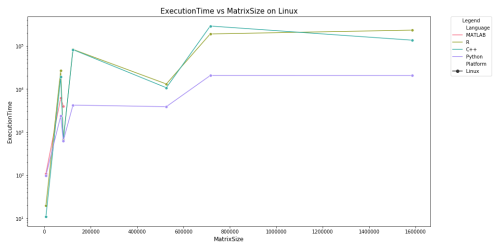
In Figure 1 it is possible to see the results relative to Linux. With regards to all the metrics, Python seems to get the best results. We feel this may be due to scikit-sparse using a more efficient implementation of CHOLMOD when compared to other environments. With regards to memory allocated, R seems to be the most memory-hungry environment. With respect to execution time, R and C++ are close to each other. Furthermore, MATLAB is only capable of solving smaller matrices, possibly due to taking more memory than other environments.



(a) Relative error on Linux.



(b) Memory allocated on Linux.

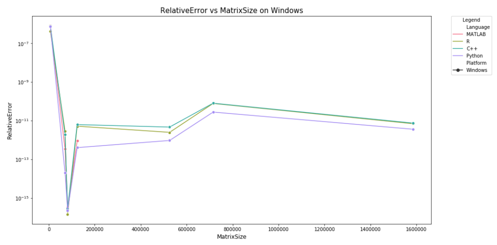


(c) Execution time on Linux.

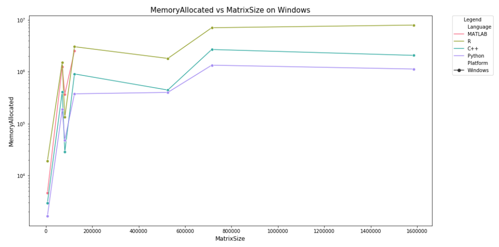
Figure 1: Plot of relative error (a), memory allocated (b) and execution time (c) on Linux.

8.2.2 Windows

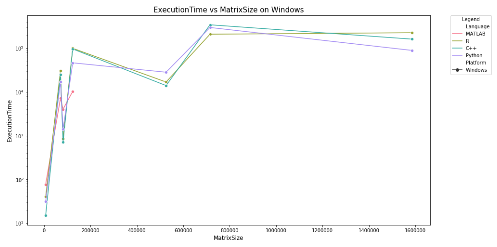
In Figure 2 it is possible to see the results relative to Windows. Python's relative error is best, while other environments look to be very similar. In terms of memory allocated, Python and C++ are somewhat close to each other; R is the environment that uses the most memory. With regards to execution time, all environments are close. As seen on Linux, MATLAB is still only capable of solving a few matrices.



(a) Relative error on Windows.



(b) Memory allocated on Windows.



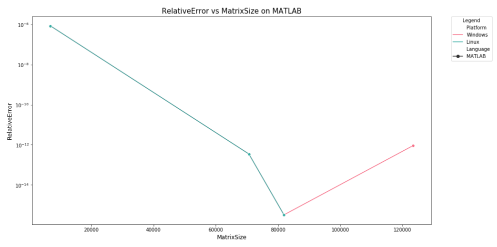
(c) Execution time on Windows.

Figure 2: Plot of relative error (a), memory allocated (b) and execution time (c) on Windows.

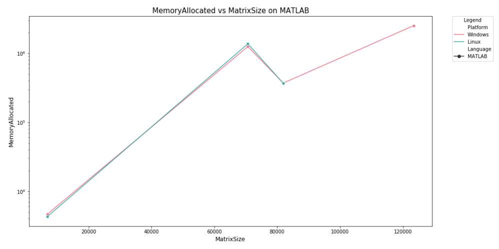
8.3 Analysis on Programming Languages

8.3.1 MATLAB

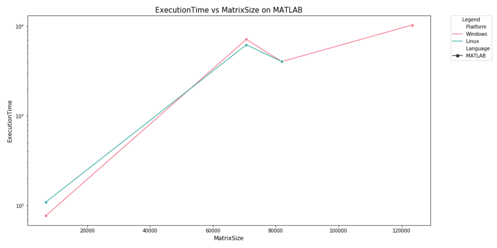
As seen in Figure 1 and 2, MATLAB is only capable of solving the linear system associated to the four smallest matrices. MATLAB's relative error is the same regardless of the operating system used, and the largest error seems to be on the smallest matrix, *ex15*. We excluded the matrix *cfd2* from the Linux results, due to the memory profiler reporting an inexplicably high value. The results also look similar for running time and memory usage, as can be seen in Figure 3.



(a) Relative error on MATLAB.



(b) Memory allocated on MATLAB.

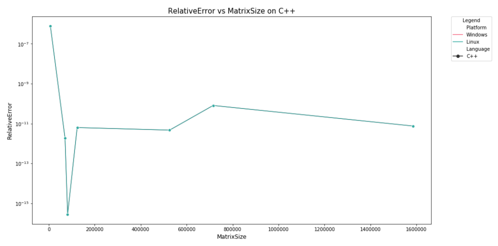


(c) Execution time on MATLAB.

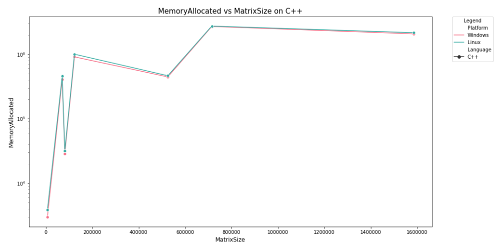
Figure 3: Plot of relative error (a), memory allocated (b) and execution time (c) on MATLAB.

8.3.2 C++

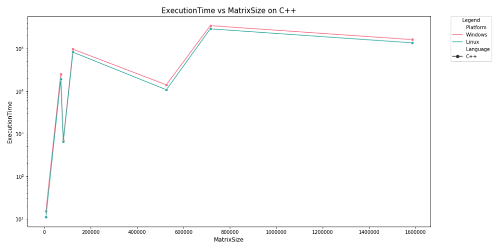
In figure 4 we can see the Eigen's results. Relative error is the same regardless of operating system, and the same can be said about memory usage. Windows appear to be slightly slower when compared to Linux.



(a) Relative error on C++.



(b) Memory allocated on C++.

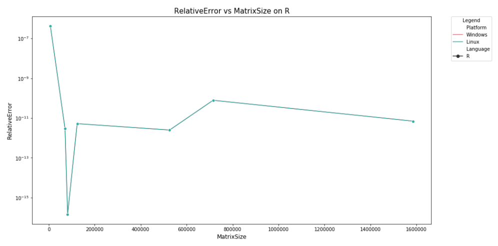


(c) Execution time on C++.

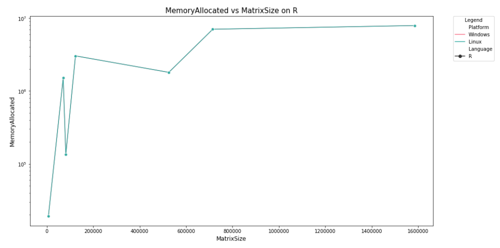
Figure 4: Plot of relative error (a), memory allocated (b) and execution time (c) on C++.

8.3.3 R

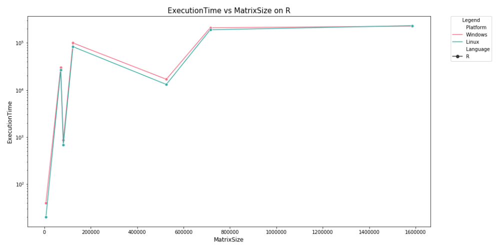
For R, both relative error and memory allocated are practically identical across different OS. In terms of time, Windows seems to be slightly slower. The results can be seen in Figure 5.



(a) Relative error on R.



(b) Memory allocated on R.

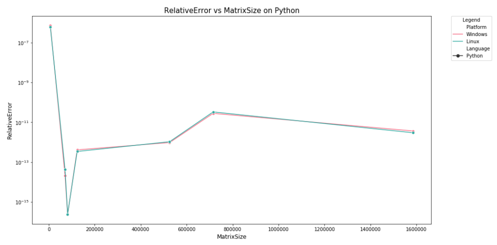


(c) Execution time on R.

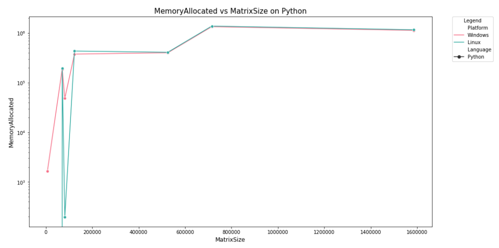
Figure 5: Plot of relative error (a), memory allocated (b) and execution time (c) on R.

8.3.4 Python

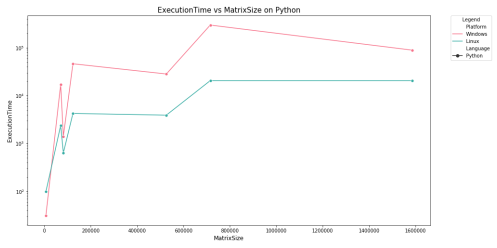
Relative error appears to be different depending on the operating system. In terms of memory usage, Python is much more efficient on Linux rather than Windows on the smallest matrices. Without taking into consideration *ex15*, time looks to be much faster on Linux, as can be seen in Figure 6. Python seems to be the environment with the most different performances across different operating systems.



(a) Relative error on Python.



(b) Memory allocated on Python.



(c) Execution time on Python.

Figure 6: Plot of relative error (a), memory allocated (b) and execution time (c) on Python.

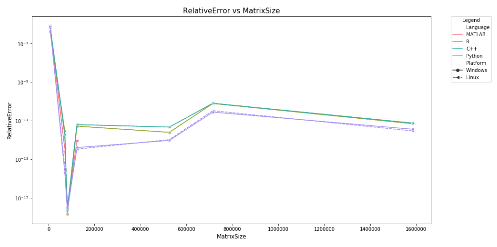
9 Conclusion

We have compared various programming languages for solving sparse linear systems by using the Cholesky decomposition.

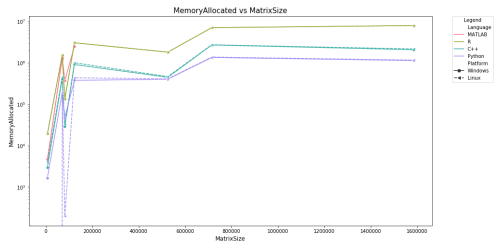
MATLAB, the only closed-source paid environment, is capable of solving only the smaller matrices probably due to high memory usage, making it difficult to compare it to other environments in a fair manner.

Open-source environments, on the other hand, were capable of solving almost all matrices (excluding the two heaviest ones in terms of size, *StocF-1465* and *Flan_1565*), making the comparison much easier.

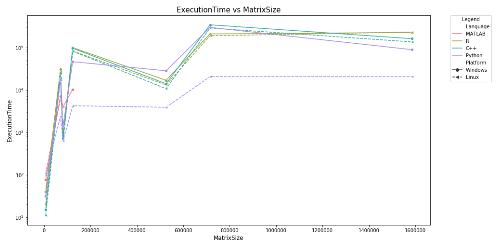
As a general result, Python on Linux seemed to obtain the best results according to all benchmarks. R appears to be the most memory-hungry of the three, while C++ seems a good all-rounder. These results can be seen in Figure 7 and are generated starting from the data in Tables 2, 4 and 3.



(a) Relative error.



(b) Memory allocated.



(c) Execution time.

Figure 7: Plot of relative error (a), memory allocated (b) and execution time (c).

While Python may seem the best overall choice, the difficulties experienced during its installation on Windows make it hard to recommend, as it is also not maintained anymore. It may still be worthwhile if the user or the company is Linux based and does not mind the lack of updates of the library.

Taking everything into consideration, the choice boils down to R and C++. We feel that R's easier syntax may make it more preferable to someone not already familiar with C++. If performances are a priority, then C++ will be the correct choice.

With regards to different operating systems, we found out that there was no noticeable difference between Windows and Linux (excluding Python), so either can be chosen, according to which one is preferred by the user or the company.

As a final recap: on Linux, our first pick would be Python, followed by C++ and R depending on the programmer's experience. On Windows, we would pick C++ or R, for the same reason. MATLAB could still be useful if the license fee is not a problem and a very large amount of RAM is available.

Name	**MATLAB**		**C++**		**R**		**Python**	
	Windows	**Linux**	**Windows**	**Linux**	**Windows**	**Linux**	**Windows**	**Linux**
ex15	8.573567e-07	8.573567e-07	7.953e-007	7.953e-07	4.331533e-07	4.331533e-07	7.466742e-07	6.211012e-07
cfd1	3.478955e-13	3.478955e-13	7.953e-007	1.88455e-12	2.956428e-12	2.956428e-12	2.073463e-14	4.257452e-14
shallow_water1	3.198500e-16	3.198500e-16	2.79838e-016	2.79841e-16	1.43627e-16	1.43627e-16	2.323659e-16	2.377267e-16
cfd2	9.157916e-13	9.157916e-13	6.31791e-012	6.31791e-12	5.241967e-12	5.241967e-12	4.071927e-13	3.38567e-13
parabolic_fem	-	-	4.70603e-012	4.70603e-12	2.504905e-12	2.504905e-12	9.519163e-13	1.047877e-12
apache2	-	-	8.08389e-011	8.08389e-11	7.854829e-11	7.854829e-11	2.802706e-11	3.30815e-11
G3_circuit	-	-	7.5445e-012	7.5445e-12	7.006596e-12	7.006596e-12	3.633075e-12	2.957097e-12
StocF-1465	-	-	-	-	-	-	-	-
Flan_1565	-	-	-	-	-	-	-	-

Table 2: Relative error.

Name	**MATLAB**		**C++**		**R**		**Python**	
	Windows	**Linux**	**Windows**	**Linux**	**Windows**	**Linux**	**Windows**	**Linux**
ex15	77	109	15	11	40	20	31	98
cfd1	7127	6214	25118	19061	30365	26717	17094	2376
shallow_water1	4028	4032	714	649	859	677	1406	632
cfd2	10306	9876	96273	81975	99583	82993	46328	4233
parabolic_fem	-	-	13985	10716	16868	13091	28266	3901
apache2	-	-	343655	288682	209148	189423	298406	20553
G3_circuit	-	-	161642	135745	226368	232104	89125	20528
StocF-1465	-	-	-	-	-	-	-	-
Flan_1565	-	-	-	-	-	-	-	-

Table 3: Execution time (ms).

Name	**MATLAB**		**C++**		**R**		**Python**	
	Windows	**Linux**	**Windows**	**Linux**	**Windows**	**Linux**	**Windows**	**Linux**
ex15	4661	4294	2981	3837	19149	19148	1655	0
cfd1	1246396	1365281	407842	455782	1516198	1516196	188273	193720
shallow_water1	368939	367084	28598	31899	134193	134192	48230	197
cfd2	2509894	-	906997	998621	3035863	3035862	375550	432488
parabolic_fem	-	-	444002	462872	1802744	1802743	399516	408883
apache2	-	-	2682703	2721669	7043548	7044512	1337291	1374970
G3_circuit	-	-	2065018	2152210	7918518	7917771	1126941	1159635
StocF-1465	-	-	-	-	-	-	-	-
Flan_1565	-	-	-	-	-	-	-	-

Table 4: Memory allocated (KB).

References

[1] Suitesparse matrix collection. https://sparse.tamu.edu/. Accessed: 2020-06-05.

[2] Matlab - il linguaggio del calcolo. https://it.mathworks.com/products/matlab.html. Accessed: 2020-06-02.

[3] Matlab documentation. https://it.mathworks.com/help/matlab/. Accessed: 2020-06-05.

[4] Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page. Accessed: 2020-06-02.

[5] Armadillo. http://arma.sourceforge.net/. Accessed: 2020-06-06.

[6] Matrix. https://cran.r-project.org/web/packages/Matrix/index.html. Accessed: 2020-06-02.

[7] Matrix documentation. https://www.rdocumentation.org/packages/Matrix. Accessed: 2020-06-05.

[8] scikit-sparse. https://scikit-sparse.readthedocs.io/en/latest/overview.html. Accessed: 2020-06-02.

[9] Suitesparse: A suite of sparse matrix software. http://faculty.cse.tamu.edu/davis/suitesparse.html. Accessed: 2020-06-12.

[10] suitesparse-metis-for-windows. https://github.com/jlblancoc/suitesparse-metis-for-windows. Accessed: 2020-06-05.

[11] Cholmod-scikit-sparse-windows. https://github.com/xmlyqing00/Cholmod-Scikit-Sparse-Windows. Accessed: 2020-06-05.

[12] Sparse cholesky decomposition. https://scikit-sparse.readthedocs.io/en/latest/cholmod.html. Accessed: 2020-06-05.

A Code

A.1 MATLAB Code

```
clear all
close all
clc

matrixFolder = 'SuiteSparse/MAT';
addpath(matrixFolder);
matrixList = dir(fullfile(matrixFolder, '*.mat'));
```

```
...

for i = 1:length(matrixList)
    load(matrixList(i).name);
    fileNameMatrix = split(matrixList(i).name,'.');
    matrixName = fileNameMatrix{1};
    fprintf('MatrixName %s \n',matrixName);

    matrixSize = size(Problem.A,1);
    fprintf('MatrixSize %d \n',matrixSize);
    nonZero = nnz(Problem.A);
    %spy(Problem.A)

    xEs = ones(matrixSize,1);
    b = Problem.A*xEs;

    logFileWrite = true;

    try
        profile clear
        profile -memory -history on
        tic;

        R = chol(Problem.A);
        x = R\(R'\b);

        executionTime = round(toc*1000);
        memoryAllocated = round((profile('info')
                            .FunctionTable().TotalMemAllocated)/1000);
        profile -memory -history off

        relativeError = norm(x- xEs)/norm(xEs);
    catch
        executionTime = 0;
        memoryAllocated = 0;
        relativeError = 0;

        logFileWrite = false;
    end

...

end
```

A.2 C++ Code

```
matrixSolved matrixSolver(const char *matrixList)
{
        matrixSolved matrixSolved;

        SpMat A;
        loadMarket(A, matrixList);

        matrixSolved.matrixSize = A.rows();
        cout << "MatrixSize " << matrixSolved.matrixSize << endl;
        matrixSolved.nonZero = A.nonZeros();

        VectorXd xEs = VectorXd::Ones(matrixSolved.matrixSize);
        VectorXd b = A * xEs;
        VectorXd x = VectorXd::Zero(matrixSolved.matrixSize);

        matrixSolved.logFileWrite = true;

        try
        {
                size_t memoryAllocatedStart = getCurrentRSS();
                high_resolution_clock::time_point start =
                    high_resolution_clock::now();

                SimplicialLLT<SpMat> solver(A);
                x = solver.solve(b);

                high_resolution_clock::time_point stop =
                    high_resolution_clock::now();
                duration<double, milli> differenceTime = (stop - start);
                size_t memoryAllocatedEnd = getCurrentRSS();

                matrixSolved.memoryAllocated =
                    (memoryAllocatedEnd - memoryAllocatedStart) / 1000;
                matrixSolved.executionTime = round(differenceTime.count());
                matrixSolved.relativeError = (x - xEs).norm() / xEs.norm();
        }
        catch (...)
        {
                matrixSolved.memoryAllocated = 0;
                matrixSolved.executionTime = 0;
                matrixSolved.relativeError = 0;

                matrixSolved.logFileWrite = false;
        }
        return matrixSolved;
}
```

A.3 R Code

```
matrixSolver <- function(matrix){
  fileNameMatrix <- unlist(strsplit(matrix, "\\/"))[3]
  matrixName <- unlist(strsplit(fileNameMatrix, "\\."))[1]
  cat("MatrixName ", matrixName, "\n")

  matrixProblem <- readMat(matrix)

  i = 1
  while(i < dim(matrixProblem$Problem)[1] &
    !identical(typeof(matrixProblem$Problem[[i]]), "S4")){
    i = i+1
  }

  matrixSize <- dim(matrixProblem$Problem[[i]])[1]
  cat("MatrixSize ", matrixSize, "\n")

  nonZero <- nnzero(matrixProblem$Problem[[i]])

  xEs <- matrix(1, matrixSize)
  b <- matrixProblem$Problem[[i]] %*% xEs

  logFileWrite <- TRUE

  tryCatch(
    expr = {
    profiler <- profmem({
      start <- Sys.time()

      R <- Cholesky(matrixProblem$Problem[[i]])
      x <- solve(R, b)

      executionTime <- difftime(Sys.time(), start, units = "secs") * 1000
      executionTime <- unlist(
        strsplit(as.character(as.integer(executionTime)), " "))
      relativeError <- signif(as.numeric((norm(x - xEs)) / norm(xEs)), 7)
      })
    memoryAllocated <- as.integer(sum(profiler$bytes) / 1000)
    },
    error = function(e){
      executionTime <- '0'
      memoryAllocated <- '0'
      relativeError <- '0'

      logFileWrite <- FALSE
    }
  )

  return(c(matrixName, matrixSize, nonZero,
         executionTime, memoryAllocated, relativeError, logFileWrite))
}
```

A.4 Python Code

```
def matrixSolver(matrix):
    fileNameMatrix = str(matrix).split(os.sep)[2]
    matrixName = fileNameMatrix.split(".")[0]
    print('MatrixName', matrixName)

    matrixProblem = scipy.io.loadmat(matrix)

    matrixSize = ((matrixProblem['Problem'])['A'][0][0]).shape[0]
    print('MatrixSize', matrixSize)
    nonZero = ((matrixProblem['Problem'])['A'][0][0]).getnnz()

    xEs = ones(matrixSize)
    b = ((matrixProblem['Problem'])['A'][0][0])*xEs
    x = empty(matrixSize)

    logFileWrite = True

    try:
        start = time.process_time()
        startMemoryAllocated = psutil.virtual_memory().used

        R = cholesky((matrixProblem['Problem'])['A'][0][0])
        x = R(b)

        endMemoryAllocated = psutil.virtual_memory().used

        executionTime = str(round((time.process_time() - start) * 1000))
        memoryAllocated = str(
            round((endMemoryAllocated - startMemoryAllocated) / 1000))
        relativeError = str("{:.7g}".format(norm(x - xEs) / norm(xEs)))
    except:
        executionTime = '0'
        memoryAllocated = '0'
        relativeError = '0'

        logFileWrite = False

    return (matrixName, matrixSize, nonZero,
            executionTime, memoryAllocated, relativeError, logFileWrite)
```

A.5 PlotResult Code

```
def drawPlotParam(y, hue, style, param, x = 'MatrixSize'):
    plt.figure(figsize=(15, 8))
    sns.lineplot(x = x, y = y, hue = hue, style = style, err_style = "bars",
                 markers = ['o', '<', '*'],
                 palette = 'husl', data = df.loc[df[style] == param])
    plt.yscale('log')
    plt.xlabel(x, fontsize = 12)
    plt.ylabel(y, fontsize = 12)
    plt.title(str(y) + ' vs ' + str(x) + ' on ' + str(param), fontsize = 15)
    plt.legend(title = 'Legend', bbox_to_anchor = (
        1.05, 1), loc = 2, borderaxespad = 0.)
    plt.show()

# Linux
drawPlotParam('RelativeError', 'Language', 'Platform', 'Linux')
drawPlotParam('ExecutionTime', 'Language', 'Platform', 'Linux')
drawPlotParam('MemoryAllocated', 'Language', 'Platform', 'Linux')

# MATLAB
drawPlotParam('RelativeError', 'Platform', 'Language', 'MATLAB')
drawPlotParam('ExecutionTime', 'Platform', 'Language', 'MATLAB')
drawPlotParam('MemoryAllocated', 'Platform', 'Language', 'MATLAB')


def drawPlotTotal(y, hue, style, x = 'MatrixSize'):
    plt.figure(figsize=(15,8))
    sns.lineplot(x = x, y = y, hue = hue, style = style,
        err_style = "bars", markers = ['o', '<', '*'],
        palette = 'husl', data = df)
    plt.yscale('log')
    plt.xlabel(x, fontsize= 12)
    plt.ylabel(y, fontsize = 12)
    plt.title(str(y) + ' vs ' + str(x), fontsize = 15)
    plt.legend(title = 'Legend', bbox_to_anchor = (1.05, 1),
        loc = 2, borderaxespad = 0.)
    plt.show()
```

```
drawPlotTotal('RelativeError', 'Language', 'Platform')
drawPlotTotal('ExecutionTime', 'Language', 'Platform')
drawPlotTotal('MemoryAllocated', 'Language', 'Platform')
```

A.6 MatrixConversion Code

MatrixConversion.m

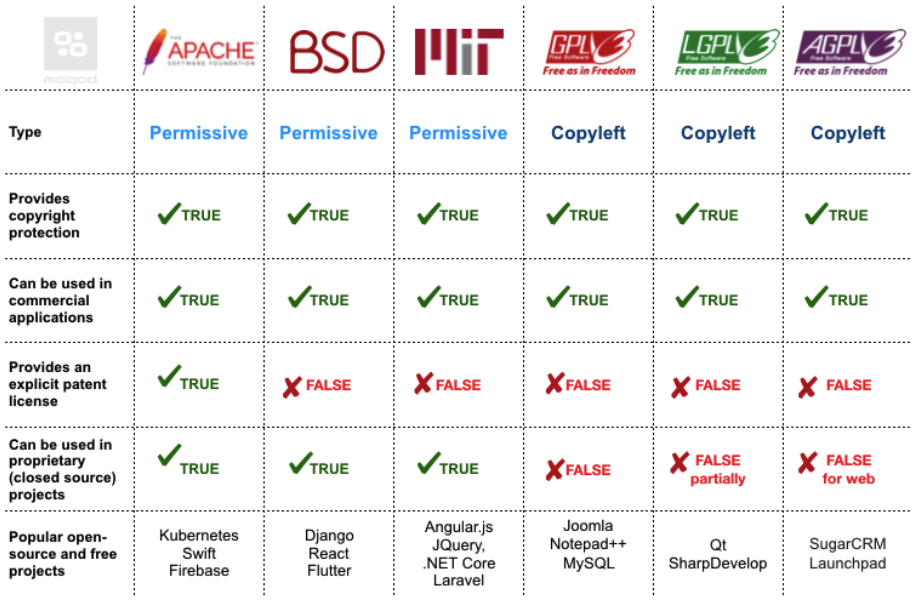
```
clear all
close all
clc

matrixFolderSource = 'SuiteSparse/MAT/';
addpath(matrixFolderSource);
matrixList = dir(fullfile(matrixFolderSource, '*.mat'));

matrixFolderDestination = 'SuiteSparse/MTX/';
addpath(matrixFolderDestination);

fileExtension = '.mtx';

for i = 1:length(matrixList)
  load(matrixList(7).name);

  fileName = split(matrixList(7).name,'.');
  matrixName = fileName{1};
  matrixName = strcat(matrixFolderDestination,matrixName,fileExtension)

  mmwrite_SRS(matrixName,Problem.A);

  clearvars fileName matrixName Problem;
end
```

mmwrite_SRS.m

```
% Original File
% https://math.nist.gov/MatrixMarket/mmio/matlab/mmiomatlab.html

function [ err ] = mmwrite_symmetric(filename,A,comment,precision)
mattype = 'real';
if (nargin > 2)
  precision = 16;
elseif (nargin == 2)
  comment = '';
  precision = 16;
end
```

```
mmfile = fopen([filename],'w');
if (mmfile == -1)
 error('Cannot open file for output');
end;

[M,N] = size(A);
if (issparse(A))
  [I,J,V] = find(A);

  issymm = 0;
  symm = 'general';
  NZ = nnz(A);

  rep = 'coordinate';

  fprintf(mmfile, '%%%%MatrixMarket matrix %s %s %s\n',
    'coordinate', mattype, symm);
  [MC,NC] = size(comment);
  if (MC == 0)
    fprintf(mmfile, '%% Generated %s\n', [date]);
  else
    for i = 1:MC,
      fprintf(mmfile, '%%%s\n', comment(i, :));
    end
  end

  fprintf(mmfile, '%d %d %d\n', M, N, NZ);
  realformat = sprintf('%%d %%d %% .%dg\n', precision);

  for i = 1:NZ
    fprintf(mmfile, realformat, I(i), J(i), V(i));
  end;
end
fclose(mmfile);
```

A.7 License Comparison



	APACHE	BSD	MIT	GPLv3	LGPLv3	AGPLv3
Type	Permissive	Permissive	Permissive	Copyleft	Copyleft	Copyleft
Provides copyright protection	✓ TRUE	✓ TRUE	✓ TRUE	✓ TRUE	✓ TRUE	✓ TRUE
Can be used in commercial applications	✓ TRUE	✓ TRUE	✓ TRUE	✓ TRUE	✓ TRUE	✓ TRUE
Provides an explicit patent license	✓ TRUE	✗ FALSE	✗ FALSE	✗ FALSE	✗ FALSE	✗ FALSE
Can be used in proprietary (closed source) projects	✓ TRUE	✓ TRUE	✓ TRUE	✗ FALSE	✗ FALSE partially	✗ FALSE for web
Popular open-source and free projects	Kubernetes Swift Firebase	Django React Flutter	Angular.js JQuery, .NET Core Laravel	Joomla Notepad++ MySQL	Qt SharpDevelop	SugarCRM Launchpad

Figure 8: License Comparison.