PACS' LABORATORY 1

by

Daniel Ariza Antón (775545) and Marcos Ilarraza Sarto (797720)

In this lab the objective is to compare the efficiency of a library which has been optimized with an implementation that solves the same problem but that hasn't been optimized. Specifically, it asks for an implementation of an algorithm that multiplies matrices of **NxN dimensions** and to compare it with the *eigen* library which is specialized in matricial operations, with and without vectorization.

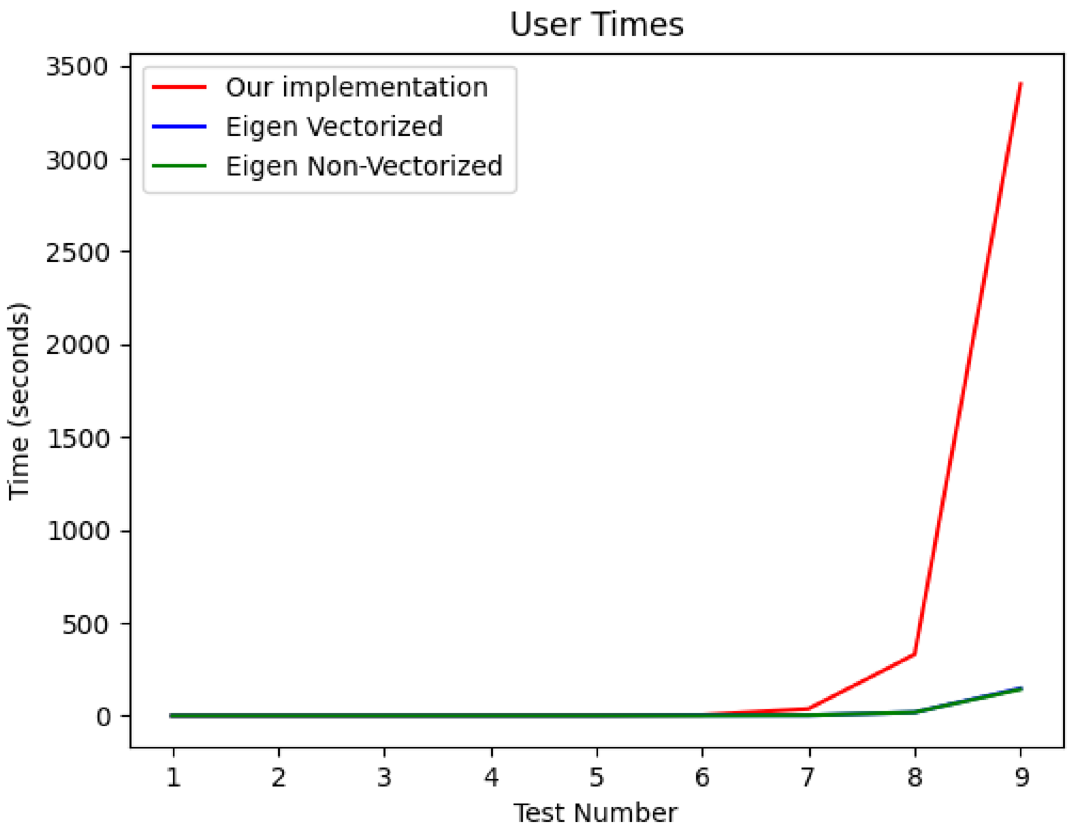
To do so, three **C++** files have been written, one for each case: without the *eigen* library (from now on the *first* algorithm), with the *eigen* library and vectorization enabled (from now on it will be referred as the *second* algorithm), and with the *eigen* library but without enabling the vectorization (which will be mentioned as the *third* algorithm). In each file there are the same nine scenarios, each one with a different size of matrices, in which the algorithm creates two matrices with random numbers and multiplies them. This random numbers are between ten and minus ten. The reason is that by testing the *random* function provided by the *eigen* library it was observed that it only produced numbers in that range. Therefore, the implementation of the *first* algorithm also fills its matrices with numbers in that range.

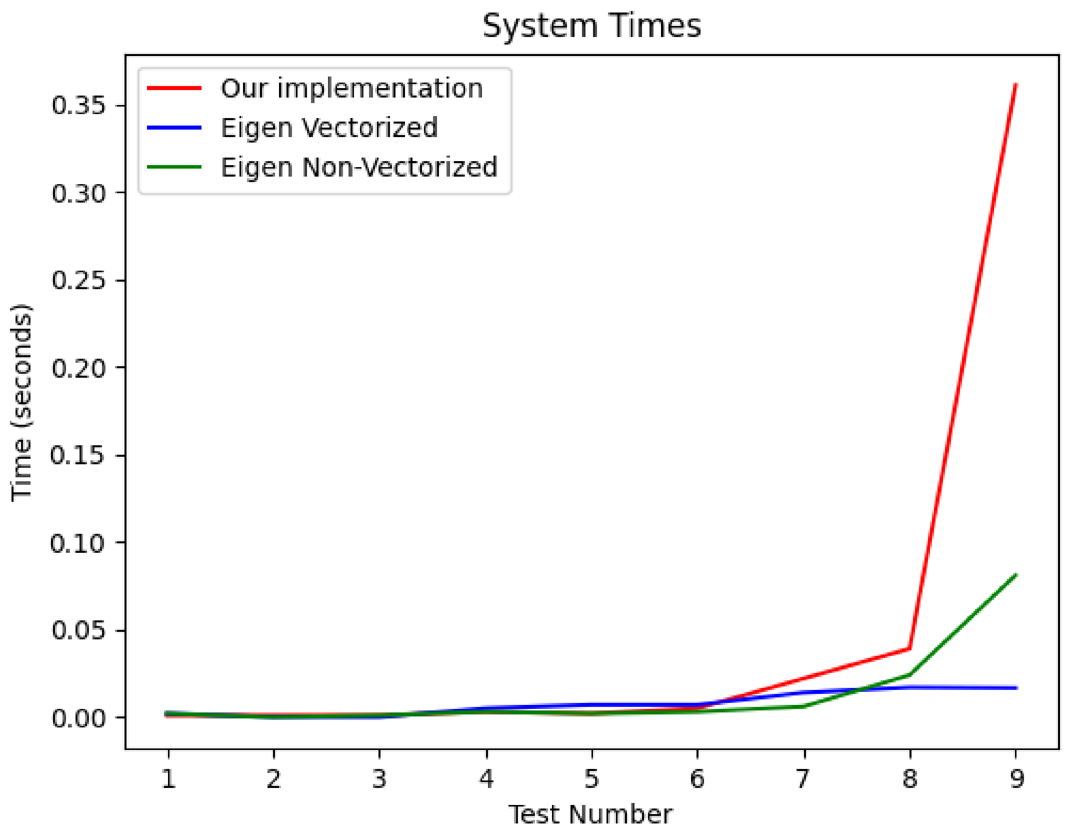
Each one of the algorithms has been tested with all the cases, and the results provided by the *time* command can be seen in the following table:

Alg \ N	**32 \ 1**	**100 \ 2**	**150 \ 3**	**708 \ 4**	**900 \ 5**	**1200 \ 6**	**2500 \ 7**	**5000 \ 8**	**10000 \ 9**
Ours (without eigen)	0.001s 0.001s	0.002s 0.001s	0.006s 0.001s	0.781s 0.003s	1.599s 0.002s	3.647s 0.005s	36.278s 0.022s	5m 29.159s 0m 0.039s	56m 41.547s 0m 0.361s
Eigen with Vect	0.000s 0.002s	0.002s 0.000s	0.003s 0.000s	0.059s 0.005s	0.263s 0.007s	2.263s 0.007s	2.266s 0.014s	17.734s 0.017s	2m 26.548s 0m 0.0167s
Eigen without Vect	0.000s 0.002s	0.002s 0.000s	0.001s 0.001s	0.062s 0.003s	0.119s 0.003s	0.267s 0.003s	2.274s 0.006s	17.738s 0.024s	2m 21.751s 0m 0.081s

This table includes both the *user* (the number on the top) and *system* (the number beneath) times that the algorithms have needed to solve both the creation and multiplication of the matrices.

The next graphs represent the evolution of the recorded times graphically for better comprehension and comparison:


User Times
Our implementation
Eigen Vectorized
Eigen Non-Vectorized
Time (seconds)
Test Number
3500
3000
2500
2000
1500
1000
500
0
1
2
3
4
5
6
7
8
9


System Times
Our implementation
Eigen Vectorized
Eigen Non-Vectorized
Time (seconds)
Test Number
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1
2
3
4
5
6
7
8
9

Except for the last scenario for the first algorithm, which lasted for almost an hour, each one of the tests has been executed several times in order to avoid outliers. It is also important to take into account that the values which equal to zero do not necessarily mean that it did not take time, it means that the time it took was shorter than a millisecond.

Both the *second* and *third* algorithm take similar times to solve each one of the scenarios, only seeing that the first is a bit faster than the former in the ninth scenario, in which it is almost 0.2 seconds faster. It is also noticeable that when using vectorization the amount of system time is slightly bigger in the case of the smaller matrices and gets smaller compared to the non-vectorized with the bigger matrices. To see their differences more clearly, it would be necessary to use bigger matrices.

By taking a look at the results of the *first* algorithm it is evident that the time that takes it to solve matricial multiplication increases much faster than the case of the algorithms which use the *eigen* library. Being the first great difference seen in the third scenario: the 708 by 708 matrices.

Regarding the implementation of the first algorithm, the matrices were defined as an struct which contains two integers to define the dimensions of the matrix and an integer pointer which will point to the integer vector that will contain its values. This integer vector is allocated in dynamic memory, which is necessary in order to use the same struct for declaring matrices of different dimensions. Also, it is an integer vector instead of a vector of pointers to integer because even though it is easier to know how to traverse the matrix it would imply a higher cost to access each element. Its multiplication method checks whether the matrices can be multiplied in the order that they were provided or not. So it only executes it if they can be multiplied. It implies that this implementation can deal with non-square matrices, although it was not necessary. It also has a method that given a positive number *a* it fills the matrix with real numbers in the range [-*a*,*a*], both included.