Performance Comparison of Open Source and Commercial Computing Tools in Educational and Other Use — Scilab vs. MATLAB

Mikac, Matija; Logožar, Robert; Horvatić, Miroslav

Source / Izvornik: **Tehnički glasnik, 2022, 16, 509 - 518**

Journal article, Published version
Rad u časopisu, Objavljena verzija rada (izdavačev PDF)

https://doi.org/10.31803/tg-20220528171032

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:122:899650

ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20220528171032

Original scientific paper
Received: 2022-05-28, Accepted: 2022-08-31

Performance Comparison of Open Source and Commercial Computing Tools in Educational and Other Use — Scilab vs. MATLAB

Matija Mikac*, Robert Logožar, Miroslav Horvatić

Abstract: In this paper, the authors compare the features and the overall performance of the two high-level numerical computing and modeling software environments: the freeware Scilab and commercially available industry-standard MATLAB. The motivation for the work emanated from the educational use of these tools at the college and university level, but with a perspective to their professional and scientific use as well. Their performance is tested by measuring the execution times of several combined-task benchmarks implemented as test functions, built upon nine common numerical tasks that are often found in programs for solving standard engineering problems. They include basic algebra and matrix calculations, signal generation, signal analysis, and storing and retrieving data to and from the hard disk drive. Although MATLAB outperforms Scilab in all the benchmarks except the disk file manipulations, in the presumed vectorization versions of the benchmarks, it is not for much. The overall performance of the freeware rival is very satisfactory, making it a good choice not only for educational use but also for scientific and professional purposes, especially when funding is critical.

Keywords: calculation benchmarks; mathematical and modeling software; MATLAB; programming; performance comparison; Scilab

1 INTRODUCTION

In contemporary education oriented to science, technology, engineering, and mathematics — now covered by the widely popular acronym STEM — there are many courses in which numerical, modeling, and simulation problems are solved by computers running specialized, high-level numerical programming software. One of the most known software packages of that kind is MATLAB® [1]. Although there are discounted and student versions of this software, MATLAB is primarily a commercial product with a stern licensing policy. Thus, unless the special fees are paid, the students and teachers will not be able to use this software on their computers, outside the classrooms. This notably limits the use of MATLAB in the educational process.

The answer to the high-cost and licensing renewal problems with a commercial product can be in turning to free and open source substitutes, such as Scilab, GNU Octave, Sage FreeMat, and Maxima [2, 3, 4, 5, 6]. However, they are all mainly considered as the "MATLAB alternatives". Their functionality and performance are standardly compared to the functionality and performance of MATLAB, today's undisputed standard and the leader in the field [7].

Having mentioned that, the question that many lecturers, scientists, and professionals raise is what to choose. Is it better to stick to the proven though expensive commercial tool with its standard syntax and superior performance or to use Scilab and other similar products with possibly different syntax and less than superior performance — just for the sake of their unrestricted use and financial savings?

Without pretensions to answer this ubiquitous question unanimously, this article aims to provide an objective performance comparison of the commercial MATLAB and the free, open-source Scilab, and thus help the reader to answer the question herself.

After presenting the motivation for this paper, here we briefly describe its further contents. The second section begins with some basic information about the two compared tools: MATLAB as the educational and industry standard, and Scilab as its alternative. The section continues with a description of the benchmarking test environment. In the third section, we outline the most important features of MATLAB and Scilab programming languages and a few useful recommendations for translating the source code from one language to another. We start the fourth section with a short survey of the previous research on MATLAB and Scilab performances and then carry on with the description of our benchmarking methodology and the original benchmarks introduced in this work. Section five presents and comments on the obtained execution times of the implemented benchmarks and compares the performances of both calculation tools. The last, sixth section concludes the paper with the final thoughts and remarks about possible future work.

2 MATLAB, SCILAB, AND THE TESTING PLATFORM

In this section, we present basic information about MATLAB and Scilab and describe the test platform that we used for their performance measurements.

2.1 MATLAB and Scilab

MATLAB — as it was already pointed out in the introduction — is almost consensually described as the leading numerical computing and simulation software nowadays. Since its appearance in the 1980s, it has been used and approved in many high-scale scientific and professional projects and has become the de-facto standard in many scientific and industrial areas, starting from mathematics, automatic control, digital signal processing, machine learning, numerical simulations of all kinds, system modeling, and many more. It is often the first choice of professionals and because of that, it is also used in higher-level educational institutions all over the world.

MATLAB releases are commercially available with different licensing options, including academic and student licensing [8]. Although these licenses are considered the low-cost versions, they are still not free, and this can present a

serious obstacle to recommending MATLAB to students. Furthermore, the "low-cost" versions, either for educational or professional use, may still be quite pricy for the buyers, especially in the underdeveloped parts of the world.

If the user's finances are critical or highly limited, one should consider the MATLAB alternatives as the way to equip more educational or professional workplaces, including those at home. The free open-source software alternative to MATLAB discussed in this paper is **Scilab**, available under GNU v3 license [9].

Scilab was initiated by two French institutions: INRIA (*Institut National de Recherche en Informatique et en Automatique*) and ENPC (*École Nationale des Ponts et Chaussées*), and was first released in 1994. Since 2003, it is governed by the *Scilab Consortium* and is now part of the ESI Group [10, 11]. Although it originated from a scientific and academic background, Scilab soon proved its usage in industry applications, too, and has evolved into a reputable numerically oriented modeling platform, with a good user interface and a large number of versatile simulation modules [12].

For several years, Scilab has been used at the University North in Varaždin in the courses Signals and Systems, Automatic Control, and Digital Signal Processing. Even after the purchase of MATLAB licenses for classrooms, it serves as a versatile numerical software that can be freely distributed to students.

2.2 Testing Platform and the Software Versions

All our benchmark tests were performed on the same Windows 10 64-bit computer with an Intel Core i3 5005U CPU, 8 GB of RAM memory, and a 1TB standard hard disk drive, which was running the following:

1. MATLAB Version 9.5 (R2018b), 64-bit release (2018);
2. Scilab Version 6.0.2, 64-bit release (2019).

At the time of performing the tests, which was in the first half of 2019, MATLAB 9.6 was released, but we had no official commercial license and kept using the 9.5 version. Both tools were installed with the standard setup and settings.

3 UNDERLYING PROGRAMMING LANGUAGES

MATLAB and Scilab software packages contain a development environment and a numerically oriented high-level programming language suitable for matrix calculations. The first part of this section describes the most important features of these programming languages. Since there are certain differences in the syntax of the MATLAB and Scilab programming languages, the second part of this section ends with a few useful recommendations for translating the source code from one language to another.

3.1 High-Level, Matrix-Based Programming Languages

Besides the usual programming forms and structures, like the data types, arithmetic and other operations, program flow (conditional statements and loops), functions, and all other aspects of the standard programming languages, the MATLAB and Scilab programming languages have many standard and specialized functions and features for numerical and, in general, mathematical solving of scientific and technical problems.

The basic and essentially only inherent data structure of the MATLAB and Scilab languages is the *multidimensional matrix*, implemented as a multidimensional array. The standard specializations of such an array are as follows:

1. 2-*dimensional* $N \times M$ *matrix*, which is equivalent to the common $N \times M$ matrix;
2. 1-*dimensional matrix*, equivalent to a 1-row matrix or vector with N elements, or the standard 1-dim array;
3. 0-*dimensional matrix*, or 1-component vector, equivalent to a scalar, i.e. a single numerical value.

The elements of the multidimensional arrays are weakly typed. The numbers are implicitly of the double-floating point type but can be simply declared as different integer or other standard data types.

This was designed with one aim in mind: to provide a software environment for (heavy) numerical matrix and vector computation. Thus, these languages provide the standard matrix and vector (array) operations, with elements that are both real and complex numbers, right "out of the box". There is no need for any additional programming or inclusion of any extra libraries and packages. Also, the MATLAB and Scilab languages offer many advanced and specialized functions and features for dealing with these structures — either as with the standard mathematical objects or merely as with data sets organized in that way. For example, an n-component 1-dimensional matrix, or a vector, can be used to represent a data series.

The MATLAB and Scilab languages were originally designed as interpreters. Later on, MATLAB introduced the possibility of the code compilation, as well as the concept of *just-in-time* (JIT) compilation [13]. Thus, although MATLAB documentation emphasizes the improved JIT compilation and its many benefits in the recent software versions, it also advises the users to write the "normal programming code", without forcing the compilation by any side tools, and to obey the standard rules for writing "good interpreted computer programs" [14]. Regarding that, Scilab programming language has been always clearly defined as being interpreted-only [15], which makes it especially sensitive to the used programming practice (cf. §4.2).

3.2 MATLAB and Scilab Source Code Conversion

The MATLAB programming language was copy-right-protected from its very beginning. Nevertheless, the Scilab programming language was — as it seems — designed with the aim to be as alike the MATLAB language as possible but without being a direct copy. Scilab provided M2SCI tools to convert the MATLAB code to Scilab code [16]. The list of equivalent functions in MATLAB and Scilab is also available, together with the conversion tips [17].

The M2SCI tools were not used in this paper. Namely, in the case of relatively simple programming, as was done here, the conversions from one language to the other could be done

simply in the code editor of the targeted software by using its find-replace function. Some of the replacements that were used in this work are listed in Tab. 1. By this method, our benchmark source codes were easily and effectively translated from one language to the other. Of course, the programmer who performs such translations should be acquainted with the syntax of both languages, because if some problems occur, she should be able to correct the code and effectively use the available user manuals.

4 BENCHMARKING METHODOLOGY

Because numerical computing and modeling software packages often solve difficult computational tasks, the measurement of their performance is of crucial interest to both their designers and users. After a survey of the existing MATLAB and Scilab benchmarking in the works of others, this section describes and justifies the concept of the benchmark programs designed by the authors and used for benchmarking purposes in this paper.

4.1 Benchmarking in the Works of Others

For measuring the performance of a particular software version installed on a certain computer platform, MATLAB offers the `bench` function [18]. It performs the required number of runs of the following six tasks: standard matrix calculations, solving the system of linear equations and the nonlinear differential equations, finding Fast Fourier Transform (FFT), and performing one 2-D and one 3-D animated graphics. The common result is returned in the form of the execution speed, which is inversely proportional to the execution time. Scilab uses the `bench_run()` function [19], which performs a large series of predefined tests and returns their execution time in milliseconds, along with the number of the test repetitions. For the performance comparison of two or more software packages, one must choose the same benchmark tests and run them in the same environment.

The first comparison of features and performances of numerical and mathematical software packages known to the authors of this paper is given in [20]. The report compared some commercial software, including MATLAB, and some free software tools, including Scilab.

In [21], MATLAB, Scilab, GNU Octave, and NSP (a descendant of the early, pre-Java version of Scilab) were compared by testing their performance on a set of originally proposed benchmarks given in the form of closed functions. The author documented the descriptions and source codes of these functions in Scilab, which makes this work a good ground for vendor-independent benchmarking. The obtained summarized results of all those tests showed that MATLAB was a winner. Scilab was better than or equal to MATLAB in 6 of the 28 tests (21.4%), with the best performances in the calculation of Fibonacci numbers and summation of harmonic series, where it was 1.58, i.e. 1.48 times faster than MATLAB. Of the remaining 22 tests (78.6%), Scilab had the worst behavior in the for-loop tests, where it was more than 124 times slower! The obvious reason for that was its lack of the JIT compilation (see also §5.1).

Table 1 Some basic differences between MATLAB and Scilab syntax

Syntax form	*MATLAB*	*Scilab*
Inline comments.	`%`	`//`
π, the Ludolf's number In MATLAB: a keyword, In Scilab: a constant.	`pi`	`%pi`

In Scilab, the keyword `then` appears after the condition in parenthesis. There is no `then` in MATLAB.	`if (cond.)`
	`  statement1`
	`else`
	`  statement2`
	`end;`
		`if (cond.) then`
		`  statement1`
		`else`
		`  statement2`
		`end`

The end of function.	`end`	`endfunction`
Standard writing function with several target and format options.	`fprintf`	`mprintf`
Quicksort function.	`sort`	`gsort`

MATLAB, Scilab, and GNU Octave were further compared in [22], where the author examined how their performance is influenced by the use of different versions of BLAS (Basic Linear Algebra Subprograms) library. Scilab and Octave were tested with four different versions: RefBLAS, Atlas, OpenBLAS, and Intel's MKL (*Math Kernel Library*, a versatile library, free for non-commercial use). MATLAB was tested with the MKL only. Besides the afore-mentioned benchmark set from [21], there were two more sets composed by the other authors: `poisson` [23] and `ncrunch` [20]. That work showed the expected overall supremacy of MATLAB over Scilab and GNU Octave, especially in some of the tests.

In [24], the authors give a comparative analysis of several numerical computationally-demanding tests that were performed in MATLAB, GNU Octave, Scilab, FreeMat, R, and IDL, and run on a high-performance computing facility. Their execution time measurements (some lasting even up to 20 hours!) showed dramatically decreased performance of those 2012-Scilab with the increased problem sizes. Various special aspects of MATLAB versus its alternatives were also investigated in [25, 26].

4.2 Benchmark Methodology Used in This Work

In this paper, our primary goal is to check and compare the performance of the observed computational tools on several common vector and matrix calculations, which can be found in standard scientific professional and educational use. To that aim, we have created several original benchmark functions with the following types of calculations and operations:

- basic algebra on the number series,
- basic matrix operations,
- simple matrix equations,
- signal generation and analysis,
- disk access operations, for storing and retrieving data.

Both MATLAB and Scilab, as well as other similar computing environments, are famous for using *vectorization* [27, ch. 29], in which the standard loop-based implementation of an iterative operation on a single matrix (vector)

element is substituted by the inherently optimized single operation on multiple elements of the same type. This is the core concept of *array programming* [28]. In recent decades, it can be accomplished also without the explicit parallelism (which requires multiple processors), thanks to the CPU's *vector-based* instructions, implemented on the modern general-purpose processors [29]. In this way, the *implicit parallelism* is achieved.

For example, in scalar-based languages (like C/C++), creating a vector of N randomly generated values requires a loop to generate N single random values. In contrast to that, the `rand` function, available in both tools, does it without programming on the programmer's side. Similarly, instead of finding the minimum or maximum value in a vector by using a loop, one uses the optimized `min`/`max` functions. Instead of multiplying the matrices `A` and `B` by multiplying their elements within the nested `for`-loops, one simply writes the product as `A*B`, leaving the evaluation of the code and its optimization to the underlying language. This approach not only simplifies the task for programmers but also replaces either interpreted or JIT-compiled execution of the operation on a single value with highly optimized operations on multiple values. In general, as already hinted in §3.1, using vectorization by applying the math-like syntax of vector and matrix operations is the standard good-programming practice in array programming [27, ch. 29]. The advantages of using it instead of writing the standard code will be especially important for the interpreted-only languages, as is Scilab. We shall elaborate on this in sec. 5 (cf. also [30]).

MATLAB and Scilab include many functions to visualize the numerical results, but such functions, as well as all GUI-related operations, were all turned off during the benchmark test computations. No special and advanced computing techniques, like multi-threading, multi-core, or GPU-based computing were applied to any of the implemented functions. Both compared tools support several of these techniques, but in this work, we have restricted the investigation to their basic performance.

4.3 Benchmarks Designed for This Work

For the basic calculation benchmarks, we have chosen nine different common calculation tasks. Most of them have a few internal implementation variants, which will be discussed soon. These tasks were then implemented as separate functions written in the corresponding MATLAB and Scilab programming languages. The benchmarks are listed in Tab. 2. They follow the ideas of the tasks performed by the internal MATLAB and Scilab benchmark functions mentioned in §4.1, as well as implement the tasks that the authors of this paper have encountered in solving problems in the areas mentioned at the end of §2.1, in their educational, professional, and scientific practice.

As can be seen from the table, all the implemented benchmarks have the `Nrp` parameter, which defines the number of repetitions of the whole function by an additional `for`-loop. Regarding the other parameters, the most common is `N`. It is the vector size, i.e. the number of elements in the corresponding 1-dim array, which governs both the needed memory space and the overall size of the problem that is being solved. For the square matrices, this size is N^2. For the non-square matrices, the parameter `M` is added, defining the problem size of $N \times M$. The meaning of other parameters is explained in Tab. 2.

Table 2 List of the implemented benchmarks

Abbr.	*Function name / Description*
b_1	`sumIntSquared(N, Nrp)` Sum of the squares of integers $1,\ 2,\ \ldots, N$; implemented by using the vectorization concept, and for comparison also with a `for`-loop and `while`-loop.
b_2	`mtrxGenMN(N, M, r, Nrp)` Generation of a $N \times M$ matrix populated by the floating-point elements $x_{i,j} = ij/r$, where i (j) is the row (column) index: $i = 1, 2, \ldots, N$, $j = 1, 2, \ldots, M$, and $r \in \mathbb{R}$ is a decimal num.
b_3	`mtrxGenNNPow(N, k, r, Nrp)` Generation and calculation of the k-th power of the (square) $N \times N$ matrix, initially populated by the elements $x_{i,j} = i^2/r$, where i and j are the row and column indices, respectively: $i, j = 1, 2, \ldots, N$, and $r \in \mathbb{R}$ is a decimal number.
b_4	`rndMtrxMulti(N, M, Nrp)` Multiplication of the two, $N \times M$ and $M \times N$ matrices, with randomly generated elements.

$b_{5,\mathrm{i}}$
$b_{5,\mathrm{b}}$
	`rndMtrxDivInv(N, Nrp)`
	`rndMtrxDivBack(N, Nrp)`
	Solution of the matrix equation $\mathbf{Ax} = \mathbf{b}$ for the $N \times 1$ vector $\mathbf{x}$, where $\mathbf{A}$ is a randomly generated $N \times N$ matrix, and $\mathbf{b}$ is a $N \times 1$ randomly generated vector. $b_{5,\mathrm{i}}$ finds the solution by the standard method, i.e. by using the $\mathbf{A^{-1}}$ inverse matrix, $\mathbf{x} = \mathbf{A^{-1}b}$. $b_{5,\mathrm{b}}$ finds the solution by the back-divide method (matrix operator) written as $\mathbf{x} = \mathbf{A \backslash b}$.

b_6
$b_{6,0c}$
	`rndMtrxDet(N, Nrp)`
	`rndMtrxDet0c(N, Nrp)`
	b_6 calculates the determinant of a randomly generated square matrix of dimension $N \times N$. $b_{6,0c}$ does the same, but for a matrix with a randomly chosen zero-column (Fig. 1a and 1b).

b_7	`rndSort(N, Nrp)` Sorting the N randomly generated floating point numbers, or, equivalently, sorting the vector's N elements, using the internal implementation of the quicksort algorithm.
b_8	`sigAMSinFFT(Fc, Fm, Nrp)` Generation of an AM signal with the sinus carrier and sinus modulation signal of frequencies F_c and F_m, respectively. After that, the spectrum is calculated by using FFT.
b_9	`rndMtrxSaveLoad(N, Nrp)` Disk read/write test; a randomly generated $N \times N$ matrix is first saved to a file, and then the file is reloaded to another matrix.

The calculation results of the benchmark functions are returned via two output vectors: `D`, of size `Nrp`, for the function results, and `T`, of size 2 for the time measurements. With them, the functions with p parameters (and with $p_1 = p - 1$) are declared in MATLAB and Scilab as:

function `[D, T] = bfunctName(N, Par2, …, Parp1, Nrp)` .

The return parameters are defined in the square brackets after the keyword function. The function name is followed by the list of input parameters in parenthesis. For the i-th repetition of the function, $i = 1, 2, \ldots, N_{rp}$, the function result is returned via the i-th element of vector `D`. Via the two components

of vector `T`, the overall `Ttot` execution time for the `Nrp` repetitions and the average `Tavg` time per one repetition of the benchmark routine are returned.

For some benchmarks that operate on the N-size vectors or on the $N \times M$ (N^2) matrices, besides the "default," vectorized form, we have provided the alternative, loop-form. Additionally, the same benchmarks were implemented with different data types. All this was done to investigate the influence of different factors on benchmark performance.

In the following sub-subsections, we shall first briefly discuss the implemented benchmarks and then present the source code of a selected exemplary benchmark function.

4.3.1 Description and Discussion of the Benchmarks

Benchmark b_1 sums the squares of the successive integers. It is one of the benchmarks realized in the vectorized version and in the version with the standard for- and while-loops. Furthermore, it has versions with 64-bit integer and 64-bit floating point types. Benchmark b_2 creates an $N \times M$ matrix and populates it with the elements whose values are calculated from their indices and a given constant real number as $x_{i,j} = ij/r$. Again, besides the version with vectorization, it has the loop-based version, too. Benchmark b_3 creates a square, $N \times N$ matrix, populates it with the elements whose values are calculated in a similar way as in b_2, and then, additionally, calculates the k-th power of the matrix.

Benchmarks from b_4 to b_7 create the matrices initially populated with randomly generated elements and — on top of that — perform some additional calculations. Benchmark b_4 multiplies the two generated matrices. There are two versions of benchmark b_5, which solve the linear system of equations, $\mathbf{Ax} = \mathbf{b}$, in two ways: $b_{5,i}$ by finding the inverse matrix of $\mathbf{A}$, and $b_{5,b}$ by using the back-divide method (operator) for the matrix division (cf. [31, 32]). Benchmark b_6 calculates the determinant of a matrix by applying the `det` function (valid in both tools). The standard version of the benchmark forms the matrix with the randomly generated elements. In addition to that, the variant $b_{6,0c}$ inserts a randomly chosen zero-column into the matrix, making the determinant equal to zero. Thus, one can test if the `det` function checks the existence of zero-columns before performing the standard calculation of the determinant.[1] Benchmark b_7 generates an N-size vector populated with random values and then sorts its elements by using the function `sort` (`gsort`) in MATLAB (Scilab), which performs the standard quicksort algorithm.

Benchmark b_8 illustrates a standard educational example in the field of signals and digital signal processing: the generation of an amplitude-modulated (AM) signal with the sinus carrier signal of frequency F_c, modulated (multiplied) by the sinusoidal signal of frequency F_m [33]. The Fast Fourier Transform (FFT) is applied to the obtained AM signal by using the original MATLAB and Scilab FFT functions. After that, the benchmark draws the spectra of the individual signals and the modulated signal.

Finally, benchmark b_9 checks the efficiency of the operations with the external memory, which was in our case the standard hard disk (cf. §2.2). After randomly generating an $N \times N$ matrix, it saves the matrix to the disk and then reloads it into another matrix. The original `save` and `load` functions from both tools were used.

4.3.2 Source Code of an Exemplary Benchmark

Because the presentation of the source code for all our benchmarks would be too voluminous for this paper format, here we exemplify the source code of the $b_{6,0c}$ benchmark function. It is a bit more elaborate version of b_6, with a few specifics. Its MATLAB and Scilab versions are shown in Fig. 1a and 1b. The similarity between the two codes is striking.

In lines number 1, `rndMtrxDet0c` benchmark function is declared. In lines number 2, the return parameters `D` and `T` are declared as the vectors of size `Nrp` and `2`, respectively, and their elements are set to zero. In lines number 3, the specialized stopwatch timer function `tic` starts the measurement of the execution time. Both MATLAB and Scilab include the stopwatch time functions `tic` and `toc`, which measure the elapsed time with millisecond precision [34, 35]. In our case, they will measure the time of the `Nrp` repetitions of the same task, governed by the for-loops which start at lines number 4. The rationale for measuring the total (`Ttot`) and not the particular time of each repetition was to avoid the possible errors when the measured time is shorter than the not-so-great precision of the provided stopwatch functions.

The body of the `for`-loops, in lines 5 to 19, is the actual task routine of the benchmark. Each routine starts with the declaration and initialization of the single-row and single-column vectors, `zrow` and `zcol` (lines 5 and 6). After that, the benchmark generates the random $N \times N$ square matrix `A` (lines 7). Generally, the declarations in the array programming are demanding and important benchmark parts.

The code part from lines 8 to 10 is used for zeroing the randomly selected column of the matrix `A`, assuring that (at least) one such column exists in the matrix.

The lines from 11 to 19 contain the parts of the benchmarks that calculate the matrix-`A` determinant. First, there is a check if there is at least one zero-column or one zero-row in the matrix. Although in this benchmark we did not provide the insertion of (at least) one zero-row, we still provide the check for such a row for the sake of completeness. These checks are done in lines 11 to 16. The variable `is0` serves as a logic flag that is first set to 0 (false). An additional `for` loop is started that runs the column or row indices from 1 to `N`, and checks if some of the columns, `A(:, r)`, or rows, `A(r, :)`, are equal to the previously defined zero-valued `zcol` and `zrow`. If either of these is true, the `is0` flag is set to 1 (true) and the `for`-loop is exited by the `break` in

[1] For the matrices with two or more rows or columns being equal — and because of that having the zero determinant — we have encountered a precision problem worth noticing. Namely, in some cases, especially when such matrices are very large, both MATLAB and Scilab do not return the zero result but a value of the order of magnitude of 10^{-20}. This problem probably occurs because of the accumulation of errors after many additions and subtractions of numbers with the absolute values differing less than the calculation precision. Although small, the occurrence of such results require additional checking and zeroing if they are lower than a certain threshold.

```
1  function [D, T] = rndMtrxDet0c(N, Nrp)
2     D = zeros(Nrp); T = zeros(2);
3     tic;
4     for j = 1 : Nrp
5        zrow = zeros(1, N);
6        zcol = zeros(N, 1);
7        A  = rand(N, N);
8        c0 = round(N*rand());
9        if (c0 == 0) c0 = 1; end
10       A(:, c0) = 0;
11       is0 = 0;
12       for r = 1 : N
13         if (A(:, r) == zcol) is0 = 1; end
14         if (A(r, :) == zrow) is0 = 1; end
15         if (is0 == 1) break; end
16       end
17       if (is0 == 1)  d = 0;
18        else d = det(A); end
19       D(j) = d;
20    end
21    Ttot = toc;
22    Tavg = Ttot/Nrp;
23    T(1) = Ttot;
24    T(2) = Tavg;
25 end
```

Figure 1a Source code of the benchmark $b_{6,0c}$ function in MATLAB

```
1  function [D, T] = rndMtrxDet0c(N, Nrp)
2     D = zeros(Nrp); T = zeros(2);
3     tic();
4     for j = 1 : Nrp
5        zrow = zeros(1, N);
6        zcol = zeros(N, 1);
7        A  = rand(N, N);
8        c0 = round(N*rand());
9        if (c0 == 0) then c0 = 1; end
10       A(:, c0) = 0;
11       is0 = 0;
12       for r = 1 : N
13         if (A(:, r) == zcol) then is0 = 1; end
14         if (A(r, :) == zrow) then is0 = 1; end
15         if (is0 == 1) then break; end
16       end
17       if (is0 == 1) then d = 0
18       else d = det(A); end
19       D(j) = d;
20    end
21    Ttot = toc();
22    Tavg = Ttot/Nrp;
23    T(1) = Ttot;
24    T(2) = Tavg;
25 endfunction
```

Figure 1b Source code of the benchmark $b_{6,0c}$ function in Scilab

lines number 15. In this part of the benchmark, we combine vectorization (for the comparison of rows and columns) with the standard implementation of the iterative programming structure. A careful reader will note that by adding one more conditional `break` after line 13, an unnecessary checking for the zero-row could have been avoided if a zero-column had been found. If `is0 = 1`, the determinant `d` is set to zero (lines 17). In our case, matrix `A` will always have at least one zero-column, so this will always be the case. Still, the other possibility, in which the determinant is calculated by using the `det` function (lines 18), is also accounted for.

Lines 19 finish the benchmark routine by assigning the determinant value (here `d = 0`) found in the *j*-th iteration of the outer for-loop to the *j*-th element of vector `D`.

The last part of the function starts by assigning the value returned by the end of the time-measurement function `toc` to the variable `Ttot` (lines 21). After that, the average $\bar{T}_{\mathrm{B}}$ time per one repetition of the benchmark routine is calculated and stored in the variable `Tavg` (cf. tables in sec. 5). Finally, both of these times are returned via vector `T` (lines 23 and 24).

The source code of the other benchmark functions developed in this work is available in [36].

4.3.3 Benchmarking Scripts

The benchmarking scripts for each of the two observed calculation tools include the following parts:

1. Definitions of all constants needed to run the benchmark routines and their capturing function:
   - the number N_{set} of the benchmark *measurement sets*, i.e. of the repetition of each benchmark function, in our case `Nset = 5`;
   - the number N_{rp} of the benchmark repetitions in each measurement, i.e. in each benchmark function, which is most often `Nrp = 100`;
   - the problem size (N, $N \times M$), which is taken as a value of an element from the vector `Nps` of size 5.
2. Procedure for executing the benchmark functions.
3. Procedure for storing the obtained results in a text file, suitable for additional analysis in spreadsheets or other types of software.

Fig. 2 depicts a piece of Scilab script source code for measuring the $b_{6,0c}$ benchmark performance. It results with `Nset = 5` benchmark measurement sets, one for each predefined problem size, `Nps(i)`, in the `Nps` vector. As explained in the previous sub-subsection (§4.3.2), each benchmark function executes the benchmark routine `Nrp` times and returns the total and average execution times. The shown script outputs these results to the standard output (the Scilab window). To store the results to a text file, instead of `mprintf`, one should use the `mfprintf` function.

```
1  Nset = 5;
2  Nrp  = 100;
3  Nps  = [100, 500, 1000, 2000, 4000];
4  mprintf('Repetitions=%d\n', Nrp);
5  mprintf('Function(size)\tTotal\tAvg\n');
6  for i= 1 : Nset
7    [r,t] = rndMtrxDet0c(Nps(i), Nrp);
8    mprintf('rndMtrxDet0c(%d)\t%f\t%f\n',Nps(i),t(1),t(2));
9  end
```

Figure 2 Part of Scilab script, which calls the benchmark function `rndMtrxDet0c` ($b_{6,0c}$) and displays the obtained results for `Nset` benchmark measurement sets with the problem size `Nps(i)`, `i = 1, 2, … , Nset`.

5 MATLAB AND SCILAB BENCHMARK PERFORMANCE

In this section, we present the results of the benchmark execution times measurements performed in MATLAB and Scilab and discuss them.

In modern, multitasking operation systems, there are a lot of background processes that cannot be controlled by the user and whose running may interfere with the execution of the benchmark functions. Indeed, when analyzing the obtained results, there were sporadic cases of enlargements of the execution times of some benchmarks. To exclude that kind of result deviation, we were selecting the three best out of the five performed time measurements and took their mean value as the indicator of the benchmark performance. The benchmarks described in the previous section were grouped according to the division in §4.2, and the results of their benchmark execution times are presented in the following subsections.

5.1 Basic Calculation Benchmarks ($b_1 - b_3$)

Tab. 3 presents the average execution times of the basic vector and matrix calculations in benchmarks b_1 and b_2, and slightly more complex operations in b_3 (cf. Tab. 2). For these benchmarks, there are versions with and without the use of vectorization. Additionally, for b_3, the execution times for the k-th matrix power are given with $k = 3$ and $k = 11$.

By observing the obtained results, we can easily note that the execution times for both of the calculation tools are proportional to the total problem size, i.e. to the number of elements in the vectors or matrices. Next, if we focus on comparing the benchmark versions with and without vectorization, we can note that—for benchmark b_1—MATLAB gives surprisingly better results for the version without vectorization! The respective performance, for the problem size $N = 1 \times 10^6$ ($N = 4 \times 10^6$), is even 7.95 (8.11) times better than for the corresponding vectorized versions. It must be that the simplicity of this benchmark operation performed on the number series in a vector, combined with the effective MATLAB JIT compiler, results in a very efficient code that largely surpasses vectorization. Anyhow, this peculiarity does deserve additional investigation.

For the remaining two benchmarks, MATLAB versions with vectorization outperform those without it, but still not by much. For b_2 with $N^2 = 1 \times 10^6$ ($N^2 = 4 \times 10^6$), the vectorized benchmarks are 49.3% (82.2%) faster than the non-vectorized. However, for b_3, this superiority is severely diminished. With $k = 3$ and $N^2 = 1 \times 10^6$ ($N^2 = 4 \times 10^6$), vectorization improves performance for only 3.1% (2.0%). For $k = 11$ and the same problem size(s), the improvement is only 1.4% (1.1%). Again, we may attribute this to the effective JIT compilation.

On the other hand, in Scilab, the benchmarks with vectorization consistently and hugely outperform those without it. The cause of that is the strictly interpreted programming code in Scilab, without any sort of compilation, which leads to the very bad performance of the code with programmed loops (§4.2). Thus, the vectorized versions of

Table 3 Average $\overline{T}_B$ execution times of the basic calculation benchmarks, $b_1 - b_3$. The averages are calculated from the 3 best out of the total of 5 sets of measurements, each with `Nrp = 100` repetitions of every benchmark.

Benchmark		*Problem size*	$\overline{T}_B$/ms		$\overline{T}_{B,Sci} / \overline{T}_{B,MAT}$
Abbr.	*Variant*	N or $N \times M$	*MATLAB*	*Scilab*	
b_1	Vectoriz.	1×10^6	12.25	22.46	1.83
		4×10^6	49.15	92.35	1.88
	No vectoriz. (`for`-loop)	1×10^6	1.54	1239.72	805.01
		4×10^6	6.06	4956.01	818.82
b_2	Vectoriz.	$1{\times}10^3 \times 1{\times}10^3$	7.93	25.04	3.16
		$2{\times}10^3 \times 2{\times}10^3$	31.18	101.00	3.24
	No vectoriz. (`for`-loop)	$1{\times}10^3 \times 1{\times}10^3$	11.84	1910.73	161.38
		$2{\times}10^3 \times 2{\times}10^3$	56.81	7668.45	134.98
b_3	Vectoriz. ($k = 3$)	$1{\times}10^3 \times 1{\times}10^3$	109.64	191.86	1.75
		$2{\times}10^3 \times 2{\times}10^3$	748.54	1185.42	1.58
	Vectoriz. ($k = 11$)	$1{\times}10^3 \times 1{\times}10^3$	263.50	325.96	1.24
		$2{\times}10^3 \times 2{\times}10^3$	1766.77	2151.14	1.22
	No vectoriz. ($k = 3$)	$1{\times}10^3 \times 1{\times}10^3$	113.06	2180.06	19.28
		$2{\times}10^3 \times 2{\times}10^3$	763.21	8789.01	11.52
	No vectoriz. ($k = 11$)	$1{\times}10^3 \times 1{\times}10^3$	267.30	2263.58	8.47
		$2{\times}10^3 \times 2{\times}10^3$	1785.77	9736.38	5.45
$\sum$*(vectr. only):*		20×10^6	2988.96	4095.23	**1.37**
$\sum$*(non-vectr. only):*		20×10^6	3005.59	38743.94	12.89
$\sum$*(all):*		40×10^6	5994.55	42839.17	7.15

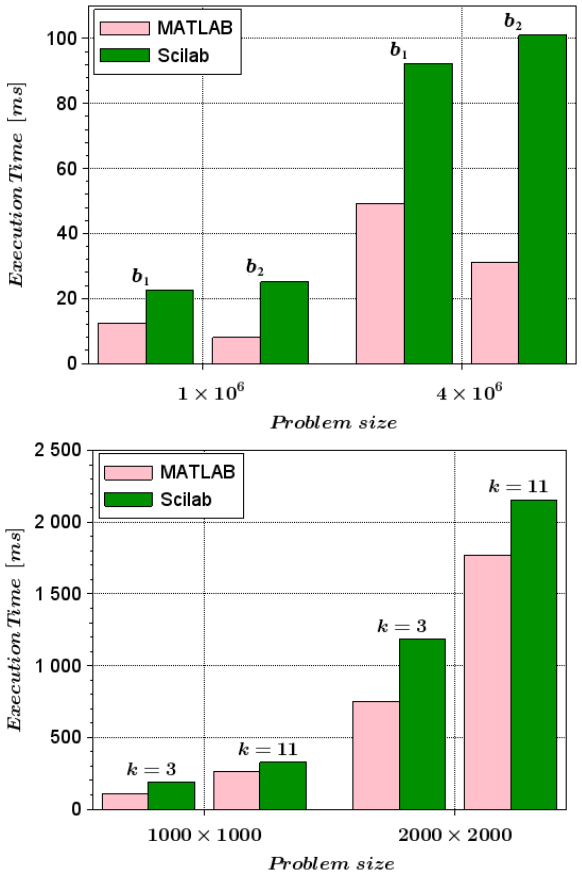

Figure 3 Graphical presentation of the benchmark execution times in MATLAB and Scilab: b_1 and b_2 (up) and b_3 (down). Shorter is better.

the benchmarks with $N^2 = 1 \times 10^6$ ($N^2 = 4 \times 10^6$) are faster than the non-vectorized ones as follows: for b_1, 55.2 (53.7) times; for b_2, 76.3 (75.9) times; for b_3, with $k = 3$, 11.4 (7.41) times, and with $k = 11$, 6.9 (4.5) times. It turns out the efficiency of vectorization highly depends on the character of the benchmark calculations.

As for the comparison of the MATLAB and Scilab benchmark performances, the last column in Tab. 3 shows that MATLAB is better in all of them and — as discussed in the previous paragraph — greatly superior in all cases without vectorization. Furthermore, it is interesting to note that Scilab vectorized benchmark versions perform worse than the corresponding MATLAB vectorized or non-vectorized versions of the same problem size.

The last three rows in Tab. 3 show the cumulative execution times of all the benchmarks and their ratio in the last column. Overall, MATLAB is faster than Scilab by more than 7 times. Even more, it is faster by almost 13 times when observing the non-vectorized benchmark routines only.

On the other hand, for the benchmark routines with vectorization — which is, after all, the recommended form of programming code in this sort of languages — Scilab shows very good performance. It is slower than MATLAB from only 1.22 for b_3 up to 3.24 for b_2. Overall, it is 37% slower than MATLAB, which can be considered an excellent performance. Fig. 3 shows these relations graphically.

5.2 Matrix Calculation and Sorting Benchmarks ($b_4 - b_7$)

Tab. 4 shows the performance of the benchmarks b_4 to b_7, all with vectorized versions only. An interesting thing to note is that the execution times are only nearly linearly proportional to the problem size, and not dependent only on it. In b_4, the four total problem sizes are 1×10^6, 2×10^6, 2×10^6, and 4×10^6, and the corresponding execution times in MATLAB (Scilab) expressed relative to the first problem size go as: 1.00 (1.00), 1.91 (1.91), 2.88 (2.73), and 5.49 (5.11). The matrices in the second and third cases have the same number of elements, but in the latter case, there is a slowdown of 51% for MATLAB and 43% for Scilab. The possible cause of this is the double number of rows ($N = 2M$) in the matrix in the third case (the first of the two matrices in the multiplication), which are half as long as the rows in the second case. Recalling the matrix multiplication rules, this means that there are twice as many element-multiplication "chains" that are half as long as in the previous case, possibly causing the less efficient multiplication. Still, the proportion of this slowdown is quite surprising.

Comparing the performance of the two tools, MATLAB is again faster in all proposed benchmarks, but Scilab performs very well, especially in benchmarks b_6, $b_{5,b}$, and b_4 (Fig. 4). In them, Scilab lags by a factor of only 1.14 – 1.29. It performs much worse in the quicksort (b_7), especially when the numbers are not rounded to integers (Fig. 5).

Both tools solve the matrix equation (b_5) faster by using the back-divide algorithm ($b_{5,b}$) than by their internal function ($b_{5,i}$), suggesting its possible improvement (Fig. 4).

Table 4 Average $\bar{T}_B$ execution times of the benchmarks $b_4 - b_7$ (cf. Tab. 3)

Benchmark		***Problem size***	$\bar{T}_B$ / ms		$\bar{T}_{B,Sci} / \bar{T}_{B,MAT}$
Abbr.	*Variant*	$N \times M$ or N	*MATLAB*	*Scilab*	
b_4	Vectoriz.	$1\times10^3 \times 1\times10^3$	94.06	121.42	1.29
		$1\times10^3 \times 2\times10^3$	179.53	232.20	1.29
		$2\times10^3 \times 1\times10^3$	270.68	331.25	1.22
		$2\times10^3 \times 2\times10^3$	516.78	620.04	1.20
$b_{5,i}$	Vectoriz., invrs. mtrx.	$1\times10^3 \times 1\times10^3$	106.68	193.50	1.81
		$2\times10^3 \times 2\times10^3$	580.96	1390.01	2.39
$b_{5,b}$	Vectoriz., back-divide	$1\times10^3 \times 1\times10^3$	66.31	81.81	1.23
		$2\times10^3 \times 2\times10^3$	307.17	373.84	1.22
b_6	Vectoriz.	$1\times10^3 \times 1\times10^3$	55.45	63.42	1.14
		$2\times10^3 \times 2\times10^3$	267.98	316.94	1.18
$b_{6,0c}$	Vectoriz.	$1\times10^3 \times 1\times10^3$	23.43	48.08	2.05
		$2\times10^3 \times 2\times10^3$	96.70	185.37	1.92
b_7	(Quicksort)	1×10^6	101.61	533.04	5.25
		2×10^6	212.34	1063.18	5.01
	With. round.	2×10^6	173.94	601.06	3.46
$\sum$*(all, except b_7):*		20×10^6	2565.73	3957.88	**1.54**
$\sum$*(all):*		40×10^6	3053.62	6155.16	2.02

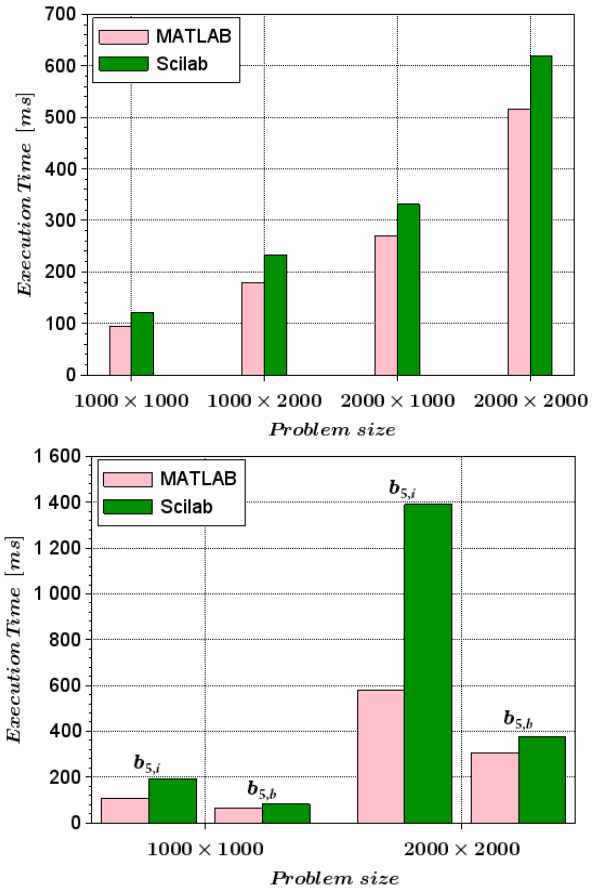

Figure 4 Performance of the benchmarks b_4 (up) and b_5 (down)

Overall, for these benchmarks, Scilab is two (2) times slower, but if the sorting is excluded, it is only about one and a half times (1.5) slower than MATLAB. For the relatively short absolute times, this is again a very good performance.

5.3 Signal Generation and DSP, Disk File Manipulation (b_8, b_9) and the Benchmarks Overall

Tab. 5 shows the performance of the benchmark b_8. The changes of the parameters influence the execution times rather unexpectedly. Furthermore, the enlargement of `Nrp` for ten times should not influence $\bar{T}_\mathrm{B}$. Of course, this illustrates how too small `Nrp` values may result in too short times, which are below the time measurement accuracy, and thus produce wrong measurements (grayed out in the table). In other words, we ought to observe the times obtained for `Nrp = 50` only. Here, MATLAB is superior again, but Scilab performs satisfactory, especially if one considers that the whole job was done in a few milliseconds.

The performance of our last benchmarks, for the disk file manipulation (b_9), is visible in Tab. 6 (cf. also §2.2 and Tab. 2). To reduce the number of disk operations, the number of repetitions was decreased to 10. Surprisingly, in these operations, Scilab is faster than MATLAB! The only and very slight exception is for the load of 100×100 matrices. Moreover, this advantage is better for larger files, where the Scilab save and load operations are roughly 6 up to 15 times faster than those of MATLAB.

Tab 7 contains the cumulative benchmark execution times for the vectorized benchmarks from Tab. 3, 4, and 5 & 6. To reduce the influence of the disk operations, the sums for b_8 and b_9 were multiplied with the weight factor of 0.1. By that, Scilab is overall only 57% slower than MATLAB, and without that, it would be equal (faster by 0.005%)!

6 CONCLUSION

In this research, the primary goal was to compare the performance of the two well-known specialized computing software environments, MATLAB and Scilab, on a set of purposefully created benchmarks that resemble the often-used vector and matrix calculations in the professional, scientific and educational domain (sec. 4).

The benchmark execution times, presented in sec. 5, generally show the supremacy of MATLAB that was known from the previous works of others (cf. §4.1). However, after excluding the bad Scilab performance in the non-vectorized benchmarks due to its interpreted-only nature, we showed that this popular freeware can compete well with its highly-commercial rival in several tasks. For the benchmarks b_3 (with $k = 11$), b_4, $b_{5,b}$, and b_6 (the basic version), it lags behind MATLAB for as little as 10% – 30%; for the benchmarks b_1, b_3 (with $k = 3$), $b_{5,i}$, and $b_{6,0c}$ it is slower for a factor from approximately 1.5 to 2.4. The cumulative results in Tab 7 — our version of `bench` function — show this clearly, and qualify Scilab as a serious choice for several needs.

Even for the benchmarks in which Scilab performed significantly worse than MATLAB, like those in b_2 (the k-th matrix power) and b_7 (quicksort), the fact that it was slower than MATLAB for the factor of approximately 3 up to 5, should be considered in the context of the overall very short absolute times of these calculations. The benchmarks in b_2 perform calculations on 1×10^6 (4×10^6) array elements in around 25 ms (100 ms), and the benchmarks in b_7 sort 1×10^6 (2×10^6) elements in 0.53 s (1.1 s, i.e. 0.6 s for the numbers rounded to integers). Obviously, such performance, though worse than that of MATLAB, will satisfy not only educational but also many demanding professional and scientific requirements.

Of the few surprises from this research, the first one concerns MATLAB: benchmark b_1 with vectorization performed significantly slower than the one without it, as

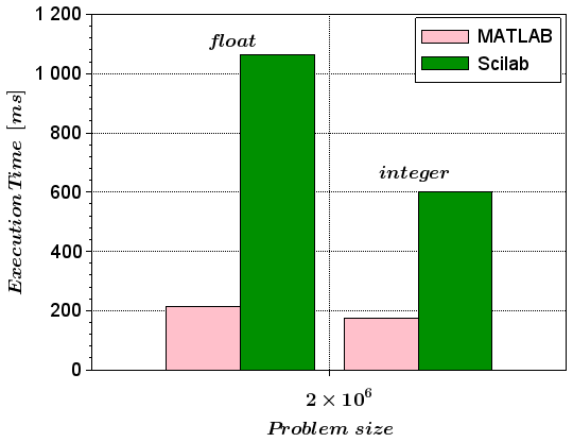

Figure 5 Performance of the benchmark b_7 (quicksort)

Table 5 Average $\bar{T}_\mathrm{B}$ execution times of the benchmark b_8, for the generation and FFT analysis of the AM signal with different F_c and F_m parameters and the variable N_{rp} parameter. The rest of the markings are as in Tab. 3.

Benchmark		*(Probl. size)*	$\bar{T}_\mathrm{B}$/ms		$\bar{T}_\mathrm{B,Sci}/\bar{T}_\mathrm{B,MAT}$
Abbr.	*Par.* (F_c, F_m)/Hz	N_{rp}	*MATLAB*	*Scilab*	
b_8	1000, 100	5	0.12	1.43	11.92
		50	0.75	2.53	3.37
	4000, 100	5	0.20	0.59	2.95
		50	2.56	7.48	2.92
		∑ ($N_{rp} = 50$ *only):*	3.31	10.01	**3.02**

Table 6 Average $\bar{T}_\mathrm{B}$ execution times of the benchmark b_9, for the file-handling save and load disk operations, with `Nrp = 10`.

Benchmark		***Problem size***	$\bar{T}_\mathrm{B}$/ms		$\bar{T}_\mathrm{B,Sci}/\bar{T}_\mathrm{B,MAT}$
Abbr.	*Operation*	$N \times M$	*MATLAB*	*Scilab*	
b_9	Save	$1{\times}10^2 \times 1{\times}10^2$	28.55	20.72	0.73
	Load	– ‖ –	1.80	1.85	1.03
	Save	$1{\times}10^3 \times 1{\times}10^3$	387.80	24.18	0.06
	Load	– ‖ –	99.60	14.56	0.15
	Save	$3{\times}10^3 \times 3{\times}10^3$	3 406.63	445.87	0.13
	Load	– ‖ –	888.86	91.02	0.10
	∑ *(all):*	20.02×10^6	4 813.24	598.20	**0.12**

Table 7 Cumulative execution times for benchmarks with vectorization

Benchmrks. *(with vectrz.)*	***Wght.*** w_{b_i}	***Tot. probl. size*** $\sum(N, N \times M)$	$\sum \bar{T}_\mathrm{B}$/ms		$\bar{T}_\mathrm{B,Sci}/\bar{T}_\mathrm{B,MAT}$
			MATLAB	*Scilab*	
$b_1 - b_3$	1.0	20.0×10^6	2 988.96	4 095.23	1.37
$b_4 - b_7$	1.0	34.0×10^6	3 053.62	6 155.16	2.02
b_8, b_9	0.1	2.0×10^6	481.66	60.82	0.13
	∑:	56.0×10^6	6 524.24	10 311.21	**1.58**

commented in §5.1. The second is the rather large benchmark b_4 slowdown (≈50%) in both tools when the first, non-square matrix in multiplication has the same number of elements but twice as many rows as columns (§5.2). Lastly, the third is that Scilab outperforms MATLAB in the disk file operations several times, especially for the (very) large files (§5.3). These, and a few other observations deserve additional investigation. Furthermore, the inclusion of multiprocessing and distributed computing in these software environments present challenging topics for future work.

Finally, what can we suggest to the reader? The winner of this race is obvious and could have been predicted even before this research (cf. §4.1). If the ultimate speed and the compatibility of the programming code with the industry-standard language are a must, and if the finances for the initial and yearly renewal of the licenses are not an issue, MATLAB is a sure pick. However, if the developers are either writing their code from scratch or are willing to adapt it by performing straightforward changes, this paper shows that Scilab is an excellent calculation environment that will in most cases perform either almost as well as MATLAB or rather close. It will even manipulate the disk files faster than its big rival, and — of course — it will cost you nothing!

7 REFERENCES

[1] MathWorks Inc., MATLAB: https://www.mathworks.com/products/matlab.html

[2] Scilab Home Page, https://www.scilab.org

[3] GNU Octave, https://www.gnu.org/software/octave/index

[4] SageMath, https://www.sagemath.org

[5] FreeMat, http://freemat.sourceforge.net

[6] Maxima, https://maxima.sourceforge.io

[7] Almeida, E. S., Medeiros, A. C., & Frery, A. C. (2012). How good are MATLAB, Octave and Scilab for computational modelling? *Computational & Applied Mathematics, 31*(3), 523-538. https://doi.org/10.1590/S1807-03022012000300005

[8] MathWorks, Pricing and Licensing — MATLAB & Simulink, https://www. mathworks.com/pricing-licensing.html

[9] Scilab Open Source, https://www.scilab.org/about/scilab-open-source-software

[10] Scilab History, https://www.scilab.org/about/company/history

[11] ESI Group, https://www.esi-group.com

[12] Scilab, Use cases, https://www.scilab.org/use-cases

[13] MathWorks, MATLAB Execution Engine, https://www.mathworks.com/products/matlab/matlab-execution-engine.html

[14] MathWorks, Techniques to Improve Performance, https://www.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html

[15] Scilab Wiki, Introduction, https://wiki.scilab.org/Introduction

[16] Scilab Help, About M2SCI tools, https://help.scilab.org/docs/5.3.3/en_US/About_M2SCI_tools.html

[17] Scilab Help, MATLAB to Scilab Conversion Tips, https://help.scilab.org/docs/5.3.3/en_US/section_c592a9ecd0ed2b4d08f8a4de718ee9aa.html

[18] MathWorks Help, bench – MATLAB benchmark, https://mathworks.com/help/ matlab/ref/bench.html

[19] Scilab Help, bench_run, https://help.scilab.org/docs/6.1.1/en_US/bench_run.html

[20] Steinhaus, S. (2008). Comparison of Mathematical Programs for Data Analysis, Munich. Retr.: https://www.additive-net.de/images/software/wolfram/publicon/downloads/numbercrunch5.pdf

[21] Pinçon, B. (2022). Quelques tests de rapidité entre différents logiciels matriciels. University of Lorraine, Retrieved from https://cermics.enpc.fr/~jpc/scilab-gtk-tiddly/bench.pdf

[22] Baudin, R. (2016). Run time comparison of MATLAB, Scilab and GNU Octave on various benchmark programs, Retrieved from http://roland65.free.fr/benchmarks/benchmarks-0.2.pdf

[23] Sharma, N. & Gobbert, M. K. (2010). A comparative evaluation of MATLAB, Octave, FreeMat, and Scilab for Research and Teaching. Department of Mathematics and Statistics, University of Maryland, Retr. from https://userpages.umbc.edu/~gobbert/papers/SharmaGobbertTR2010.pdf

[24] Coman E., Brewster, M. W., Popuri, S. K., Raim, A. M., & Gobbert, M. K. (2012). A Comparative Evaluation of Matlab, Octave, FreeMat, Scilab and R on Tara, Retrieved from http://profs.scienze.univr.it/~caliari/pdf/octave.pdf

[25] Shaukat, K., Tahir, F., Iqbal, U., & Ajmad S. (2018), A Comparative Study of Numerical Analysis Packages. *International Journal of Computer Theory and Engineering, 10* (3), 67-72. https://doi.org/10.7763/IJCTE.2018.V10.1201

[26] Leros, A., Andreatos, A., & Zagorianos A. (2010). Matlab — Octave science and engineering benchmarking and comparison. Proc. of the 14th WSEAS inter. conf. on Computers (II) 746-754. https://dl.acm.org/doi/10.5555/1984366.1984421

[27] MathWorks (1984-2022). *MATLAB Programming Fundamentals (R2022a)*. Natic, MA. MathWorks Inc. Retr. from: https://www.mathworks.com/help/pdf_doc/matlab/matlab_prog.pdf

[28] Wkp. article: Array Programming; https://en.wikipedia.org

[29] Intel, Vectorization Basics for Intel® Architecture Processors.

[30] Mikac, M., Horvatić, M., & Mikac, V. (2020). Using vectorized calculations in Scilab to improve performances of interpreted environment. *INTED2020 Proceedings – the 14th International Technology, Education and Develop. Conf.*, Valencia, Spain, 2127-2136. https://doi.org/10.21125/ inted.2020.0664

[31] MathWorks Help, mldivide, \, MATLAB, https://www.mathworks.com/help/matlab/ref/mldivide.html

[32] Scilab Help, backslash, https://help.scilab.org/docs/6.0.2/en_US/backslash.html

[33] Mikac, M. & Horvatić, M. (2020). Using Open-Source Numerical Computation Software in Education — Basic Performance Comparison and Lab Examples. *EDULEARN20 Proc. – The 12th International Conf. on Education and New Learning Technologies*, 2319-2327. https://doi.org/10.21125/edulearn.2020.0714

[34] MathWorks Help, Start stopwatch time, MATLAB, https://www.mathworks.com/help/matlab/ref/tic.html

[35] Scilab Help, tic – Start a stopwatch time, https://help.scilab.org/doc/6.0.2/en_US/tic.html

[36] Mikac, M., Logožar, R., & Horvatić M. (2022). Scilab vs. MATLAB, The benchmark functions source-code repository: https://papers.unin.com.hr/matscilab2022.html

Authors' contacts:

Matija Mikac, M.Sc.E.E.
(Corresponding author)
University North,
Jurja Križanića 31b, 42000 Varaždin, Croatia
matija.mikac@unin.hr

Robert Logožar, Ph.D. C.S.
University North,
Jurja Križanića 31b, 42000 Varaždin, Croatia
robert.logozar@unin.hr

Miroslav Horvatić, M.E.E.
University North,
Jurja Križanića 31b, 42000 Varaždin, Croatia
miroslav.horvatic@unin.hr