`EPSfun`: a Matlab toolbox for the simplified topological ε–algorithms User Guide

Claude Brezinski[*] Michela Redivo–Zaglia[†]

The `EPSfun` toolbox is described in the paper

C. Brezinski and M. Redivo-Zaglia.
The simplified topological ε-algorithms: software and applications
NUMERICAL ALGORITHMS, 74 (2017), pp. 1237-1260
DOI: 10.1007/s11075-016-0238-0

This document includes some additional information, also pretty technical, which was not included there.

1 The toolbox

To install the toolbox, it is sufficient to uncompress the archive file containing the software. This creates the directory `EPSfun` and its subtree.

The main directory contains the following subdirectories:

`STEAfun`: This directory contains the functions needed for implementing the simplified topological ε-algorithms and an utility script. In particular:

```
clearSEAW - Script for clearing log files and persistent variables produced by SEAW
SEAW      - Scalar epsilon-algorithm with Wynn's particular rules
STEA1_1   - First Simplified Topological Epsilon-Algorithm (formula 1)
STEA1_2   - First Simplified Topological Epsilon-Algorithm (formula 2)
STEA1_3   - First Simplified Topological Epsilon-Algorithm (formula 3)
STEA1_4   - First Simplified Topological Epsilon-Algorithm (formula 4)
STEA2_1   - Second Simplified Topological Epsilon-Algorithm (formula 1)
STEA2_2   - Second Simplified Topological Epsilon-Algorithm (formula 2)
STEA2_3   - Second Simplified Topological Epsilon-Algorithm (formula 3)
STEA2_4   - Second Simplified Topological Epsilon-Algorithm (formula 4)
```

`TEAEfun`: This directory contains the functions for implementing the original topological ε-algorithms (TEA1, TEA2) and the vector ε-algorithm of Wynn (not needed for using the simplified algorithms and inserted only for allowing possible comparisons). It also contains an utility script.

```
clearVEAW - Script for clearing log files and persistent variables produced by VEAW
TEA1      - First Topological Epsilon-Algorithm
TEA2      - Second Topological Epsilon-Algorithm
VEAW      - Vector epsilon-algorithm with Wynn's particular rules
```

---

[*] Laboratoire Paul Painlevé, UMR CNRS 8524, UFR de Mathématiques, Université de Lille - Sciences et Technologies, 59655–Villeneuve d'Ascq cedex, France (`Claude.Brezinski@univ-lille1.fr`)

[†] Università degli Studi di Padova, Dipartimento di Matematica "Tullio Levi-Civita", Via Trieste 63, 35121–Padova, Italy (`Michela.RedivoZaglia@unipd.it`)

`templates`: This directory contains some examples of simple scripts for implementing the AM (Acceleration Method) and the RM (Restarted Method). We inserted here the files for solving the example described in the Section 4.4 of the paper.

```
AMmain_EA - Script for Acceleration Method (sequence of El-Sayed, Al-Dbiban method)
AMmain_MR - Script for Acceleration Method (sequence of Monsalve, Raydan method)
Fiter_EA  - Computes the next iterate (Method of El-Sayed and Al-Dbiban)
RMmain_EA - Script for Restarted Method (sequence of El-Sayed, Al-Dbiban method)
RMmain_MR - Script for Restarted Method (sequence of Monsalve, Raydan method)
```

`demo`: This directory contains scripts able to replicate all the examples of Section 5 of the paper by using the Acceleration Method (AM) and the Restarted Method (RM). We provided also additional functions needed by these scripts. There are also simple examples for using the scalar and vector ε-algorithms of Wynn and the original topological ε-algorithms of Brezinski (described in Sections 2.2.2, 3 and 4 of this user guide).

```
AMdemo    - Demo script for Acceleration Method
exhelp    - Describes the Examples 1 to 12 of the paper
exinit    - Initializes the Examples 1 to 12 of the paper
Fiter     - Performs the computation of the next term for Examples 1 to 12
plot_demo - Script to create figures and obtain information after errors of the demo
RMdemo    - Demo script for Restarted Method
testSEAW  - Test script for the scalar epsilon-algorithm of Wynn
testTEA12 - Test script for the topological epsilon-algorithms of Brezinski
testVEAW  - Test script for the vector epsilon-algorithm of Wynn
```

The directory `STEAfun` is the only one that must be added to the Matlab search path (either by the `addpath` command, or using the menus available in the graphical user interface) for running the demo, the template or the user scripts that use the simplified topological epsilon-algorithms, starting from any other directory.

If the user also wants to use the original topological epsilon-algorithms `TEA1` and `TEA2` or the vector epsilon-algorithm `VEAW`, the directory `TEAEfun` must also be added to the path.

To test the functions, the user must change the current directory to `EPSfun/templates` or `EPSfun/demo`. In the second case it is sufficient to execute the script `AMdemo` (for the Acceleration Method) or `RMdemo` (for the Restarted Method) and choose the example of the article that he wants to run. In the first case, for both AM and RM, two different methods for solving the example of Section 4.4 of the paper are proposed.

Full documentation for every function of the directory `STEAfun` (if added to the path) is accessible via the Matlab help command, from any current directory. The same is true for the functions contained in the directory `TEAEfun` if it was also added to the path:

`help <func name>`	for the functions in the directories inserted in the path (`STEAfun` and possibly `TEAEfun`)

All the codes are themselves extensively commented.

For the other functions in the supplementary directories, the user can change the current directory to one of them and use again the command `help` for accessing the list of the files or the documentation of one of them. For instance, after setting the current directory to `EPSfun/templates`, he can give

`help Contents`	for obtaining the list of files of the current directory, or simply
`help <func name>`	for one of the functions in the current directory

2 The scalar ε-algorithm

Let us describe the structure and the implementation of the scalar ε-algorithm of Wynn. This algorithm implements in a recursive way the transformation for scalar sequences proposed by Shanks in 1955. Let (S_n) a scalar sequence. The Shanks transformation $(S_n) \to \{(e_k(S_n))\}$ transforms a sequence into a set of sequences that can be expressed as a ratio of two determinants. The ε–algorithm of Wynn computes scalars $\varepsilon_k^{(n)}$ by using the following rules

$$\left\{ \begin{array}{lcl} \varepsilon_{-1}^{(n)} & = & 0, \quad n = 0, 1, \ldots, \\ \varepsilon_{0}^{(n)} & = & S_n, \quad n = 0, 1, \ldots, \\ \varepsilon_{k+1}^{(n)} & = & \varepsilon_{k-1}^{(n+1)} + (\varepsilon_{k}^{(n+1)} - \varepsilon_{k}^{(n)})^{-1}, \quad k, n = 0, 1, \ldots, \end{array} \right.$$

The quantities $\varepsilon_k^{(n)}$ are usually represented in a two-dimensional array, called the ε–array (see Table 1). Only the values in the *even columns*, that is the columns containing quantities with an even lower index, are interesting and directly related to the scalar Shanks transformation by $\varepsilon_{2k}^{(n)} = e_k(S_n)$. The terms of the original scalar sequence are stored in column 0. Thus, having $2k+1$ terms of a sequence in column 0, that is $\varepsilon_0^{(i)} = S_i$ for $i = 0, \ldots, 2k$ we are able to complete the ε-array up to the vertex $\varepsilon_{2k}^{(0)}$.

$$\begin{array}{cccccccc}
\varepsilon_{-1}^{(0)} = 0 & & & & & & & \\
 & \varepsilon_{0}^{(0)} = S_0 & & & & & & \\
\varepsilon_{-1}^{(1)} = 0 & & \varepsilon_{1}^{(0)} & & & & & \\
 & \varepsilon_{0}^{(1)} = S_1 & & \varepsilon_{2}^{(0)} & & & & \\
\varepsilon_{-1}^{(2)} = 0 & & \varepsilon_{1}^{(1)} & & \ddots & & & \\
\vdots & \varepsilon_{0}^{(2)} = S_2 & \vdots & & & \ddots & & \\
\vdots & \vdots & \vdots & & & & \ddots & \\
\vdots & \vdots & \vdots & & & & & \varepsilon_{2k}^{(0)} \\
\vdots & \vdots & \vdots & & & & \iddots & \\
\vdots & \vdots & \vdots & & & \iddots & & \\
\vdots & \vdots & \vdots & & \iddots & & & \\
\vdots & \varepsilon_{0}^{(2k-1)} = S_{2k-1} & \vdots & \iddots & & & & \\
\varepsilon_{-1}^{(k)} = 0 & & \varepsilon_{1}^{(2k-1)} & & & & & \\
 & \varepsilon_{0}^{(2k)} = S_{2k} & & & & & &
\end{array}$$

Table 1: The ε-array.

Remark that the column with a lower index equal to -1 only means that in the computation of column 1 we have to use the formula $\varepsilon_1^{(n)} = (\varepsilon_0^{(n+1)} - \varepsilon_0^{(n)})^{-1}, n = 0, 1, \ldots$. This is in fact made in our implementation and we don't store the values of column -1 by using this simplified formula.

Theoretical results or computational issues (in the presence of a large number of elements of the initial sequence) can suggest not to complete the entire ε-array up to its vertex, but to stop the computation when a certain even column (say `MAXCOL`) is reached and, thus, the ε–array becomes as in Table 2.

Since only the columns containing quantities with an even lower index are interesting, it is not necessary to give the values of the odd columns. Thus, in the iterative call of our functions for implementing the scalar ε-algorithm (and this is also true for the vector or topological algorithms), the even quantities of this array are given in a staircase scheme. For example, see Table 3, where the values computed and returned when `MAXCOL=4` are emphasized in a box. In this table, we omitted the column -1 since it is not used, as previously said, and we numbered the columns and the starting quantity of the ascending diagonals since it will be useful in the sequel.

As the normal rule shows, the terms involved in the computation (except for column 1 since the rule simplifies) are located at the four corners of a lozenge, as showed in Table 4. Thus, knowing $\varepsilon_{k-1}^{(n+1)}, \varepsilon_k^{(n)}$ and $\varepsilon_k^{(n+1)}$, it is possible to compute $\varepsilon_{k+1}^{(n)}$.

$$
\begin{array}{ccccccc}
\varepsilon_{-1}^{(0)} = 0 & & & & & & \\
& \varepsilon_{0}^{(0)} = S_0 & & & & & \\
\varepsilon_{-1}^{(1)} = 0 & & \varepsilon_{1}^{(0)} & & & & \\
& \varepsilon_{0}^{(1)} = S_1 & & \varepsilon_{2}^{(0)} & & & \\
\varepsilon_{-1}^{(2)} = 0 & & \varepsilon_{1}^{(1)} & & \ddots & & \\
& \varepsilon_{0}^{(2)} = S_2 & & \vdots & & \varepsilon_{\texttt{MAXCOL}}^{(0)} & \\
\vdots & \vdots & \vdots & \vdots & & & \\
\vdots & \vdots & \vdots & \vdots & & \varepsilon_{\texttt{MAXCOL}}^{(1)} & \\
\vdots & \vdots & \vdots & \vdots & & & \\
\vdots & \vdots & \vdots & \vdots & & \varepsilon_{\texttt{MAXCOL}}^{(2)} & \\
\vdots & \vdots & \vdots & \vdots & & & \\
\vdots & \varepsilon_{0}^{(2k-1)} = S_{2k-1} & \vdots & \vdots & & \varepsilon_{\texttt{MAXCOL}}^{(3)} & \\
\varepsilon_{-1}^{(k)} = 0 & & \varepsilon_{1}^{(2k-1)} & \vdots & & & \\
\vdots & \varepsilon_{0}^{(2k)} = S_{2k} & \vdots & \vdots & & \varepsilon_{\texttt{MAXCOL}}^{(4)} & \\
\vdots & \vdots & \vdots & \vdots & & \vdots &
\end{array}
$$

Table 2: The ε-array with a `MAXCOL`.

$$
\begin{array}{cccccc}
& & \text{column 1} & \text{column 2} & \text{column 3} & \text{column 4} \\
\text{diagonal 1} & \boxed{\varepsilon_{0}^{(0)}} = S_0 & & & & \\
& \downarrow & \varepsilon_{1}^{(0)} & & & \\
\text{diagonal 2} & \boxed{\varepsilon_{0}^{(1)}} = S_1 & \longrightarrow & \boxed{\varepsilon_{2}^{(0)}} & & \\
& & \varepsilon_{1}^{(1)} & \downarrow & \varepsilon_{3}^{(0)} & \\
\text{diagonal 3} & \varepsilon_{0}^{(2)} = S_2 & & \boxed{\varepsilon_{2}^{(1)}} & \longrightarrow & \boxed{\varepsilon_{4}^{(0)}} \\
& & \varepsilon_{1}^{(2)} & & \varepsilon_{3}^{(1)} & \downarrow \\
\text{diagonal 4} & \varepsilon_{0}^{(3)} = S_3 & & \varepsilon_{2}^{(2)} & & \boxed{\varepsilon_{4}^{(1)}} \\
& & \varepsilon_{1}^{(3)} & & \varepsilon_{3}^{(2)} & \downarrow \\
\text{diagonal 5} & \varepsilon_{0}^{(4)} = S_4 & & \varepsilon_{2}^{(3)} & & \boxed{\varepsilon_{4}^{(2)}} \\
& & \varepsilon_{1}^{(4)} & & \varepsilon_{3}^{(3)} & \\
\text{diagonal 6} & \varepsilon_{0}^{(5)} = S_5 & & \varepsilon_{2}^{(4)} & & \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
$$

Table 3: Values of the ε–array obtained with `MAXCOL=4` .

Of course, the easiest way to built the entire ε-array is to compute the columns one by one, starting from the column 1, and using, except for column 1, the values in the two preceding ones. But, if we want to add a new term, say S_{2k+1}, all the computations must be started again. From the point of view of storage requirements and computational effort this is of course possible with scalars, but it is a nonsense when the terms of the sequence are vectors or matrices (as in the other algorithms with a similar structure). The usual way for implementing the algorithms is a technique called *moving lozenge*, originally used by Wynn: we proceed by ascending diagonals (in the implementation, a diagonal will be a scalar vector), that is, we compute each new ascending diagonal of the ε–array by using the previous diagonal. For instance, let us assume that the ascending diagonal 2, containing $\varepsilon_0^{(1)}$ and $\varepsilon_1^{(0)}$ has already been computed and stored. We add the new term $\varepsilon_0^{(2)} = S_2$ of the sequence and, by using the diagonal 2, we are able to compute the new diagonal 3 that will contain $\varepsilon_0^{(2)}$, $\varepsilon_1^{(1)}$ and $\varepsilon_2^{(0)}$, and to store it by replacement of the previous one. Thus, for proceeding in the algorithm, only one vector containing the last computed diagonal (called `EPSSCA`) is given in input to the function `SEAW`, and the function will output the new diagonal which is needed

$$\begin{tikzcd}
 & \varepsilon_k^{(n)} \arrow[dr] & \\
\varepsilon_{k-1}^{(n+1)} \arrow[ur] \arrow[dr] & & \boxed{\varepsilon_{k+1}^{(n)}} \\
 & \varepsilon_k^{(n+1)} \arrow[ur] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \varepsilon_k^{(n)} \arrow[dr] & \\
\varepsilon_{k-1}^{(n+1)} \arrow[ur] \arrow[dr] & & \text{AUX0} \\
 & \text{AUX1} \arrow[ur] & \\
\text{AUX2} & &
\end{tikzcd}$$

Table 4: The ε–array lozenge (left) and the moving lozenge (right).

for the next call. For the scalar ε-algorithm, 3 temporary variables (called `AUX2`, `AUX1` and `AUX0`) are needed for such a kind of technique (see Table 4). The quantities $\varepsilon_{k-1}^{(n+1)}$ and $\varepsilon_k^{(n)}$ are in the preceding diagonal. By using the rule, we compute the term $\varepsilon_{k-1}^{(n+2)}$, and we store it into the temporary variable `AUX2`. We compute the term $\varepsilon_k^{(n+1)}$, and we store it into the temporary variable `AUX1`. Finally we compute the term $\varepsilon_{k+1}^{(n)}$, and we store it into the temporary variable `AUX0`. Now the term $\varepsilon_{k-1}^{(n+1)}$ is no more needed and we shift the lozenge by assigning the value `AUX2` in the diagonal element were $\varepsilon_{k-1}^{(n+1)}$ was previously stored, and we translate the lozenge by assigning

```
AUX2 = AUX1
AUX1 = AUX0
```

and this is made until the entire new diagonal has been computed.

In all our Matlab functions we proceeded in this way, with some tricks and additional local or persistent variables, for implementing the algorithms with more complicated rules, but the input/output arguments are in practice the same. This technique possesses another important advantage since, as the construction progresses, the terms of the original sequence can be given (and eventually computed) one by one. In this sense, the way in which the new term is computed can be treated as a black box and, if it is computationally expensive, we are able to compute only one term, then to proceed in the completion of the ε–array, and to continue the loop (if the user wants to execute it a fixed number of times), or to decide to stop the procedure (if a stopping criteria has been defined).

2.1 The function `SEAW`

We show in the sequel the function `SEAW` and a brief explanation of its input/output parameters. The meaning of these parameters will be clarified in the rest of this Section.

```
[EPSINIS,EPSSCA,NSING] = SEAW(EPSINIS,EPSSCA,MAXCOL,TOL,NDIGIT,IFLAG)
```

`EPSINIS`: input/output argument. In input, at each call, a new term of the scalar sequence is given. In output, we obtain only the values of the even columns in the descending staircase that can be computed with the number of terms of the initial sequence given in the successive executions of the function, up to the column `MAXCOL` for which all the terms of the column are given in the output argument one by one. The successive values stored in the output variable `EPSINIS` are highlighted in a box in Table 2.

`EPSSCA`: input/output argument. In input, this vector contains the last stored ascending diagonal and, in the output argument, the new computed ascending diagonal is returned, since it is needed for the next call.

`MAXCOL`: input even argument. It is not mandatory to construct a "complete" ε-array, in the sense that the user may want to stop at a certain even `MAXCOL` column, and proceed the scheme by computing and returning the successive values of the `MAXCOL` column. See for example the Table 2 showing how the function proceeds when `MAXCOL=4`. Remind that, for reaching the first value of the column `MAXCOL`, at least `MAXCOL+1` terms of the original sequence have to be given (and so the same number of calls of the function). If this input argument is odd (a nonsense since the interesting quantities are in even columns) an error occurs.

`TOL`: input argument used in tests for near-breakdown. If `abs(X) < TOL`, then `X` is considered to be zero, and an error occurs.

`NDIGIT`: input argument used in tests for detecting an isolated singularity and, as a consequence, for applying the particular rules. Please remark that if this value is too small, it could easily happens that the function detects non-isolated singularities and, thus, the execution is stopped with an error. Notice that the maximum number of non-isolated singularities that can be found corresponds to `MAXCOL-3` and that, when `MAXCOL<=2` the particular rules cannot be used.

`IFLAG`: input argument. When the value of the input argument is set to zero, we indicate that we want to treat a new sequence and the scheme will be restarted by using the value `EPSINIS` as its first term. As long as the input value is different from zero, or is missing, the function proceeds in the construction of the previously started ε–array. Thus, for a new application of the algorithm starting from the first value $\varepsilon_0^{(0)}$, the user must remember to call the function with `IFLAG` equal to zero.

`NSING`: output argument. There is a variable inside the function, that counts the number of non-isolated singularities and, at the end of each call, this value is returned in `NSING`. Thus, at the end of the loop, `NSING` contains the total number of this kind of singularities treated.

2.2 Tests for numerical instability of the scalar ε-algorithm

2.2.1 Particular rules

Let us explain better when and how we can treat an instability in this algorithm. Let us consider, in the ε-array, a wide lozenge involving five columns and three ascending diagonals as in Table 5 where in the middle, for simplicity, we denote the ε's quantities in the left with small and capital letters.

$$\begin{array}{ccccc} & & \varepsilon_{k-1}^{(n)} & & \\ & \varepsilon_{k-2}^{(n+1)} & & \varepsilon_{k}^{(n)} & \\ \varepsilon_{k-3}^{(n+2)} & & \varepsilon_{k-1}^{(n+1)} & & \varepsilon_{k+1}^{(n)} \\ & \varepsilon_{k-2}^{(n+2)} & & \varepsilon_{k}^{(n+1)} & \\ & & \varepsilon_{k-1}^{(n+2)} & & \end{array} \qquad \begin{array}{ccccc} & & N & & \\ & a & & b & \\ W & & C & & E \\ & e & & d & \\ & & S & & \end{array} \qquad \begin{array}{ccccc} & & N & & \\ & a \simeq \alpha & & b \simeq \alpha & \\ W & & C \simeq \infty & & E? \\ & e \simeq \alpha & & d \simeq \alpha & \\ & & S & & \end{array}$$

Table 5: Particular rule scheme.

Suppose we have already computed and stored the values of the upper diagonal containing W, a, and N. In computing the new diagonal, in particular, after computing e it happens that $e \neq a$ but are *almost equal*. Using the normal form of the algorithm we have

$$C = W + 1/(e - a).$$

But since $e \simeq a$, thus $C \simeq \infty$ (a condition often called *near-breakdown*), and so

$$b = a + 1/(C - N) \simeq a.$$

In computing the next diagonal we have

$$d = e + 1/(S - C) \simeq e \simeq a$$

and

$$E = C + 1/(d-b) \qquad \text{is undetermined.}$$

For avoiding such an instability we proceed in this way: when it happens that $a \simeq e$, we compute C and b with the normal rule but, at the same time, we compute and save persistent arrays, the quantities

$$\begin{aligned} A &= W(1-W/C)^{-1} \\ B &= N(1-N/C)^{-1}. \end{aligned}$$

Then, in the next call, we compute S and d, with the normal rules, but for computing E we use a mathematically equivalent formula, more stable, and we proceed as follows

$$\begin{aligned} D &= S(1-S/C)^{-1} \\ r &= D+B-A \\ E &= r(1+r/C)^{-1}. \end{aligned}$$

Two quantities are declared *almost equal*, that is a *singularity* occurs, when, for a fixed integer `NDIGIT` defined by the user,

$$|\varepsilon_{k-2}^{(n+2)} - \varepsilon_{k-2}^{(n+1)}|/|\varepsilon_{k-2}^{(n+1)}| < 10^{-\texttt{NDIGIT}},$$

that is, with our notations, when

$$|e-a|/|a| < 10^{-\texttt{NDIGIT}}.$$

The check on the almost equal quantities is made all along the diagonal that is in computation, thus allowing to store and treat at most `MAXCOL-3` singularities. Sometimes this condition is not sufficient enough for detecting the singularity. Thus, in the function,s more complicated tests are made. But it is unuseful to describe them in details here. The output variable `NSING` is a possibly increasing counter that indicates how many isolated singularities have been found in all the previous calls of the function `SEAW`.

This particular rule, implemented in the `SEAW` function and also, with appropriate changes, in the vector ε-algorithm of Wynn (`VEAW` function), is valid only when an *isolated singularity* occurs, that is we must have $\varepsilon_{k-2}^{(n+2)}$ almost equal to $\varepsilon_{k-2}^{(n+1)}$, but $\varepsilon_{k-2}^{(n+3)}$ is not almost equal to $\varepsilon_{k-2}^{(n+2)}$. If this last condition is not satisfied, then a *non isolated singularity* occurs and the program stops with an error. Wynn's particular rule was extended by Cordellier to the case of an arbitrary number of equal or almost equal quantities in the ε-algorithm, but this rule has not yet been implemented in this toolbox.

2.2.2 Other tests and `.log` file

The ε-algorithm implemented with the `SEAW` function, but also all the other algorithms, have rules containing a division by a scalar quantity that, for avoiding numerical instability, cannot be too small. This condition is detected by checking the scalar in the denominator by a user defined quantity called `TOL`. Then, when a denominator, say X, is $<$ `TOL`, it is considered to be zero, an error occurs and the program stops with an appropriate description.

As said before, the ε-array is built diagonal by diagonal in a loop. For checking how the construction is made, and over all, in which position the singularities have been found, during the iterative call, a `.log` file is written. For the scalar ε-algorithm the file is called `SEAW_<date>_n.log`, where `<date>` indicates the date of the run of the main script, and `n` is a counter that is automatically increased since the script can consider several constructions of an entire ε-array. This is, for example, the case of our demo scripts: `AMdemo` produces two ε-array, one for the STEA1 and one for the STEA2; `RMdemo` produces `NCY` (the number of cycles) ε-array, both for the STEA1 and for the STEA2.

In practice, each time the `IFLAG` variable is zero a new `.log` file is produced. Of course the user can wish to look at these files only when an error occurs or for investigating the singularities treated by the particular rule. Thus a special function, named `clearSEAW` without a space (`clearVEAW` for the vector ε-algorithm), is provided and inserted into the demo and template scripts for erasing the external previously created `.log` files, and for resetting the persistent variables that manage the `n` suffix of the file.

Let us present an example showing the use of the particular rule and the `.log` file it produces. Let

$$\begin{aligned} u_s &= x^s/s! \qquad s = 0, 1, \ldots \\ S_0 &= 0 \\ S_n &= \sum_{0}^{n-1} u_s \qquad n = 1, 2, \ldots \end{aligned}$$

We set $x = 2$ and by using the `SEAW` function, with `TOL=0` (in this case we will obtain two equal quantities and, thus for avoiding the function to stop with an error due to a *breakdown*, this trick ensures the use of the particular rule and the result is obtained), `NBC=5` (it represents the number of terms of the original sequence we have), `MAXCOL=4`, any value for `NDIGIT`, we are able to obtain a good approximation of the value $\varepsilon_0^{(4)} = 5$. In exact arithmetic the ε-array is

$$\begin{array}{llllll}
 & & \text{column 1} & \text{column 2} & \text{column 3} & \text{column 4} \\
\text{diagonal 1} & \varepsilon_0^{(0)} = 0 & & & & \\
 & & \varepsilon_1^{(0)} = 1 & & & \\
\text{diagonal 2} & \varepsilon_0^{(1)} = 1 & & \varepsilon_2^{(0)} = -1 & & \\
 & & \varepsilon_1^{(1)} = \frac{1}{2} & & \varepsilon_1^{(0)} = \frac{1}{2} & \\
\text{diagonal 3} & \varepsilon_0^{(2)} = 3 & & \boxed{\varepsilon_2^{(1)} = \infty} & & \boxed{\varepsilon_4^{(0)} = 5} \\
 & & \varepsilon_1^{(2)} = \frac{1}{2} & & \varepsilon_3^{(1)} = \frac{1}{2} & \\
\text{diagonal 4} & \varepsilon_0^{(3)} = 5 & & \varepsilon_2^{(2)} = 9 & & \\
 & & \varepsilon_1^{(3)} = \frac{3}{4} & & & \\
\text{diagonal 5} & \varepsilon_0^{(4)} = \frac{19}{3} & & & &
\end{array}$$

Numerically, the value obtained is $\varepsilon_4^{(0)} = 4.999999999999996$. In the sequel the `.log` file produced:

```
=== SEAW.log file ===
=== Cycle or Run 1 ===

=== Compute the diagonal 1 ===
=== Compute the diagonal 2 ===
NORMAL rule in column 1
=== Compute the diagonal 3 ===
NORMAL rule in column 1
NORMAL rule in column 2
=== Compute the diagonal 4 ===
NORMAL rule in column 1
New point of instability found in column number 2
NORMAL rule in column 2
PARTICULAR rule for computing A in column 2
NORMAL rule in column 3
PARTICULAR rule for computing B in column 3
=== Compute the diagonal 5 ===
NORMAL rule in column 1
NORMAL rule in column 2
NORMAL rule in column 3
```

```
 PARTICULAR rule for computing D in column 3
 PARTICULAR rule for computing E in column 4
```

As can be seen, the instability in computing $\varepsilon_2^{(1)}$ in the diagonal 4, due to the fact that $\varepsilon_1^{(1)} = \varepsilon_1^{(2)}$, has been detected, treated and, in the computation of $\mathtt{E} = \varepsilon_4^{(0)}$, the particular rule has been used. A script for this example has been inserted in the `demo` directory, and named `testSEAW.m`. Let us remark that, in the case of a true breakdown, the particular rule simplifies to

$$E = S + N - W.$$

By adding a perturbation of 10^{-7} to the initial terms, a near-breakdown occurs and, with `NDIGIT=15` and `TOL=1e-20`, the particular rule is applied (check what happens with `NDIGIT=20`).

2.3 A simple pseudocode for the ε-algorithm

Acceleration Method with the ε-algorithm

```
Initializations
    Set MAXCOL, TOL, NDIGIT, NBC
Computations
    IFLAG ← 0
    EPSINIS ← $S_0$
    First call of SEAW
    Output EPSINIS
    for $n = 1 : \mathrm{NBC} - 1$
        EPSINIS ← $S_n$
        Call SEAW
        Output EPSINIS
    end for $n$
    Output NSING
```

As previously explained, the output values in `EPSINIS` are returned following the staircase scheme as in Table 3. The first call (with `IFLAG=0`), for starting a new ε-array, is mandatory for setting all the variables needed in the function.

3 The vector ε-algorithm

If the terms of the sequence are vectors in $\mathbb{R}^m$, it is possible to use the vector ε–algorithm of Wynn. By defining the inverse of a vector $\mathbf{u}$ as $\mathbf{u}^{-1} = \mathbf{u}/(\mathbf{u}, \mathbf{u})$, where $(\cdot, \cdot)$ is the usual inner product, the rules are the same as those of the scalar ε–algorithm, and the scheme and the implementation by ascending diagonals are similar. A particular rule, similar to the scalar one, also exists. Even if it is not the main purpose of this toolbox, we also inserted in the directory `TEAEfun` a function, called `VEAW`, for implementing it. In practice everything described above also apply to this version that uses cell variables instead of simple scalar variables. The particular rule is used when

$$||e - a||/||a|| < 10^{-\mathtt{NDIGIT}}.$$

Other tests for numerical instability are made. In particular we check the quantity in the denominator (that is the scalar product between two vectors) with the parameter `TOL`.

The use of the function is:

```
[EPSINI,EPSVEC,NSING] = VEAW(EPSINI,EPSVEC,MAXCOL,TOL,NDIGIT,IFLAG)
```

where `EPSINI` and `EPSVEC` are the new names given to `EPSINIS` and `EPSSCA`.

A `.log` file, called `VEAW_<date>_n.log` is produced, for checking the instability locations in the ε-array, and a function `clearVEAW` is also provided. If the user wants to use this algorithm from any other directory, he must add the directory `TEAEfun` to the Matlab search path. This must be done if the user wants to run the example of script inserted in the `demo` directory, and named `testVEAW.m`, that uses the vector ε-algorithm and its particular rule. From the theory for such a sequence transformation, in column 6 all the extrapolated values $\varepsilon_6^{(i)}, \forall i$, must coincide with the zero vector. In the example we see that, with `NDIGIT=8`, the particular rule is applied twice, and the components of the resulting vectors are close to the computer precision. If `NDIGIT=10`, the particular rule is not used and the vectors in column 6 are very far from the theoretical result which is zero.

4 The topological ε-algorithms TEA1 and TEA2

The topological Shanks transformation, proposed by Brezinski in 1975, is more general since we consider sequences of elements of a topological vector space E on $\mathbb{R}$ (or $\mathbb{C}$). There exist two versions of this transformation and two different algorithms for implementing them. The idea was based on the definition of the inverse of a couple $(\mathbf{u}, \mathbf{y}) \in E \times E^*$ defined as $\mathbf{u}^{-1} = \mathbf{y}/ < \mathbf{y}, \mathbf{u} > \in E^*$ and $\mathbf{y}^{-1} = \mathbf{u}/ < \mathbf{y}, \mathbf{u} > \in E$. Both algorithms need to perform operations involving elements of the algebraic dual space E^* of E, that is the vector space of linear functionals on E. These algorithms now involve two different rules for the even lower index terms and the odd ones.

In Table 6, we recall the rules of the *first topological ε–algorithm* (TEA1) for computing the elements $\widehat{\mathbf{e}}_k(\mathbf{S}_n) \in E$ and of the *second topological ε–algorithm* (TEA2) for computing the elements $\widetilde{\mathbf{e}}_k(\mathbf{S}_n) \in E$. As in the scalar or vector cases of the ε-algorithm of Wynn, the only interesting transformed terms are the even ones and we have $\widehat{\varepsilon}_{2k}^{(n)} = \widehat{\mathbf{e}}_k(\mathbf{S}_n)$ and $\widetilde{\varepsilon}_{2k}^{(n)} = \widetilde{\mathbf{e}}_k(\mathbf{S}_n)$. No particular rules exist for these algorithms. The structure of the ε-array is always the same as in Table 1, but the relations between the terms of the array are a little bit more complicate as showed in Table 7.

TEA1 algorithm

$$\left\{\begin{array}{lll}
\widehat{\varepsilon}_{-1}^{(n)} & = \mathbf{0} \in E^*, & n = 0, 1, \ldots, \\
\widehat{\varepsilon}_{0}^{(n)} & = \mathbf{S}_n \in E, & n = 0, 1, \ldots, \\
\widehat{\varepsilon}_{2k+1}^{(n)} & = \widehat{\varepsilon}_{2k-1}^{(n+1)} + \dfrac{\mathbf{y}}{< \mathbf{y}, \widehat{\varepsilon}_{2k}^{(n+1)} - \widehat{\varepsilon}_{2k}^{(n)} >} \in E^*, & k, n = 0, 1, \ldots, \\
\widehat{\varepsilon}_{2k+2}^{(n)} & = \widehat{\varepsilon}_{2k}^{(n+1)} + \dfrac{\widehat{\varepsilon}_{2k}^{(n+1)} - \widehat{\varepsilon}_{2k}^{(n)}}{< \widehat{\varepsilon}_{2k+1}^{(n+1)} - \widehat{\varepsilon}_{2k+1}^{(n)}, \widehat{\varepsilon}_{2k}^{(n+1)} - \widehat{\varepsilon}_{2k}^{(n)} >} \in E, & k, n = 0, 1, \ldots
\end{array}\right.$$

TEA2 algorithm

$$\left\{\begin{array}{lll}
\widetilde{\varepsilon}_{-1}^{(n)} & = \mathbf{0} \in E^*, & n = 0, 1, \ldots, \\
\widetilde{\varepsilon}_{0}^{(n)} & = \mathbf{S}_n \in E, & n = 0, 1, \ldots, \\
\widetilde{\varepsilon}_{2k+1}^{(n)} & = \widetilde{\varepsilon}_{2k-1}^{(n+1)} + \dfrac{\mathbf{y}}{< \mathbf{y}, \widetilde{\varepsilon}_{2k}^{(n+1)} - \widetilde{\varepsilon}_{2k}^{(n)} >} \in E^*, & k, n = 0, 1, \ldots, \\
\widetilde{\varepsilon}_{2k+2}^{(n)} & = \widetilde{\varepsilon}_{2k}^{(n+1)} + \dfrac{\widetilde{\varepsilon}_{2k}^{(n+2)} - \widetilde{\varepsilon}_{2k}^{(n+1)}}{< \widetilde{\varepsilon}_{2k+1}^{(n+1)} - \widetilde{\varepsilon}_{2k+1}^{(n)}, \widetilde{\varepsilon}_{2k}^{(n+2)} - \widetilde{\varepsilon}_{2k}^{(n+1)} >} \in E, & k, n = 0, 1, \ldots
\end{array}\right.$$

Table 6: The TEA1 and TEA2 algorithms

Despite the more complicated rules, it is again possible, by using some algorithmic tricks, to proceed by diagonals. In each call, we need the storage of one and a half ascending diagonal of the ε-array (TEA1), and only one ascending diagonal for TEA2.

Odd rule (TEA1, TEA2)

$$\begin{tikzcd}
 & \varepsilon_{2k}^{(n)} \arrow[dr] & \\
\varepsilon_{2k-1}^{(n+1)} \arrow[ur] \arrow[dr] & & \boxed{\varepsilon_{2k+1}^{(n)}} \\
 & \varepsilon_{2k}^{(n+1)} \arrow[ur] &
\end{tikzcd}$$

Even rule (TEA1)

$$\begin{tikzcd}
\widehat{\varepsilon}_{2k}^{(n)} \arrow[dr] & & \\
 & \widehat{\varepsilon}_{2k+1}^{(n)} \arrow[dr] & \\
\widehat{\varepsilon}_{2k}^{(n+1)} \arrow[ur] \arrow[dr] & & \boxed{\widehat{\varepsilon}_{2k+2}^{(n)}} \\
 & \widehat{\varepsilon}_{2k+1}^{(n+1)} \arrow[ur] &
\end{tikzcd}$$

Even rule (TEA2)

$$\begin{tikzcd}
 & \widetilde{\varepsilon}_{2k+1}^{(n)} \arrow[dr] & \\
\widetilde{\varepsilon}_{2k}^{(n+1)} \arrow[ur] \arrow[dr] & & \boxed{\widetilde{\varepsilon}_{2k+2}^{(n)}} \\
 & \widetilde{\varepsilon}_{2k+1}^{(n+1)} \arrow[ur] & \\
\widetilde{\varepsilon}_{2k}^{(n+2)} \arrow[ur] & &
\end{tikzcd}$$

Table 7: The relations for TEA1 and TEA2 algorithms (in the odd rule $\boldsymbol{\varepsilon}$ is $\widehat{\boldsymbol{\varepsilon}}$ for TEA1 and $\widetilde{\boldsymbol{\varepsilon}}$ for TEA2).

The main drawback, avoided by the simplified versions of these algorithms, is that the duality product appears in the denominators of the rules and, moreover, it has to be taken in the even rules with linear functionals computed recursively by the odd rules. Thus, the original algorithms can be easily used only when $E = \mathbb{R}^m$ since it is its own dual space and the duality product is the inner product.

As in the case of the vector ε-algorithm, we included in the directory `TEAEfun` the two functions implementing these algorithms, for possible comparisons, and named `TEA1` and `TEA2`. The use of these functions is similar to that described before, and the meaning of the parameter is almost the same.

```
[EPSINI,EPSVEC] = TEA1(EPSINI,EPSVEC,MAXCOL,Y,TOL,IFLAG)
[EPSINI,EPSVEC] = TEA2(EPSINI,EPSVEC,MAXCOL,Y,TOL,IFLAG)
```

Of course, since particular rules do not exist, the arguments `NSING` and `NDIGIT` are no longer present. But a new input argument, `Y` which represents $\mathbf{y}$ appearing in Table 6, is needed (and it has to be maintained the same in all the consecutive calls):

`Y`: Input real arbitrary vector used in the rules for computing the inner products.

As usual, the successive values stored in the output variable `EPSINI` follow the staircase path as showed in Table 2.

A script showing the use of these algorithms was inserted in the `demo` directory, and named `testTEA12.m`. Do not forget to insert the directory `TEAEfun` into the path before running it.

5 The simplified topological ε-algorithms STEA1 and STEA2

With this new algorithms it is possible to avoid the manipulation of elements of E^* since the linear functional and the dual product are used only in their initializations. Moreover there is only one rule involving only even lower index terms, and the storage (only of elements of E) is reduced. In fact, in each call, we need the storage of two half ascending diagonals of the ε-array (STEA1), and only half of an ascending diagonal for STEA2. This is due to the fact that we use in cascade the scalar ε-algorithm of Wynn with these new equivalent forms of the topological ε-algorithms. For each of them, there are four equivalent formulæ as showed in Table 8.

The structure of the ε-array is as in Table 1, but only the even lower index terms are computed. Of course, since we have to use also the scalar ε–algorithm for obtaining the scalar terms appearing in the formulæ, and because it also computes the odd terms, these algorithms again need $2k+1$ initial terms $\mathbf{S}_n$ for obtaining $\widehat{\boldsymbol{\varepsilon}}_{2k}^{(0)}$ (or $\widetilde{\boldsymbol{\varepsilon}}_{2k}^{(0)}$).

The relations between the terms of the array are now very simple, as showed in Table 9, and it is easy to proceed by ascending diagonals.

5.1 The Matlab `STEAn_v` functions

In the toolbox we implemented the four formulæ for both STEA1 and STEA2. The function have been inserted in the directory `STEAfun` and they have been named `STEA1_1`, `STEA1_2`, `STEA1_3`, `STEA1_4`, `STEA2_1`, `STEA2_2`, `STEA2_3`, `STEA2_4`. The calling structure, although the implementation is different, has been on purpose made identical for helping the user to pass from a formula to another one. Here we remind the complete description. Remark that `STEAn_v` denotes Formula `v` of the STEAn.

```
[EPSINI,EPSVEC] = STEAn_v (EPSINI,EPSVEC,EPSSCA,MAXCOL,TOL,IFLAG);
```

`EPSINI`: input/output argument. In input, `EPSINIS` must contain the new term of the original sequence of elements $\mathbf{S}_n \in E$ (vector or matrix). In the successive calls, the values in output follow again a descending staircase scheme in the simplified topological ε-array (up to `MAXCOL`), that is, for STEA1, the elements $\widehat{\boldsymbol{\varepsilon}}_0^{(0)}, \widehat{\boldsymbol{\varepsilon}}_0^{(1)}, \widehat{\boldsymbol{\varepsilon}}_2^{(0)}, \widehat{\boldsymbol{\varepsilon}}_2^{(1)}, \ldots, \widehat{\boldsymbol{\varepsilon}}_{\mathtt{MAXCOL}}^{(0)}, \widehat{\boldsymbol{\varepsilon}}_{\mathtt{MAXCOL}}^{(1)}, \widehat{\boldsymbol{\varepsilon}}_{\mathtt{MAXCOL}}^{(2)}, \widehat{\boldsymbol{\varepsilon}}_{\mathtt{MAXCOL}}^{(3)}, \ldots$, or the tilde ones for STEA2.

`EPSVEC`: input/output cell array argument. It contains after the k-th call the last computed backward diagonal of the topological epsilon scheme (only even column terms):
- if $k <= \mathtt{MAXCOL}+1$, k odd
$\boldsymbol{\varepsilon}_0^{(k-1)}, \boldsymbol{\varepsilon}_2^{(k-3)}, \boldsymbol{\varepsilon}_4^{(k-5)}, \ldots, \boldsymbol{\varepsilon}_{k-1}^{(0)}$
- if $k <= \mathtt{MAXCOL}+1$, k even
$\boldsymbol{\varepsilon}_0^{(k-1)}, \boldsymbol{\varepsilon}_2^{(k-3)}, \boldsymbol{\varepsilon}_4^{(k-5)}, \ldots, \boldsymbol{\varepsilon}_{k-2}^{(1)}$
- if $k > \mathtt{MAXCOL}+1$
$\boldsymbol{\varepsilon}_0^{(k-1)}, \boldsymbol{\varepsilon}_2^{(k-3)}, \boldsymbol{\varepsilon}_4^{(k-5)}, \ldots, \boldsymbol{\varepsilon}_{\mathtt{MAXCOL}}^{(k-1-\mathtt{MAXCOL})}$

$\boldsymbol{\varepsilon}$ is $\widehat{\boldsymbol{\varepsilon}}$ for STEA1 and $\widetilde{\boldsymbol{\varepsilon}}$ for STEA2. Before the first call, its input value must be an empty array.

`EPSSCA`: input scalar vector argument. In input, it must contain the last ascending diagonal computed and returned by the function `SEAW` since the rules of the STEA algorithms need values computed by the scalar ε–algorithm. Before the k-th call of the function, it must contain
- if $k <= \mathtt{MAXCOL}+1$
$\varepsilon_0^{(k-1)}, \varepsilon_1^{(k-2)}, \ldots, \varepsilon_{k-1}^{(0)}$
- if $k > \mathtt{MAXCOL}+1$
$\varepsilon_0^{(k-1)}, \varepsilon_1^{(k-2)}, \ldots, \varepsilon_{\mathtt{MAXCOL}}^{(k-1-\mathtt{MAXCOL})}$,

where the ε's are the scalars obtained by the scalar ε–algorithm.

`MAXCOL`: input argument. It represents the index of the last column of the epsilon scheme that the user wants to compute. This value must be a positive even integer number, otherwise an error occurs. This argument must have the *same value* as the value used in the function `SEAW`.

`TOL`: input argument. Input real value used in tests for near-breakdowns. If `abs(X) < TOL`, then `X` is considered to be zero, and an error occurs. Usually the same value used in the call of `SEAW`.

`IFLAG`: input argument. Input integer to be set to zero at the 'first' call of the function for the initializations of the scheme. If this value is different from zero, or the argument is missing, the function continues in expanding the previous scheme. For a new application of the algorithm, the user must remember to call the function with `IFLAG` equal to zero.

STEA1 algorithm

$$
\begin{aligned}
\text{Formula 1}\quad \widehat{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + \frac{1}{(\varepsilon_{2k}^{(n+1)} - \varepsilon_{2k}^{(n)})(\varepsilon_{2k+1}^{(n+1)} - \varepsilon_{2k+1}^{(n)})}(\widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} - \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n)}),\\
\text{Formula 2}\quad \widehat{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + \frac{\varepsilon_{2k+1}^{(n)} - \varepsilon_{2k-1}^{(n+1)}}{\varepsilon_{2k+1}^{(n+1)} - \varepsilon_{2k+1}^{(n)}}(\widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} - \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n)}),\\
\text{Formula 3}\quad \widehat{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + \frac{\varepsilon_{2k+2}^{(n)} - \varepsilon_{2k}^{(n+1)}}{\varepsilon_{2k}^{(n+1)} - \varepsilon_{2k}^{(n)}}(\widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} - \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n)}),\\
\text{Formula 4}\quad \widehat{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + (\varepsilon_{2k+1}^{(n)} - \varepsilon_{2k-1}^{(n+1)})(\varepsilon_{2k+2}^{(n)} - \varepsilon_{2k}^{(n+1)})(\widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} - \widehat{\boldsymbol{\varepsilon}}_{2k}^{(n)}),
\end{aligned}
$$

with $\widehat{\boldsymbol{\varepsilon}}_0^{(n)} = \mathbf{S}_n \in E$, $n = 0, 1, \ldots$.

STEA2 algorithm

$$
\begin{aligned}
\text{Formula 1}\quad \widetilde{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + \frac{1}{(\varepsilon_{2k}^{(n+2)} - \varepsilon_{2k}^{(n+1)})(\varepsilon_{2k+1}^{(n+1)} - \varepsilon_{2k+1}^{(n)})}(\widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+2)} - \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)}),\\
\text{Formula 2}\quad \widetilde{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + \frac{\varepsilon_{2k+1}^{(n+1)} - \varepsilon_{2k-1}^{(n+2)}}{\varepsilon_{2k+1}^{(n+1)} - \varepsilon_{2k+1}^{(n)}}(\widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+2)} - \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)}),\\
\text{Formula 3}\quad \widetilde{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + \frac{\varepsilon_{2k+2}^{(n)} - \varepsilon_{2k}^{(n+1)}}{\varepsilon_{2k}^{(n+2)} - \varepsilon_{2k}^{(n+1)}}(\widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+2)} - \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)}),\\
\text{Formula 4}\quad \widetilde{\boldsymbol{\varepsilon}}_{2k+2}^{(n)} &= \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} + (\varepsilon_{2k+1}^{(n+1)} - \varepsilon_{2k-1}^{(n+2)})(\varepsilon_{2k+2}^{(n)} - \varepsilon_{2k}^{(n+1)})(\widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+2)} - \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)}),
\end{aligned}
$$

with $\widetilde{\boldsymbol{\varepsilon}}_0^{(n)} = \mathbf{S}_n \in E$, $n = 0, 1, \ldots$.

Table 8: The STEA1 and STEA2 algorithms. The scalar quantities ε's are obtained by the scalar ε–algorithm applied to the sequence $(\langle \mathbf{y}, \mathbf{S}_n \rangle)$.

$$
\begin{tikzcd}
\text{STEA1} & & & \text{STEA2} & & \\
\widehat{\boldsymbol{\varepsilon}}_{2k}^{(n)} \arrow[rrd] & & & & & \\
\widehat{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} \arrow[rr] & & \boxed{\widehat{\boldsymbol{\varepsilon}}_{2k+2}^{(n)}} & \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+1)} \arrow[rr] & & \boxed{\widetilde{\boldsymbol{\varepsilon}}_{2k+2}^{(n)}} \\
 & & & \widetilde{\boldsymbol{\varepsilon}}_{2k}^{(n+2)} \arrow[rru] & &
\end{tikzcd}
$$

Table 9: The relations for the first (STEA1) and the second (STEA2) simplified topological ε–algorithms.

5.2 A simple pseudocode for the Acceleration Method with STEA-algorithms

Acceleration Method

```
Initializations
    Set MAXCOL, TOL, NDIGIT, NBC
    Set $\mathbf{y}$
Computations
    IFLAG $\leftarrow 0$
    EPSINI $\leftarrow \mathbf{S}_0$
    EPSINIS $\leftarrow S_0 = \langle \mathbf{y}, \mathbf{S}_0 \rangle$
    First call of SEAW
    First call of STEAn_v
    Output EPSINI
    for $n = 1 :$ NBC $- 1$
        EPSINI $\leftarrow \mathbf{S}_n$
        EPSINIS $\leftarrow S_n = \langle \mathbf{y}, \mathbf{S}_n \rangle$
        Call SEAW
        Call STEAn_v
        Output EPSINI
    end for $n$
```

where `STEAn_v` implements Formula `v` of the STEAn.

5.3 The storage requirements

The functions also contain some local working cell arrays and internal persistent arrays of scalars and cell arrays. See each function for more details. If we do not consider the scalar arrays (that are not so expensive in storage requirements), and we only take into account the storage needed for the cell arrays (that may contains vectors of matrices that could be large), we have the results of Table 10. In this table we suppose to have chosen a certain `MAXCOL`, and we don't count the storage of the original sequence in the script.

algorithm	# elements ε-array	in space	# auxiliary elements
STEA1	2 × `MAXCOL`	E	2
STEA2	`MAXCOL`	E	2

Table 10: Storage requirements of STEA algorithms.

5.4 About the choice of the STEA and of the formula

We remark that in all the examples in Section 5 of the paper, we always use the functions `STEA1_3` and `STEA2_3`. The reason is that, the third formulæ use only even terms of the scalar ε-algorithm of Wynn (the interesting ones) and this fact seems to make this choice reasonable. But numerical tests showed that, in most cases, there is not a great difference in comparison to the other formulæ.

Concerning the choice between STEA1 and STEA2, our feeling after several numerical experiments is that STEA2, in almost all cases, exhibits a better performance. This is perhaps due to the fact that, for constructing the extrapolated term, the first topological Shanks transformation is a combination of the terms $\mathbf{S}_n, \ldots, \mathbf{S}_{n+k}$ of the original sequence, and the second topological Shanks transformation uses the terms $\mathbf{S}_{n+k}, \ldots, \mathbf{S}_{n+2k}$. Moreover, as you may see in the preceding section (Table 10), the storage requirement of the STEA2 algorithm is cheeper. In any case, no theoretical results exist for indicating in a firm way which algorithm is the best in a particular example.

Since the use of all functions is the same, the user can easily change only the name of the called function in any of the scripts proposed and inserted in the toolbox. No other modifications are needed.

5.5 About the choice of the arguments

The comments inserted in this section concern the functions `STEAv_n` and `SEAW`, but it could also be applied, when appropriate, to the other functions (`VEAW`, `TEA1` and `TEA1`).

`MAXCOL`: Sometimes for theoretical reasons, we know that in exact arithmetic, the result is obtained in a certain `MAXCOL` column. This is the case in the Acceleration Method, for instance, of sequences belonging to the kernel of the transformation of sequences generated by $\boldsymbol{x}_{n+1} = A\boldsymbol{x}_n + \boldsymbol{b}$ where `MAXCOL` is equal to the double of the dimension of the system. This last condition is the same when we want to use the Generalized Steffensen Method (see Section 5.7). But, in general, there is no reason for choosing a value or another for this parameter. The experience of the test performed suggests to start by choosing a small value (2 or 4) and, after, to try to increase it. But in all the examples of the paper, for the Acceleration Method, we do not exceed the even column 12. This suggestion is also due to the fact that all the extrapolation algorithms suffer from numerical instability and the better they work, the more instabilities appear in the successive event columns. The use of the particular rules is often not sufficient for avoiding all the instability problems. In the figures of the examples given in the paper, it is easy to see that, when we are close to the solution, the curve exhibits an oscillating behaviour. Remind to give the *same value* to the functions `SEAW` and `STEAv_n`.

`TOL`: In the denominator of all the formulæ (`STEA1_4` and `STEA2_4` apart) there are differences between quantities that could become almost equal in the construction of the ε-array, thus producing a near-breakdown (sometimes they become the same floating point number and, thus, a breakdown occurs). In the function, every scalar denominator is tested with `TOL` before to proceed in the computation. Please notice that in `SEAW` this is not the parameter which has to be changed for activating the particular rules of Wynn. By setting this argument, the user may decide when he considers the near-breakdown to be too serious and, then, to stop with an error.

Since, in a certain number of examples of the scalar ε-algorithm, it is pretty usual that the intermediate odd terms are large but do not prevent to proceed without problems with the algorithm, we usually start by setting this value equal to a small value (for instance `TOL=1e-20`), and, if an error occurs, the function shows the value in the denominator, thus allowing the user to change it to a smaller value.

If the user want to avoid completely the check of the denominators, it is sufficient to set `TOL=0`. As you know, Matlab continues to run also with a division by a zero value. So, the user, by setting this parameter equal to zero, has to be prepared to eventually obtain results like `Inf` or `NaN`!

`NDIGIT` (only `SEAW`): In Section 2.2.1, it is already explained that this argument is needed for detecting an isolated singularity and, thus, for applying the particular rules. Let us remind that the particular rules allow to compute in a mathematically equivalent way a term of the ε-array that, otherwise, could be badly computed. But, in fact we can use these rules also when there is no singularity! Thus, in theory, any positive integer denoting the number of common digits that activates the particular rules, can be used. But, if this value is too small, it is very usual that the function detects non-isolated singularities and, thus, an error occurs.

Thus, in practice, the user has to be very careful in setting this value. The suggestion is to start by setting `NDIGIT=15` or `16` (thus near the machine precision) and to try to decrease it a little bit for seeing if the rules are activated (see the output parameter `NSING` and the `.log` file), possibly without obtaining non-isolated singularities. We hope to built soon a version of the scalar ε-algorithm treating non-isolated singularities, and the updating our toolbox.

5.6 About the functional y and the dual product

There are several possibilities for choosing the linear functional $\mathbf{y} \in E^*$, since now, with the simplified algorithms, it appears only in the initialization terms of the scalar ε-algorithm of Wynn.

- When $E = \mathbb{R}^m$, we can, of course, choose the usual inner product

$$\mathbf{y} : \mathbf{S}_n \in \mathbb{R}^m \longmapsto < \mathbf{y}, \mathbf{S}_n >= (\mathbf{y}, \mathbf{S}_n)$$

  or also choose a matrix $M \in \mathbb{R}^{s\times m}$, $\mathbf{y} \in \mathbb{R}^s$, and consider

$$\mathbf{y} : \mathbf{S}_n \in \mathbb{R}^m \longmapsto < \mathbf{y}, \mathbf{S}_n >= (\mathbf{y}, M\mathbf{S}_n).$$

- When $E = \mathbb{R}^{m\times m}$, we may simply take

$$\mathbf{y} : \mathbf{S}_n \in \mathbb{R}^{m\times m} \longmapsto < \mathbf{y}, \mathbf{S}_n >= \mathrm{trace}(\mathbf{S}_n).$$

- When $E = \mathbb{R}^{m\times s}$, we may choose a matrix $Y \in \mathbb{R}^{m\times s}$ and define

$$\mathbf{y} : \mathbf{S}_n \in \mathbb{R}^{m\times s} \longmapsto < \mathbf{y}, \mathbf{S}_n >= \mathrm{trace}(Y^T\mathbf{S}_n).$$

  If we set $Y = \mathbf{1}_{m\times s}$, the previous choice corresponds to consider the sum of all the elements of $\mathbf{S}_n$.

  We may also take $\mathbf{u} \in \mathbb{R}^m$ and $\mathbf{v} \in \mathbb{R}^s$ and set

$$\mathbf{y} : \mathbf{S}_n \in \mathbb{R}^{m\times s} \longmapsto < \mathbf{y}, \mathbf{S}_n >= (\mathbf{u}, \mathbf{S}_n\mathbf{v}).$$

  Of course these choices can be made also when $s = m$.

  We often remarked that the scalars $S_n = \langle \mathbf{y}, \mathbf{S}_n \rangle$ used in the scalar ε-algorithm become, in absolute value, very large. Thus, in the template scripts and in the demos producing the examples of the paper, we inserted a new variable, called `MAXD`, set, for instance, equal to `1e+20`, for testing the terms of the scalar sequence and putting a warning for the user when the term is bigger than `MAXD`.

5.7 The Restarted Method (RM)

When we have a fixed point problem, that is when we have sequences that are iteratively constructed from an arbitrary initial guess and whose limit is independent from it, instead of using the Acceleration Method we can also use the Restarted Method. It consists in constructing a complete ε-array by using $2k + 1$ terms of the original sequence (a cycle), thus obtaining $\boldsymbol{\varepsilon}_{2k}^{(0)}$. After that we restart the procedure by using the extrapolated term $\boldsymbol{\varepsilon}_{2k}^{(0)}$ as an initial guess. And so on, for a user defined number of cycles. If the system of equations we are solving has dimension m and if we take $k = m$, the Restarted Method is called the Generalized Steffensen Method (GSM).

5.7.1 A simple pseudocode for the Restarted Method with STEA-algorithms

Restarted Method

```
Initializations
    Set MAXCOL, TOL, NDIGIT, NCY
    Set $\mathbf{y}$
    NBC $\leftarrow$ MAXCOL $+ 1$
Computations
    EPSINI $\leftarrow \mathbf{S}_0$
    EPSINIS $\leftarrow S_0 = \langle \mathbf{y}, \mathbf{S}_0 \rangle$
    for $i = 1$ : NCY
        IFLAG $\leftarrow 0$
        First call of SEAW
        First call of STEAn_v
        for $n = 1$ : NBC $- 1$
            EPSINI $\leftarrow \mathbf{S}_n = F(\mathbf{S}_{n-1})$
            EPSINIS $\leftarrow S_n = \langle \mathbf{y}, \mathbf{S}_n \rangle$
            Call SEAW
            Call STEAn_v
        end for $n$
        $\mathbf{S}_0 \leftarrow$ EPSINI
        EPSINIS $\leftarrow S_0 = \langle \mathbf{y}, \mathbf{S}_0 \rangle$
    end for $i$
```

In the pseudocode, the new variable `NCY` denotes the number of cycles to be performed. The user has not to forget to reinitialize the construction of the ε-arrays by setting `IFLAG=0` in both `SEAW` and `STEAv_n` functions, and, at the beginning, to choose always `NBC=MAXCOL+1`. All the other comments are similar to those already made for the Accelerating Method.

6 The template scripts

In the directory `templates`, we inserted four examples of scripts for solving the simple example of Section 4.4 of the paper. Two of them (`AMmain_MR` and `AMmain_EA`) uses the Accelerated Method applied to two different methods for producing the original sequence. The two others, (`RMmain_MR` and `RMmain_EA`), use the same basic methods, but apply the Restarted Method. It is probably unuseful to describe in details all of them. The main difference is that the scripts implementing the EA method for producing a new element of the basic sequence use an external function, called `Fiter_EA`. This structure is quite general and it could be applied to the user's problems. Thus, we will describe only these scripts in more detail.

6.1 The template `AMmain_EA`

This script also contains some statements used for the graphical representation that we do not insert in this description. There are also a lot of test statements for the correct definition of some variables and the tests for the magnitude of the dual product (see Section 5.6 of this user guide).

The script `AMmain_EA.m`

We start by calling the function `clearSEAW` of the toolbox that closes the external `.log` file (eventually opened in the current folder by the `SEAW` function during a previous experiment) and cancels it. This function also clears the function `SEAW`, and, thus, also clears their persistent variables, in particular the variable for defining the name of the external `.log` file. This is mandatory for not creating too many unuseful different files. We also clear the variables of the workspace.

```
clearSEAW
clear variables
```

Next we initialize the variables needed for the Acceleration Method.

```
% Initializations for SEAW and STEA
% Define the algorithm STEA we want to use (1 or 2)
STEAn = 2;
% Define the formula we want to use (1,2,3 or 4)
STEAv = 3;
% Choice of the maximum even column in the epsilon-array
MAXCOL = 6;
% Choice of the number of terms of the original sequence
NBC = 51;
% Assign the tolerance for denominators
TOL = 1e-20;
% Number of common digit asked for detecting a singularity
NDIGIT = 15;
```

In this template, it is easy to change the version of the STEA and the formula wanted by the user. A function handle variable (named `STEAf`) is defined by using the choices given in the initialization part.

```
% Define the function handle
STEAs = ['STEA',num2str(STEAn),'_',num2str(STEAv)];
STEAf = str2func(STEAs);
```

Now we define the problem we want to solve and the basic method to be used (see Section 4.4 of the article).

```
% Initializations for the example and method chosen
% A is a 3x3 matrix
A = [0.37,0.13,0.12; -0.30,0.34,0.12; 0.11,-0.17,0.29];
% m is the dimension
m = size(A,1);
% Define a handle function for computing the Frobenious norm of the
% residual
nres = @(Y) (norm(Y + A'*(Y\A) - eye(size(A)),'fro'));
% X is the starting matrix X_0
X = eye(m);
% Y is the starting value for the auxiliary matrix of the iterative method
Y = X;
```

We start the computations with the first calls of `SEAW` and `STEAf` (with `IFLAG=0`) for initializing the construction of the scalar and topological ε-arrays.

```
% First call of the SEAW and STEA
% X_0 is the starting matrix X
% Compute the Frobenious norms of the residual matrix
% Original method
nX(1) = nres(X);
% Frobenious norm of the first extrapolated term
```

```
nS(1) = nres(X);
% Compute <y,X_0>= trace(X_0)
EPSINIS = trace(X);
% First call of SEAW
[EPSINIS,EPSSCA,NSING] = SEAW(EPSINIS,[],MAXCOL,TOL,NDIGIT,0);
% First call of STEA
[EPSINI,EPSVEC] = STEAf (X,[],EPSSCA,MAXCOL,TOL,0);
% Frobenious norm of the first extrapolated term
nS(1) = nres(EPSINI);
```

After that, the loop will start (`NBC-1` terms of the original sequence and their dual products are added and elaborated in the the scalar and topological ε-arrays). Please note that each new term of the original sequence is obtained by calling the external function `Fiter_EA`.

```
% Start of the loop
for n = 1:NBC-1
    % In output, X is the new element X_n
    [X,Y] = Fiter_EA(X,A,Y);
    % Compute the norm of the residual (basic method)
    nX(n+1) = nres(X);
    % Compute <y,X_n>= trace(X_n)
    EPSINIS = trace(X);
    % next call of SEAW and STEA
    [EPSINIS,EPSSCA,NSING] = SEAW(EPSINIS,EPSSCA,MAXCOL,TOL,NDIGIT);
    [EPSINI,EPSVEC] = STEAf (X,EPSVEC,EPSSCA,MAXCOL,TOL);
    % Compute the norm of the residual (new extrapolated term)
    nS(n+1) = nres(EPSINI);
end
% End of the loop
```

That's all! In the script there are of course also the commands for producing a figure and for displaying a summary of the data and of the results in the Command window. With the data given, we obtain in output

```
>> AMmain_EA
Iteration 1 completed
Iteration 2 completed
Iteration 3 completed
....................
....................
Iteration 50 completed
Iteration 51 completed

An example - Acceleration Method with STEA2_3
Basic method of S.M. El-Sayed and A.M. Al-Dbiban

 Dimension m                      = 3
 Number of terms of the sequence  = 51
 Max even column                  = 6
 Tolerance for the algorithms     = 1.0e-20
 Number of the common digits      = 15
 Maximum for the |duality product| = 1.0e+20
 Maximum number of possible isolated singularities = 3
 Total number of isolated singularities found in the eps-array = 0

 At iteration 51 we obtain
 ||f(X)||_F         = 4.94e-05
 ||f(eps)||_F       = 6.89e-11
>>
```

The external function `Fiter_EA` is

```
function [X,Y] = Fiter_EA(X,A,Y)
% Fiter_EA   Computes the next iterates of the sequences from the
%            previous ones by using the Method of El-Sayed and Al-Dbiban
%            (2005) for solving a nonlinear matrix equation
%
% [X,Y] = Fiter_EA(X,A,Y)
%
% The relations are
% Y_{k+1} = (I - X_k) Y_k + I
% X_{k+1} = I - A^T Y_{k+1} A

m = size(A,1);
Y = (eye(m)-X)*Y+eye(m);
X = eye(m)-A'*Y*A;
```

6.2 The template `RMmain_EA`

This script is similar to that of the previous Section and the considerations are similar. We will only point out the main differences.

The script `RMmain_EA.m`

We start by calling the function `clearSEAW` of the toolbox, and we clear the variables of the workspace.

```
clearSEAW
clear variables
```

Next, we initialize the variables needed for the Restarted Method. Remark that the variable `NCY` (number of asked cycles) has to be set (here we set it to 7), and that is mandatory to put `NBC=MAXCOL+1`. Since the system of the example has dimension $m = 3$, by setting `MAXCOL=6` we run, in fact, the Generalized Steffensen Method.

```
% Initializations for SEAW and STEA
% Define the algorithm STEA we want to use (1 or 2)
STEAn = 2;
% Define the formula we want to use (1,2,3 or 4)
STEAv = 3;
% Choice of the maximum even column in the epsilon-array
MAXCOL = 6;
% Choice of number of cycles
NCY = 7;
% Set the number of terms of the original sequence
NBC = MAXCOL+1;
% Assign the tolerance for denominators
TOL = 1e-20;
% Number of common digit asked for detecting a singularity
NDIGIT = 15;
```

We define the function handle variable `STEAf`.

```
% Define the function handle
STEAs = ['STEA',num2str(STEAn),'_',num2str(STEAv)];
STEAf = str2func(STEAs);
```

Now we define the problem we want to solve and the basic method to be used. This part coincides with that of the Acceleration Method.

```
% Initializations for the example and method chosen
% A is a 3x3 matrix
A = [0.37,0.13,0.12; -0.30,0.34,0.12; 0.11,-0.17,0.29];
% m is the dimension
m = size(A,1);
% Define a handle function for computing the Frobenious norm of the
% residual
nres = @(Y) (norm(Y + A'*(Y\A) - eye(size(A)),'fro'));
% X is the starting matrix X_0
X = eye(m);
% Y is the starting value for the auxiliary matrix of the iterative method
Y = X;
```

We start the outer loop (for the cycles), and we perform the first calls of `SEAW` and `STEAf` (with `IFLAG=0`) for initializing the construction of the scalar and topological ε-arrays.

```
% Start the outer loop
for i = 1:NCY
    % Compute the Frobenious norms of the residual matrix u_0
    nXit(MAXCOL*(i-1)+i) = nres(X);
    % Frobenious norm of the first extrapolated term = initial matrix
    nS(i) = nres(X);
    % Compute <y,u_0>= trace(u_0)
    EPSINIS = trace(X);
    % First call of SEAW
    [EPSINIS,EPSSCA,NSING]=SEAW(EPSINIS,[],MAXCOL,TOL,NDIGIT,0);
    % First call of STEA
    [EPSINI,EPSVEC] = STEAf(X,[],EPSSCA,MAXCOL,TOL,0);
```

After that, the inner loop will start.

```
%   Start of the inner loop
    for n = 1:NBC-1
        % In output, X is the new element of the sequence u_n
        [X,Y] = Fiter_EA(X,A,Y);
        % Compute the norm of the residual of u_{n}
        nXit(NBC*(i-1)+n+1) = nres(X);
        % Compute <y,u_n> = trace(u_n)
        EPSINIS = trace(X);
        % Next calls of SEAW and STEA
        [EPSINIS,EPSSCA,NSING]=SEAW(EPSINIS,EPSSCA,MAXCOL,TOL,NDIGIT);
        [EPSINI,EPSVEC] = STEAf (X,EPSVEC,EPSSCA,MAXCOL,TOL);
    end
    % End of the inner loop
```

Before ending the outer loop, we set the restarting values.

```
    % Set the restarting u_0=eps_{2k}^(0) element for the next outer loop
    X = EPSINI;
    % Define the corresponding auxiliary matrix for the next outer loop
    Y = inv(EPSINI);
end
% End of the outer loop
```

After the outer loop, we set the norm of the last extrapolated term.

```
% Define the last extrapolated component for the graphical representation
nS(NCY+1) = nres(EPSINI);
```

In this script, there are also commands for producing a figure and for displaying a summary of the data and of the results in the Command window. With the data given, we obtain in output

```
>> RMmain_EA
Inner iteration 1 of cycle 1 completed
Inner iteration 2 of cycle 1 completed
Inner iteration 3 of cycle 1 completed
Inner iteration 4 of cycle 1 completed
Inner iteration 5 of cycle 1 completed
Inner iteration 6 of cycle 1 completed
Inner iteration 7 of cycle 1 completed
* Outer iteration 1 completed
* Total number of singularities found in the eps-array = 0

Inner iteration 1 of cycle 2 completed
Inner iteration 2 of cycle 2 completed
Inner iteration 3 of cycle 2 completed
......................................
......................................

......................................
......................................
Inner iteration 6 of cycle 6 completed
Inner iteration 7 of cycle 6 completed
* Outer iteration 6 completed
* Total number of singularities found in the eps-array = 0

Inner iteration 1 of cycle 7 completed
Inner iteration 2 of cycle 7 completed
Inner iteration 3 of cycle 7 completed
Inner iteration 4 of cycle 7 completed
Inner iteration 5 of cycle 7 completed
Inner iteration 6 of cycle 7 completed
Inner iteration 7 of cycle 7 completed
* Outer iteration 7 completed
* Total number of singularities found in the eps-array = 0

An example - Restarted Method with STEA2_3
Basic method of S.M. El-Sayed and A.M. Al-Dbiban

 Dimension m                        = 3
 Number of terms for each cycle     = 7
 Max even column                    = 6
 Tolerance for the algorithms       = 1.0e-20
 Number of the common digits        = 15
 Number of cycles                   = 7
 Maximum for the |duality product| = 1.0e+20
 Maximum number of possible isolated singularities for each cycle = 3
 Total number of isolated singularities found in all the cycles   = 0

 At the last iteration we obtain
 ||f(X_{ori})||_F      = 6.27e-05
 ||f(eps)||_F          = 1.57e-09
>>
```

The external function `Fiter_EA` is the same as above. Remark that, with the restarted version, exactly `NCY .log` files (numbered in an incremental way) are created. This can be useful for looking where the isolated singularities have been detected, and, thus, in which cycle and in which column the particular rule has been applied. For showing the interest of the use of such rules, the user can change the values of some parameters by setting `NCY=12`. In this case, looking at the Command window, we see that 2 isolated singularities have been found and treated (one in cycle 10 and the other one in cycle 12). Looking at the external files `SEAW_<date>_10.log` and `SEAW_<date>_12.log`, the user can obtain information about the diagonals and the columns involved. If the user set `NDIGIT=16`, the particular rule in cycle 10 is not applied and a division by zero occurs in cycle 12. If, for avoiding the tests on the denominators, `TOL` is set to 0, the last extrapolated value is `||f(eps)||_F = NaN`.

Let us give some final very important remarks on the example of Section 4.4 of the article. The Accelerated Method works pretty well with both methods. But notice, for instance, that with `AMmain_MR` and by choosing STEA1, the accelerated values oscillate at the beginning, and are worse than those of the original method. For the Restarted Method, the results are very good but . . . only by using the STEA2! If the user tries to use STEA1, the results are completely wrong. This is not a surprise since, if we applied the extrapolation methods without a careful study of the theoretical method that produces the sequence, we may only hope to obtain good results! In the `RMmain_MR`, we simply restart with the extrapolated value as the new starting matrix, but it is not sure at all that we can apply the recursive formula of Monsalve and Raydan with any starting matrix instead of $X_0 = AA^T$.

7 The demo scripts

Two demo scripts have been inserted in the `demo` directory. They allow to reproduce all the Examples given in Section 5 of the paper. The script `AMdemo` concerns the Acceleration Method and both STEA1 and STEA2 (formula 3). We fix `TOL=1e-30` and `NDIGIT=15`. Some figures are also produced. The script `RMdemo` is similar, and it concerns the examples with the Restarted Method. Three additional functions are provided for running both scripts. The two functions (`exinit` and `exhelp`) are needed to initialize the example chosen, and to output in the command window the input values suggested and corresponding to those used in the paper. The third one (`Fiter`) is used by the demo for implementing the iterative method corresponding to the example chosen. Several tests are made inside of it for checking the user's choices.

Let us now report a possible use of both demos.

```
>> AMdemo
 Examples of the demo:
 1, 2, 3, 4a, 4b, 5, 6, 7, 8a, 8b, 9, 10, 11, 12

 Example number 1

 ***  AM - EXAMPLE 1 ***
 Nonlinear system, m = 5, solution x = (1,...,1)^T
 x_0 = (1/2,...,1/2)^T
 Take alpha = -0.05, MAXCOL = 4, NBC = 350

 Insert alpha -0.05
 Dimension of the system m = 5
 Insert max even column 4
 The number of iterations must be greater or equal to 5 to reach the column 4
 Insert the number of iterations 350
 Tolerance for scalar epsilon algorithm  = 1.0e-30
 Number of the common digits             = 15
 Maximum for the |duality product|       = 1.0e+20

*** STEA1
Iteration 1 completed
Iteration 2 completed
Iteration 3 completed
Iteration 4 completed
.....................
.....................
Iteration 349 completed
Iteration 350 completed
Total number of singularities found in the eps-array = 0

*** STEA2
Iteration 1 completed
Iteration 2 completed
Iteration 3 completed
Iteration 4 completed
.....................
.....................
Iteration 349 completed
Iteration 350 completed
Total number of singularities found in the eps-array = 0


 *** AM - Example 1 ***
 Dimension m          = 5
 alpha                = -5.00e-02
 Max even column      = 4
 Number of iterations = 350
 Total number of isolated singularities for STEA1 = 0
 Total number of isolated singularities for STEA2 = 0

 At iteration 350
 ||x-sol||         = 7.77e-03
 ||eps-sol|| stea1 = 1.45e-06
 ||eps-sol|| stea2 = 1.42e-06

>> RMdemo
 Examples of the demo:
 1, 2, 3, 4a, 4b, 5, 6, 7, 8a, 8b

 Example number 1

 ***  GSM - EXAMPLE 1 ***
 Nonlinear system, m = 5, solution x = (1,...,1)^T
 x_0 = (1/2,...,1/2)^T
 Take alpha = -0.05, MAXCOL = 10, NCY = 5
 Insert alpha -0.05
 Dimension of the system m = 5
 Insert max even column (= 2*m for GSM) 10
 Insert the number of outer loops 5
 Tolerance for scalar epsilon algorithm  = 1.0e-30
 Number of the common digits             = 15
 Maximum for the |duality product|       = 1.0e+20

*** STEA1
Inner iteration 1 completed
Inner iteration 2 completed
Inner iteration 3 completed
Inner iteration 4 completed
Inner iteration 5 completed
Inner iteration 6 completed
Inner iteration 7 completed
Inner iteration 8 completed
Inner iteration 9 completed
Inner iteration 10 completed
Inner iteration 11 completed
* Outer iteration 1 completed
* Total number of singularities found in the eps-array = 0

Inner iteration 1 completed
Inner iteration 2 completed
..........................
..........................
Inner iteration 10 completed
Inner iteration 11 completed
* Outer iteration 5 completed
* Total number of singularities found in the eps-array = 0

*** STEA2
Inner iteration 1 completed
Inner iteration 2 completed
Inner iteration 3 completed
Inner iteration 4 completed
..........................
..........................
Inner iteration 10 completed
Inner iteration 11 completed
* Outer iteration 5 completed
* Total number of singularities found in the eps-array = 0


 *** GSM - Example 1 ***
 Dimension m                  = 5
 alpha                        = -5.00e-02
 Max even column              = 10
 Number of terms for each
 inner iteration              = 11
 Number of cycles             = 5
 Total number of isolated singularities for STEA1 (all cycles) = 0
 Total number of isolated singularities for STEA2 (all cycles) = 0

 At last iteration of the Restarted Method
 ||x_{ori}-sol||   = 1.31e-01
 ||eps-sol|| stea1 = 6.66e-11
 ||eps-sol|| stea2 = 1.43e-08
```

Of course the user can try the same examples by changing the input values suggested in order to practice with the methods. But the results are not assured. Let us show an example that does not work. We take again the Example 1, but we change the value of α and the number of iterations (`alpha = 0.01` and `NBC = 60`). Errors occur during the execution and it stops.

```
>> AMdemo
 Examples of the demo:
 1, 2, 3, 4a, 4b, 5, 6, 7, 8a, 8b, 9, 10, 11, 12

 Example number 1

 ***  AM - EXAMPLE 1 ***
 Nonlinear system, m = 5, solution x = (1,...,1)^T
 x_0 = (1/2,...,1/2)^T
 Take alpha = -0.05, MAXCOL = 4, NBC = 350

 Insert alpha 0.01
 Dimension of the system m = 5
 Insert max even column 4
 The number of iterations must be greater or equal to 5 to reach the column 4
 Insert the number of iterations 60
 Tolerance for scalar epsilon algorithm  = 1.0e-30
 Number of the common digits             = 15
 Maximum for the |duality product|       = 1.0e+20

*** STEA1
Iteration 1 completed
Iteration 2 completed
....................
....................
Iteration 51 completed
Iteration 52 completed
Warning:  At iteration 1 the absolute value of the duality product is 4.180e+46 >
1.00000e+20
> In AMdemo (line 153)
Iteration 53 completed
Warning:  At iteration 1 the absolute value of the duality product is 2.470e+116
> 1.00000e+20
> In AMdemo (line 153)
Value of denominator 2.39220e-47 in column 2
Error using SEAW (line 382)
SEAW - Division by a value < TOL in the normal rule. Impossible to continue.

Error in AMdemo (line 159)
        [EPSINIS,EPSSCA,NSING] = SEAW(EPSINIS,EPSSCA,MAXCOL,TOL,NDIGIT);
```

For helping the user in understanding better what happens, a script called `plot_demo` is provided for showing some information and for producing figures in the Command window . If, after the preceding unsuccessful run, we use this script we obtain

```
>> plot_demo
Insert the method for which we want to obtain results and figures
(AM = 1, RM = 2) 1

 *** AM - Example 1 ***
 Dimension m          = 5
 alpha                = 1.00e-02
```

```
Max even column      = 4
Number of iterations = 60

*** STOPPED during STEA1
*** Number of iterations completed = 53

At last iteration of the Acceleration Method
||x-sol||   = 4.18e+46
||eps-sol|| stea1 = 2.17e+03
```

The figure we obtain shows that the original method diverges and that the STEA1 do not works. The `.log` files could also help.

As previously said, the values `TOL=1e-30` and `NDIGIT=15` are fixed and assigned directly in the scripts `AMdemo` and `RMdemo`. Of course the user can change them, but we suggest not to change any other statements in the scripts and in the related functions since they are pretty connected.