Advances in the Development of a High-Level Matrix Language

Luis Frank[*] Guillermo Frank[†]

Abstract
MINI is a computer program (written entirely in standard C) designed to solve matrix operations using scripts whose syntax practically emulates the traditional mathematical notation. The comparison of MINIs first version against equivalent software showed that MINI had a higher computing speed than that of other well-known software, at least for solving linear systems or for inverting large matrices. However, MINI's first version lacked many features of other popular software such as the possibility to operate with blocks of matrices or to extract eigenvalues and eigenvectors. In the paper that follows we describe how such possibilities were incorporated into a new release of MINI but without losing the original minimalist philosophy of the program. We also compare the speed and accuracy of this new version against that of leading software for scientific computation. Finally we show that even with the new additions MINI outperforms other software justifying therefore the development of new software for matrix operations.

Key Words: Matrix Language, Software, Linear Algebra

1. Introduction

MINI is an open-source computer program designed to solve matrix operations by means of scripts whose syntax resembles the handwritten mathematical notation. The program was written entirely in C language and conceived under a minimalist philosophy. To that end, the built-in functions were limited to those strictly necessary to interpret simple matrix operations (summation, multiplication and transposition) and to solve linear systems of the form $\mathbf{Ax} = \mathbf{b}$, which in turn allows the inversion of matrices. As a result a first version of MINI (hereinafter called version 0.0) was released in 2014 and presented at 2014 Joint Statistical Meetings (Frank G. and L. Frank, 2014a) together with a comparison of it's performance against other popular software for scientific computation. That version, although faster than the competing software, lacked some functions, e.g. the selection of blocks of elements within matrices, necessary for many algorithms that involve matrix manipulation.

2. Objectives

The aim of this paper is to present an improved version of MINI (version 1.0), this time compiled on two operating systems, and to evaluate its efficiency when large scripts are executed. A brief description of MINI's new built-in functions will be given in the next section, as well as an example script that decomposes a non-singular matrix $\mathbf{A}$ into the factor-matrices $\mathbf{Q}$ and $\mathbf{R}$ following a standard numerical procedure. The example is intended to explain the use of MINI's new features and the general syntax of scripts. The QR-decomposition script will then be used to compute the singular values of $\mathbf{A}$ through a larger and more complex script, which we shall use to compare MINI's performance with other softwares for scientific computation. Note that the proposed comparison is clearly

[*]University of Buenos Aires, School of Agriculture, Av. San Martín 4453, C1417DSE Buenos Aires, Argentina.
[†]University of Buenos Aires, Physics Department, Intendente Güiraldes 2160 - Pabellón I, Ciudad Universitaria, C1428EGA, Buenos Aires, Argentina.

unfavorable for MINI, as most of the programs perform the SVD-decomposition through built-in functions instead of scripts. Finally, we shall discuss the findings and suggest ideas for the development of MINI.

3. Recent improvements in MINI

New Built-in Functions

The functions listed in table 1 are available in the current version of MINI (version 1.0). The basic operations (addition, subtraction, multiplication, inverse, transpose, etc.) mentioned in the table were already available in the version 0.0 of the program and therefore do not require further explanation. The novelty of version 1.0 are three different kinds functions designed to ease the handling of matrices and blocks within matrices. These functions are of different three types depending on the type of brackets they use. The functions whose arguments are written between braces have the purpose of defining or modifying the dimensions of a matrix. Those written with brackets are used to manipulate blocks of elements, whereas those written with parentheses are functions in the traditional mathematical sense. Let's see each case in more detail.

In MINI's scripting language the braces have in three different purposes. First, they may be used for finding the dimensions of a matrix writing for example `size={A}`. Second, they can be used to assign new dimensions to an already defined matrix just writing `{A}=size`. And third, they can be used to create an empty matrix from an existing dimension vector. An example of this is discussed in the next subsection. It suffices to note that this last option is the only way to create matrices in MINI. The brackets are used to specify the indexes of certain elements of a matrix. The expression `B[i]` refers to the matrix that arises from taking the elements `B` whose indexes are declared in vector `i`. The indexes contained in `i` are simply the account of elements of `B` along the columns first and then along the rows next. Therefore, if `B` is a 3×3 matrix, the indexes $\{1, 4, 7\}$ refer to the elements of the first column; the indexes $\{1, 5, 9\}$ refer to the diagonal elements; the indexes $\{7, 8, 9\}$ refer to the elements of the last row, ans so on. The parentheses are used to define mathematical functions in the traditional sense. For example, the expression `ln(a)` returns the natural logarithm of the scalar `a`. This syntax is still under development so two remarks are pertinent:

(a) So far, the only built-in functions that support an argument are those listed in Table 1, i.e. the trace and the determinant for matrices, and the scalar functions provided in the libraries of standard C.

(b) The users can define their own functions via scripts with the extension `.min`. These scripts can in turn invoke other scripts provided that (i) the variables of both scripts do not overlap (remember that all variables in MINI are global) and (ii) the scripts invoked do not invoke other scripts. The `qr.min` script, to be discussed later, decomposes matrix $\mathbf{A}$ in two matrices ($\mathbf{Q}$ and $\mathbf{R}$) and may be invoked by typing `qr()`.

(c) The name of the input variables of the user-defined functions always remains fixed. In `qr()`, for example, the input variable is a square matrix called `A` which can not be called in any another way. For this reason the user-defined functions as `qr()` do not contain arguments between the parentheses, although we do not rule out that in future versions of MINI the user will be able change the name for the input variables.

(d) The compound functions ended in `.min` are executed directly from the command line of the operating system. From MINI's point of view, executing `.min` files or

Table 1: Built-in functions available in MINI version 1.0

Instruction	**Description**
`A`	display variable `A`. If `A` does not exist, nothing happens.
`A={}`	clear variable `A`. If `A` does not exist, nothing happens.
`mydata={}`	delete file `mydata`. If `mydata.txt` does not exist, nothing happens.
`a=3.1415`	assign (evaluate) the value `3.1415` to variable `a`.
`A=mydata`	load `mydata.txt` (in matrix format) into `a`. (wrong: `mydata=mydata`). See `A=A` below.
`A=A`	save variable `A` to a new file `A.txt`. Overwrites old files.
`A=A`	assign (copy) variable `A` to `A`.
`b={A}`	get size and memory position of `A` and copy it in `b` as [pos,rows,cols].
`{A}=b`	create a new variable `A` with dimensions specified by `b` =[pos,rows,cols]. If `A` exists, relocate data specified by `b` =[pos,rows,cols] to `A`. To be used with caution!
`C=B[a]`	get the elements of `B` sepecified by `a` and copy them to `C`. (`B=B[a]` is valid)
`B[a]=C`	place data in `C` into the elements of `B` sepecified by `a`. (`B[a]=B` is valid)
`b(A)`	function `b()` applied to `A`. Functions currently available: for matrices, trace `tr()` and determinant `det()`; for scalars, `sqrt()`, `exp()`, `ln()`, `cos()`, `sin()`, `acos()`, `asin()`, and `int()` (returns the integer part of a real number).
`f()`	user defined subroutine specified in `f.min` file.
`B=A′`	transpose matrix `A`.
`C=A+B`	matrix sum.
`C=A−B`	matrix difference.
`C=A*B`	matrix multiplication or scalar by matrix multiplication.
`C=A^b`	matrix `A` multiplied `b` times (`A` is a square matrix and `b` is an integer).
`c=a^b`	scalar power; `a` and `b` are real numbers.
`c=a/B`	means $\mathbf{aB}^{-1}$. $\mathbf{B}^{-1}$ obtained by gaussian elimination.
`c=A\b`	means $\mathbf{A}^{-1}\mathbf{b}$. $\mathbf{A}^{-1}$ obtained by gaussian elimination. Less efficient than `a/B`.
`i=>loop`	if `i` is non-zero, then goto label `loop`.
`a>b`	returns `1` if true or `0` otherwise. Same thing for `a<b`.
`a==b`	returns `1` if `a` is equal to `b`, or `0` otherwise.

commands written one by one on the command line is practically the same. So, to execute the function `myfunction.min` the user must type in the OS command line the following:

```
>./mini.e myfunction.min
```

The operator => is used to generate iterative cycles, as will be explained in more detail in the forthcoming section.

Improvements in MINI's Scripting Language

MINI's scripting language was originally conceived to follow the standard mathematical notation, but also some notation shortcuts widely spread "in the industry", that is in programs like MATLAB (MathWorks 2014), GNU Octave (2014), RLaB (Searle 2005, Kostrun 2014), Euler Math Toolbox (Grothmann 2014), etc. To review the similarities and differences among MINI and those programs, and to present the improvements introduced in MINI's latest version as well, let's see an example script written in MINI's matrix language for decomposing a full column rank matrix $\mathbf{A}$ into the factor matrices $\mathbf{Q}$ and $\mathbf{R}$. Recall that if $\mathbf{A}$ is a singular or non-square matrix then the QR-decomposition of $\mathbf{A}$ is not unique. The decomposition algorithm, adapted from Olver (2010, p.16), is as follows

```
$\mathbf{A}_0 = \mathbf{A}$
$\mathbf{Q} = \mathbf{I}_n$    ($\mathbf{Q}$ may also be set to $\mathbf{0}_{n\times n}$, whatever easier)
    for $j = 1, \dots, n$
        $\mathbf{Q}_j = \mathbf{A}_j / \sqrt{\mathbf{A}'_j \mathbf{A}_j}$    (if $\mathbf{A}'_j \mathbf{A}_j = 0$, stop)
        for $k = j+1, \dots, n$
            $\mathbf{A}_k = \mathbf{A}_k - \mathbf{Q}'_j \mathbf{A}_k \mathbf{Q}_j$
        end
    end
$\mathbf{R} = \mathbf{Q}' \mathbf{A}_0$
```

where $\mathbf{I}_n$ is an $n \times n$ identity matrix and the subscripts j and k in $\mathbf{Q}$ and $\mathbf{A}$ indicate the j-th and k-th columns of each matrix. Note that if $\mathbf{A}'_j \mathbf{A}_j = 0$ the procedure is forced to stop because $\mathbf{A}$ has at least two linear dependent columns.

Example 1. QR-decomposition of an $n \times n$ matrix $\mathbf{A}$ into the factor matrices $\mathbf{Q}$ and $\mathbf{R}$ following the modified Gram-Schmidt process with normalization during the algorithm. If $\mathbf{A}$ is a nonsingular matrix, $\mathbf{Q}$ is an orthogonal $n \times n$ matrix and $\mathbf{R}$ is an $n \times n$ upper triangular matrix. Otherwise, $\mathbf{Q}$ has r orthogonal columns (and $n - r$ zero-columns) and $\mathbf{R}$ has r non-zero-rows (r is the rank of $\mathbf{A}$), and the product $\mathbf{Q}'\mathbf{Q}$ is no longer equal to $\mathbf{I}_n$ although the identity $\mathbf{A} = \mathbf{QR}$ still holds, but is not unique.

```
A0=A
null=0
one=1
two=2

size={A}
n=size[two]
size[two]=one
```

```
{NULL}=size
{ONES}=size
{cumsum}=size

i=one
j=n
LOOP
   NULL[i]=null
   ONES[i]=one
   cumsum[i]=i
   i=i+one
   j=j-one
j=>LOOP
Q=NULL′
Q=Q⋆NULL

cumsum-ONES
cumsum=ans⋆n
cumsum=ans+ONES

i=cumsum
j=n
LOOPP
   x=A[i]
   x′
   x⋆ans
   sqrt(ans)
   q=x/ans
   Q[i]=q
   k=i+ONES
   h=j-one
   loop
      a=A[k]
      a′
      q⋆ans
      ans⋆q
      a-ans
      A[k]=ans
      k=k+ONES
      h=h-one
      h>null
   ans=>loop
   i=i+ONES
   j=j-one
   j>one
ans=>LOOPP
x=A[i]
x′
x⋆ans
sqrt(ans)
q=x/ans
Q[i]=q
R=Q′
R=R⋆A0
```

First, note that in the example above all scalars and matrices are assigned to a variable because MINI only allows operations among variables. The assignment may be done directly, as e.g. `one=1`, or by loading a matrix stored in a text file, as `A=A.txt` (not in the example), where the left-hand-side `A` is the newly created variable and the file in the right-hand-side is a text file with the extension `.txt` or no extension at all.

Next, we create the matrix $\mathbf{Q} = \mathbf{0}_n$. To do so, we first get the size-vector of any matrix, for example $\mathbf{A}$, writing `size={A}` between braces. So far, this is the only way to create a vector in MINI. The newly created vector `size` is a row vector in which the first element is the memory position where matrix $\mathbf{A}$ is stored, and the second and third elements are the number of rows and columns of $\mathbf{A}$. Later in the script we modify the number rows of `size` by writing `size[two]=one`, which means that the second element of `size` is replaced by the variable `one`. Once we have defined a matrix with the desired number of rows and columns, we can create an empty vector of that size, called for example `NULL`, simply writing `{NULL}=size` and then "fill it in" element by element by means of an iterative cycle to be explained below. Note that simultaneously with `NULL` we also create a row vector called `cumsum` whose elements $\{0, \ldots, (i-1)n+1, \ldots, (n-1)n+1\}$ are the indexes of the elements of the first column of a vectorized $n \times n$ matrix. As will become apparent shortly the purpose of this vector of indexes is to update the elements of the columns of $\mathbf{Q}$ and $\mathbf{A}$ using e.g. the syntax `Q[i]=ans`, where `i` is a vector of indexes and `Q[i]` the elements of $\mathbf{Q}$ listed in `i`.

Once we create the vector `NULL`, we are able to compute $\mathbf{Q}$ multiplying `NULL′` by `NULL`. Recall that MINI performs one operation at a time, so $\mathbf{Q}$ has to be computed in two steps. In the first step we transpose `NULL` and in the second we multiply the result by `NULL` to get a zeros-matrix. However, to create `NULL` we used an interative cycle that starts at `LOOP` and ends at `j=>LOOP`. This syntax indicates the program to return to the point labeled `LOOP` every time the equality $j = 0$ is not satisfied at `j=>LOOP` or to continue with the next operations otherwise.[1] So, defining j as a natural number bigger than 0 and introducing a counter `j=j-one` that updates j each time the operations within the cycle are performed, we get an iterative cycle similar to the for-cycle of MATLAB, GNU Octave, RLaB or Euler Math Toolbox. However, if instead of introducing a counter, we just set $j = 0$ after some arbitrary condition is met, then we get the traditional if-cycle of the other softwares. Besides, note that the starting label (`LOOP`) may be placed anywhere before or after the evaluation point `j=>LOOP`, so that it turns out that the operator "`=>`" also works as a kind of "`goto`" command which does not exist in structured languages. The identation on the left margin along the cycles only has exhibition purposes, but is not recommended in practice to avoid interpretation mistakes. At the end of the example script the reader may write the commands `Q=Q` and `R=R` to save the matrices $\mathbf{Q}$ and $\mathbf{R}$ in a `.txt` files. They are the opposite of `A=A.txt` at the begining of the script. How does MINI know if the user wants to load or to save a matrix? If the matrix does not exist, MINI understands that the user wants to load it. On the contrary, if the matrix already exists, MINI understands that the user wants to save it.

So far MINI allows the nesting of only one script within another. That is, it is possible to invoke a script from another script. To do so, the user should write the name of the invoked script followed of parenthesis, for example `qr()`, to call the script for QR-decomposition given above. The script to be invoked must be saved with the extension `.min`, for example `qr.min`. Recall, however, that all variables in MINI are global, so the user should check that the invoked script does not overwrite variables already in use in the main script. In the appendix we attach a script for SVD-decomposition of an arbitrary matrix $\mathbf{A}$ where we exploit the nesting of scripts. To run `svd` the proper syntax is

```
$./mini.e svd.min
```

written on the OS command line.

4. Testing MINI against other Softwares

In a previous paper Frank G. and Frank L. (2014a) showed that MINI was faster than other software for solving linear systems. Although this is a remarkable result, it is not extrapolable to more realistic situations in which, for example, it would be necessary to extract blocks from matrices, run iterative cycles repeatedly or hold large arrays in memory. Therefore we extended the comparison to the computation of the three matrices ($\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$) arising from the SVD-decomposition of a dense matrix $\mathbf{A}$. The script used to compute $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$ is the one shown in the appendix and the matrices used for comparison are the same used by Frank and Frank in 2014. Table 2 shows the time (in seconds) to decompose ten $10^3 \times 10^3$ and $10^3 \times 10^2$ matrices into singular values via built-in functions and scripts, respectively. The comparison involved 5 softwares on two operating systems.

Note that whenever possible every program was executed on Xubuntu Linux, MS Windows or on MS Windows but using the Cygwin environment. The SVD-decomposition was performed appealing to the built-in functions available in each software except in MINI, where we used only the script attached in the appendix. That is because the minimalist philosophy of MINI prioritizes writing scripts to the incorporation of functions into the source code. For a fair comparison, MINI's script was translated to the other four languages and executed on the other softwares. However, in all the programs the original 10^3 by 10^3 matrices proved too big to be handled with scripts and were replaced by 10^3 by 10^2 matrices.[2] The reader may note that Table 2 shows quite a few missing values because it was not

[1]"LOOP", "LOOPP" and "loop" are arbitrary labels.

[2]The reduced matrices were those defined by the first 10^2 columns of the original matrices.

Table 2: Time in seconds to decompose ten matrices into singular values.

	Ubuntu/ Xubuntu		MS Windows	
	Built-in	Script	Built-in	Script
Euler Math Toolbox 22.8 (2013)	–	–	436	1,127
GNU Octave 3.6.4 (2013)	366	10,675	183	3,666
MATLAB 7.8/6.1 (2009/2001)	89	843	150	725
MINI 1.0 (2015) sequential	–	10,580	–	3,571
MINI 1.0 (2015) parallel	–	1,067	–	479
RLaB 2.1 (2001)	–	–	239	–

possible to reproduce the experiment on all combinations of OS and source of functions. Besides, MINI was compiled in two different ways, to run in sequential and parallel mode.

5. Conclusions

MINI outperformed the benchmark software when ran in parallel mode on MS Windows, and was the second best on Xubuntu Linux. A detailed diagnosis on MINI's running processes revealed that 40 to 60% of its computing time was spent in the execution of loops and the assignment of variables to elements within matrices, while less than 10% of the time was used to solve linear systems with the gaussian elimination routine. The result supports our initial conjecture that the efficiency of most scientific computation software is nowadays limited mostly by the sophistication of the command interpreter rather than the built-in numerical algorithms.

The computation by means of scripts (in MATLAB, GNU Octave and Euler Math toolbox) was in general 5 to 10 times slower than by means of built-in functions, which turns out to be a serious limitation for MINI because its minimalist philosophy discourages the inclusion of complex built-in functions (for example for SVD-decomposition) into the source code.This limitation, however, may be partially compensated by parallel execution as parallel execution reduces the computing time almost to one tenth.

Regarding the OS we warn the reader that the performance of MATLAB, GNU Octave or MINI, when running on different OS, are not comparable because the OS were installed on a different computers. So no further conclusions may be drawn on this topic. It is encouraging however that MINI's *relative* performance on the Cygwin environment was similar to that of other programs compiled directly on MS Windows because this means that there is a chance to improve MINI's computing time by compiling the code directly on Windows.

In summary, the findings suggest that optimizing the compilation process and working in parallel mode can keep MINI competitive without adding new built-in functions that complicate the source code. These issues together with direct compilation on MS Windows will be explored in future. Besides, MINI still lacks a collection of (optimized) scripts that facilitate the migration of users from more developed softwares. This is another issue to be developed in the near future.

REFERENCES

Burden R and Douglas Faires L., 1998. "Análisis numérico." 6ta. Edición. Thomson Editores.
The Cygwin Project. Cygwin DLL 2.1.0. Downloadable from https://www.cygwin.com/
Grothmann R., 2014. "Euler Math Toolbox version 2014-06-26." Downloadable from http://euler.rene-grothmann.de/.
MathWorks 2014. "MATLAB R2014a." http://www.mathworks.com/products/matlab/
Frank G. and L. Frank, 2014a. "Designing a Computer Program for Matrix Operations." 2014

Frank G. and L. Frank, 2014b. “The Mini Matrix Language Project.” https://sites.google.com/site/scientificmini/

Olver P., 2010. Orthogonal Bases and the QR Algorithm. Univeristy of Minnesota. http://www.math.umn.edu/∼olver/aims_ / qr.pdf

Press W., Teukolsky S., Vetterling W. and B. Flannery, 2002. “Numerical Recipies in C: the Art of Scientific Computation.” Second Edition. Cambridge University Press.

GNU Octave, 2014. “GNU Octave 3.8.1.” http://www.gnu.org/software/GNU Octave/

Searle I., 2005. “RLaB 2.1.05 for Windows.” http://rlab.sourceforge.net/

Kostrun M., 2014. “RLaBplus 1.0.” http://rlabplus.sourceforge.net/

A. Singular Value Decomposition

Consider an $n \times p$ matrix $\mathbf{A}$, where $n \geq p$. This matrix may be factored as $\mathbf{A} = \mathbf{USV}'$ where the dimensions of $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$ are $n \times r$, $r \times r$, and $p \times r$, respectively. $\mathbf{S}$ is a diagonal matrix of r singular values, where r is the rank of $\mathbf{A}$, that is, r is the number of linearly independent columns of $\mathbf{A}$. This representation is called the “reduced form” of the SVD-decomposition of $\mathbf{A}$ because it omits the singular values equal to zero that correspond to linearly dependent columns of $\mathbf{A}$. However, most mathematical textbooks define $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$ as $n \times n$, $n \times p$, and $p \times p$ matrices, respectively. Such dimensions arise from the addition of $n - r$ zero-columns to $\mathbf{U}$, $p - r$ zero-rows and $p - r$ zero-columns to $\mathbf{S}$, and $p - r$ zero-columns to $\mathbf{V}$, whenever $\mathbf{A}$ is not of full rank. Hereinafter we shall consider only the reduced form of the SVD-decomposition, although the reader should keep in mind that this form is the same for full column rank matrices.

The singular values of $\mathbf{A}$ are always real numbers equal to the square root of the eigenvalues of the symmetric matrix $\mathbf{A}'\mathbf{A}$ and $\mathbf{V}$ is a matrix of the r eigenvectors of $\mathbf{A}'\mathbf{A}$. Therefore $\mathbf{V}$ is an orthogonal matrix, so that $\mathbf{V}'\mathbf{V} = \mathbf{I}_r$ is always verified. $\mathbf{U}$ is also an orthogonal matrix that satisfies $\mathbf{U}'\mathbf{U} = \mathbf{I}_r$. Then appealing to these properties we are able to get $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$ by computing the eigenvalues and eigenvectors of $\mathbf{A}'\mathbf{A}$ through the already described QR-algorithm . In short, the procedure is as follows:

(1) Compute the eigenvalues of $\mathbf{A}'\mathbf{A}$ with the QR-algorithm. To avoid further computational burden compute also the rank of $\mathbf{A}$, for example by counting the non-zero-rows of $\mathbf{Q}$.

(2) Compute the eigenvectors of $\mathbf{A}'\mathbf{A}$ with the QR-algorithm. If $\mathbf{A}$ is not of full (column) rank, that is $r < p$, resize $\mathbf{S}$ and $\mathbf{V}$ to the reduced form of the SVD-decomposition and proceed to the next step.

(3) Compute $\mathbf{U}$ solving column by column the linear system $(\mathbf{V}_{r\times r}\mathbf{S}'_{r\times r})\mathbf{U}' = \mathbf{A}'_{n\times r}$. Finally, return $\mathbf{U}_{n\times r}$, $\mathbf{S}_{r\times r}$ and $\mathbf{V}_{p\times r}$.

The following pseudocode summerizes the whole computation. For simplicity, we shall assume that $\mathbf{A}$ is full column rank.

```
Set

$\mathbf{G} = \mathbf{A}'\mathbf{A}$
$\mathbf{G}_0 = \mathbf{G}$
$\boldsymbol{\lambda}^{\text{OLD}} = \mathbf{1}_p$
$\mathbf{V} = \mathbf{I}_p$
$\mathbf{S} = \mathbf{0}_{p\times p}$
$\mathbf{U} = \mathbf{0}_{p\times p}$
```

First step. Compute eigenvalues and $\mathbf{S}$

```
for $h = 1 \ldots 1000$
    $\boldsymbol{\lambda}^{\text{NEW}} = \mathbf{0}_p$
    $\{\mathbf{Q}, \mathbf{R}\} = QR(\mathbf{G})$
    $\mathbf{G} = \mathbf{Q}'\mathbf{G}\mathbf{Q}$
        for $i = 1 \ldots p$
            ${\lambda_i}^{\text{NEW}} = g_{ij}\ ,\ \ \forall\, i = j$
        end
    $\boldsymbol{\delta} = \boldsymbol{\lambda}^{\text{NEW}} - \boldsymbol{\lambda}^{\text{OLD}}$
    if $|\boldsymbol{\delta}'\boldsymbol{\delta}| < 10^{-8}$ stop
    $\boldsymbol{\lambda}^{\text{OLD}} = \boldsymbol{\lambda}^{\text{NEW}}$
end
for $i = 1 \ldots p$
    $s_{ij} = \sqrt{{\lambda_i}^{\text{NEW}}},\ \ \forall\, i = j$
end
```

Second step. Compute eigenvectors and $\mathbf{V}$.

```
$\mathbf{G} = \mathbf{G}_0$
for $h = 1 \ldots 20$
    $\{\mathbf{Q}, \mathbf{R}\} = QR(\mathbf{G})$
    $\mathbf{V} = \mathbf{V}\mathbf{Q}$
    $\mathbf{G} = \mathbf{R}\mathbf{Q}$
end
```

Third step. Compute $\mathbf{U}$ and return the results.

```
$\mathbf{B} = \mathbf{V}\mathbf{S}'$
for $i = 1 \ldots n$
    $\mathbf{U}_i' = \mathbf{B}^{-}\mathbf{A}_i'$  (solution to the linear system $\mathbf{B}\mathbf{U}' = \mathbf{A}'$)
end
return $\{\mathbf{U}, \mathbf{S}, \mathbf{V}\}$
```

Example 2. SVD-decomposition of an $n \times p$ ($n \geq p$) full column rank matrix $\mathbf{A}$. The script returns the factor matrices $\mathbf{U}, \mathbf{S}, \mathbf{V}$ whose dimensions are $n \times p$, $p \times p$ and $p \times p$, respectively. At the end of the process it must be verified that $\mathbf{U}'\mathbf{U} = \mathbf{I}_p$, $\mathbf{V}'\mathbf{V} = \mathbf{I}_p$ and $\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}'$ as well.

```
null=0
one=1
two=2
three=3
delta=0.00000001
iter=20

A=mydata.txt
size={A}
n=size[two]
```

```
m=size[three]

zeros()
U=ans
S=U'
S=S*U
n=m
identity()
V=ans
n=one
```

```
ones()
q0=ans
p=m
c=m

A1=A
G=A'
G=G*A
G0=G

# [1] Eigenvalues

qq=q0
z=1000
LOOP01
   A=G
   qr()
   A=A1
   Q'
   G=ans*G
   G=G*Q

   i=one
   j=p
   loop01
      i-one
      ans*p
      ans+i
      G[ans]
      qq[i]=ans
      i=i+one
      j=j-one
   j=>loop01

   d=qq-q0
   q0=qq
   z=z-one
   d'
   d*ans
   sqrt(ans)
   ans<delta
   one-ans
   z=ans*z
z=>LOOP01

# [2] Eigenvectors

G=G0
z=iter
LOOP02
   A=G
   qr()
   A=A1
   V=V*Q
   G=R*Q
   z=z-one
z=>LOOP02

h={}
i={}
j={}
k={}
a={}
z={}

# [3] U,S,V computation

V0=V

n=c
identity()
W=ans

i=c
LOOP03
  j=c
  loop03
    k=i-one
    k=c*k
    k=k+j
    h=V[k]
    W[k]=h
  j=j-one
  j=>loop03
i=i-one
i=>LOOP03

V={}
V=W
W={}
k={}

i=one
j=c
LOOP04
   i-one
   ans*c
   ans+i
   h=ans
   qq[i]
   sqrt(ans)
   S[h]=ans
   i=i+one
   j=j-one
j=>LOOP04
B=V*S

z=n
n=one
m=c
ones()
ONES=ans
k=ONES
n=z
z=k

i=one
j=c
LOOP05
   k[i]=i
   i=i+one
   j=j-one
j=>LOOP05
h=k-ONES
h=h*c
h=h+ONES
m=ONES*c

j=n
LOOP06
   b=A[k]
   b=b'
   B\b
   U[h]=ans
   k=k+m
   h=h+ONES
   j=j-one
j=>LOOP06
U=U'

k=z
i=one
j=c
LOOP07
   k[i]=i
   ONES[i]=one
   i=i+one
   j=j-one
j=>LOOP07

j=p
LOOP08
   V0[k]
   V[k]=ans
   k=k+m
   j=j-one
j=>LOOP08

# [4] Return U(nxc),
# S(cxc), V(pxc) such
# that A=U.S.V'
```