Computing and Informatics, Vol. 31, 2012, 713–726

MILLER ANALYZER FOR MATLAB: A MATLAB PACKAGE FOR AUTOMATIC ROUNDOFF ANALYSIS

Attila Gáti

Óbuda University
Bécsi út 96/b
1034 Budapest, Hungary
e-mail: `gati.attila@phd.uni-obuda.hu`

Communicated by János Fodor

Abstract. We give a first report on our new software, Miller Analyzer for Matlab, which is an automatic roundoff error analyzer that extends the work of Miller et al. Using the original work of Miller, the analyzed numerical algorithm had to be expressed in a special, greatly simplified programming language. The main disadvantage of Miller's software is its own programming language and its restrictions. We have eliminated this drawback by creating a Matlab interface to the method of Miller. Applying the operator overloading based implementation technique of automatic differentiation, we have provided a means of analyzing numerical methods given in the form of Matlab m-functions. This new way of defining the analyzed numerical method is more flexible, and the integration into the Matlab environment makes the use of the software easy. Our work can be useful in analyzing the numerical stability of algorithms that use only the four basic real (not complex) arithmetic operations and square root. At the end of the paper, we discuss two such examples.

Keywords: Automatic roundoff analysis, numerical stability, automatic differentiation, Matlab, operator overloading

Mathematics Subject Classification 2010: 65G50, 65G99, 68N19, 68N20, 68W99

1 INTRODUCTION

We present a new Matlab package for automatic roundoff error analysis, based on the method developed by Webb Miller, and originally implemented in Fortran language by Webb Miller and David Spooner in 1978. Our software provides all the functionalities of the work by Miller and extends its applicability to such numerical algorithms that were complicated or even impossible to analyze with Miller's method before. As the analyzed numerical algorithm can be given in the form of a Matlab m-file, our software is easy to use. We give a first report on our software, Miller Analyzer for Matlab, which is hopefully useful for numerical people to analyze the effects of rounding errors.

More information on the use of the software by Miller and its theoretical background can be found in [10, 11, 12, 14, 15]. The software is in the ACM TOMS library with serial number 532 [13]. Using Miller's method, one can analyze the stability of numerical methods executed in floating point arithmetic. The software was intended to help algorithm designers seeking numerically stable algorithms or users doing initial testing of a new algorithm proposed for use. The basic idea of Miller's method is as follows. Given a numerical algorithm to analyze, a number $\omega(d)$ is associated with each set d ($d \in \mathbb{R}^n$) of input data. The function $\omega : \mathbb{R}^n \to \mathbb{R}$ measures rounding error, i.e., $\omega(d)$ is large exactly when the algorithm applied to d produces results which are excessively sensitive to rounding errors. A numerical maximizer is applied to search for large values of ω to provide information about the numerical properties of the algorithm. Finding a large value of ω can be interpreted as the given numerical algorithm is suffering from a specific kind of instability.

The software performs backward error analysis. The value $\omega(d) \cdot u$ (where u is the machine rounding unit) can be interpreted as the first order approximation of the upper bound for the backward error. The computation of the error measuring number is based on the partial derivatives of the output with respect to the input and the individual rounding errors. An automatic differentiation algorithm is used to provide the necessary derivatives.

In the book by Miller and Wrathall [15], the potential of the software is clearly demonstrated through 14 case studies. The answers of the software are consistent in these cases with the well known formal analytical and experimental results. The program shows correctly the stability properties of algorithms such as the Gaussian elimination without pivoting and with partial pivoting, the Gauss-Jordan elimination, the Cholesky factorization, the classical and modified Gram-Schmidt methods, the application of normal equations and Householder reflections for linear least squares problem and so on. We used the software package to examine the stability of some ABS methods [1], namely the implicit LU and several variants of the Huang method [6]. The results agreed with the already known facts about the numerical stability of the algorithms. The program has shown that implicit LU is numerically unstable and that the modified Huang method has better stability properties than the original Huang method.

The numerical method to be analyzed must be expressed in a special, greatly simplified Fortran-like language. We can construct for-loops and if-tests that are based on the values of integer expressions. There is no way of conversion between real and integer types, and no mixed expressions (that contain both integer and real values) are allowed. Hence we can define only straight-line programs, i.e., where the flow of control does not depend on the values of floating point variables. To analyze methods with iterative loops and with branches on floating point values, the possible paths through any comparisons must be treated separately. This can be realized by constrained optimization. We confine search for maximum to those input vectors by which the required path of control is realized. The constraints can be specified through a user-defined subroutine. Higham mentions the special language and its restrictions as the greatest disadvantage of the software [9]: "…yet the software has apparently not been widely used. This is probably largely due to the inability of the software to analyse algorithms expressed in Fortran, or any other standard language".

We have eliminated this drawback by creating a Matlab interface to the method of Miller. Applying the operator overloading based implementation technique of automatic differentiation [8, Chapter 5], we have provided a means of analyzing numerical methods given in the form of Matlab m-functions. In our framework, we can define both straight-line programs and methods with iterative loops and arbitrary branches. Since the possible control paths are handled automatically, iterative methods and methods with pivoting techniques can also be analyzed in a convenient way. Miller originally used the direct search method of Rosenbrock for finding numerical instability. To improve the efficiency of maximizing, we added two more direct search methods [7]: the well known simplex method of Nelder and Mead, and the so called multidirectional search method developed by Virginia Torczon [17].

In Section 2 we give precise definitions of the concepts *straight-line program* and *computational graph*. Two functions can be associated with each straight-line program: the *program function* and the *program function with presence of rounding errors*. The software of Miller can handle only straight-line programs, and uses the Jacobian of these functions to compute the error function ω. A detailed discussion of the formulation of the *program function* and the *program function with presence of rounding errors* is also given in Section 2. We extended Miller's approach to non straight-line programs by operator overloading based automatic differentiation. In Section 3 we discuss the basic idea of the operator overloading approach, and we specify the above program functions (with and without rounding errors) in that generalized case. Once we have the required Jacobian, we compute the error measuring number ω in the same way as Miller did, which is summarized in Section 4. Some technical details about our new software are given in Section 5. Two examples can be found in Section 6.

2 STRAIGHT-LINE PROGRAMS, COMPUTATIONAL GRAPHS AND AUTOMATIC DIFFERENTIATION

Miller's error analyzer treats rounding errors in a machine independent manner. The analysis is not tuned to a particular form of machine number or a particular numerical precision, instead it employs a model of floating point numbers and rounding errors. We use the standard model of the floating point arithmetic, which assumes that the relative error of each arithmetic operation is bounded by the machine rounding unit, and we ignore the possibility of overflow and underflow. The IEEE 754/1985 standard of floating point arithmetic guarantees that the standard model holds for addition, subtraction, multiplication, division and square root. Unfortunately it is not true for the exponential, trigonometric and hyperbolic functions and their inverses. Hence we limit ourselves to numerical algorithms that can be decomposed to the above mentioned five basic operations and unary minus, which is considered error-free.

The roundoff error analyzer method of Miller is based on the first order derivatives of the output with respect to the input and the belonging rounding errors. Our main improvement concerns the automatic differentiation method computing the values for the Jacobian. In order to describe the applied techniques precisely, we need to make clear the concepts of a straight-line program and a computational graph and their role in the applied automatic differentiation method.

First, we introduce the notion of a straight-line program. Informally, a numerical algorithm is a straight-line program if it does not contain branches depending directly or indirectly on the particular input values, and the loops are traversed a fixed number of times. With the loops unrolled, taking the appropriate branches at if-tests and inlining the subroutines – i.e., inserting the content of a subroutine in the place of its call –, one could create an equivalent program containing only sequence of real assignment statements to every straight-line program. By defining straight-line programs and computational graphs, we follow [2, Section 2] with slight modifications.

Now we give the formal definition of a straight-line program Π. Let m, n and t be natural numbers and introduce $X = \{x_1, x_2, \dots, x_n\}$, $V = \{v_1, v_2, \dots, v_m\}$ and $\{\leftarrow, +, -, \times, \div, \mathrm{sqrt}\}$ disjoint sets of n, m and six symbols, respectively, and let $S = \{s_1, s_2, \dots, s_t\}$ be a set of t real numbers. We shall call the elements of S constants, the elements of X inputs and the elements of V intermediate results of the straight-line program Π. A computational sequence C is an m-tuple with elements of the form $v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda$ or $v_\lambda \leftarrow \mathrm{sqrt}\, v'_\lambda$ or $v_\lambda \leftarrow -v'_\lambda$ ($1 \le \lambda \le m$, $\circ_\lambda \in \{+, -, \times, \div\}$), where v'_λ and v''_λ are either elements of V with lower index than λ, or arbitrary elements of the set $S \cup X$. A straight-line program of length m with n inputs, t constants and k outputs is an ordered quintuple $\Pi = (S, X, V, T, C)$, where $S \subset \mathbb{R}$, and X, V are disjoint sets of symbols as above with t, n, m elements, respectively. T is a subset of V with cardinality k, and C is a computational sequence of length m. The elements of T are the outputs of the straight-line program Π.

An interpretation of the straight-line program Π in the domain $\mathbb{R}$ of real numbers is a mapping $J : X \to \mathbb{R}$. If J can be extended to a mapping $S \cup X \cup V \to \mathbb{R}$ (for convenience also denoted by J) in such a way that $J(s) = s$ for every $s \in S$, and for every $1 \le \lambda \le m$ the identity

$$J(v_\lambda) = \begin{cases} J(v'_\lambda) \circ_\lambda J(v''_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda \\ \sqrt{J(v'_\lambda)} & \text{if } c_\lambda = v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda \\ -J(v'_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow -v'_\lambda \end{cases} \tag{1}$$

is defined and holds, then we call the interpretation *consistent* (c_λ denotes the entry of C with index λ). It is obvious that there exists at most one consistent way to extend a given mapping J, and such an extension exists unless we encounter values for which any of the prescribed operations are undefined (attempting to division by zero or taking the square root of a negative number). A consistent interpretation on $\mathbb{R}$ gives rise to a finite sequence of real numbers: $J(x_1)$, $J(x_2), \ldots, J(x_n)$, $J(v_1)$, $J(v_2), \ldots, J(v_m)$. The numbers $d_i = J(x_i)$ are the actual input values, and $w_j = J(v_j)$ are called intermediate values. If $T = \{v_{j_1}, v_{j_2}, \ldots, v_{j_k}\}$, then the interpretation results k outputs $p_1 = w_{j_1}$, $p_2 = w_{j_2}$, $\ldots$, $p_k = w_{j_k}$. We also say that the interpretation computes the outputs $p_1, p_2, \ldots, p_k$ from the inputs $d_1, d_2, \ldots, d_n$.

Let $\Pi = (S, X, V, T, C)$ be a straight line program. We associate to Π a labeled directed acyclic graph $G(\Pi)$ whose set of nodes is $S \cup X \cup V$. The elements of $S \cup X$ represent nodes that are not the starting point of any edges, and the corresponding elements of $S \cup X$ will also be the label of these nodes. The nodes labeled by elements of S are called constant nodes whereas the nodes labeled by elements of X are called input nodes. The elements of the set V are the arithmetic nodes of G. If $v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda$ is an element of the computational sequence C, then G contains two edges from the node v_λ to v'_λ and v''_λ, and the node v_λ is labeled by $\circ_\lambda$. Analogously, if c_λ is of the form of $v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda$ or $v_\lambda \leftarrow -v'_\lambda$ an edge goes from v_λ to v'_λ, and v_λ is labeled by sqrt and $-$, respectively. Finally we add k additional nodes (output nodes) labeled by the elements of T. G contains an edge from each output node to the arithmetic node giving the value of that output. We call $G(\Pi)$ the computational graph associated to the straight-line program Π.

A given computational graph may be associated to different straight-line programs, which, however, compute the same outputs. Hence from now on, we consider the straight-line program and its computational graph equivalent.

We define the program function of the straight-line program on domain $\mathbb{R}$ as follows. Let D be the set of all vectors $d \in \mathbb{R}^n$ for which the corresponding interpretation $J(x_i) = d_i$, $(i = 1, 2, \ldots, n)$ is consistent. For every $d \in D$ the consistent interpretation computes the output vector $p \in \mathbb{R}^k$ resulting in a function $P : D \to \mathbb{R}^k$. This mapping will be called the program function of the straight-line program on domain $\mathbb{R}$.

Let $\Pi = (S, X, V, T, C)$ be a straight-line program as above, and let $\delta \in \mathbb{R}^m$ be the vector of rounding errors hitting each operation in C. The standard model of floating point arithmetic guarantees that $|\delta_j| \leq u$ for all $j = 1, 2, \ldots, m$, where u is small positive number, the machine rounding unit. An interpretation of the straight-line program Π on the domain $\mathbb{R}$ in the presence of error vector δ is a mapping $J : X \to \mathbb{R}$. If J can be extended to a mapping $\widehat{J} : S \cup X \cup V \to \mathbb{R}$ in such a way that $\widehat{J}(s) = s$ for every $s \in S$, and for every $1 \leq \lambda \leq m$ the identity

$$\widehat{J}(v_\lambda) = \begin{cases} \left(\widehat{J}(v'_\lambda) \circ_\lambda \widehat{J}(v''_\lambda)\right) \cdot (1 + \delta_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda \\ \sqrt{\widehat{J}(v'_\lambda)} \cdot (1 + \delta_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda \\ -\widehat{J}(v'_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow -v'_\lambda \end{cases} \tag{2}$$

is defined and holds, then we call the interpretation *consistent.*

Let $\widehat{D}$ be the set of all pairs (d, δ) of vectors $d \in \mathbb{R}^n$, $\delta \in \mathbb{R}^m$ for which the corresponding interpretation $J(x_i) = d_i$ $(i = 1, 2, \ldots, n)$ is consistent in the presence of rounding errors δ. For every $(d, \delta) \in \widehat{D}$ the consistent interpretation computes the output vector $\widehat{p} \in \mathbb{R}^k$ resulting in a function $R : \widehat{D} \to \mathbb{R}^k$, the program function of the straight-line program on domain $\mathbb{R}$ with presence of rounding errors. It is obvious that for all $d \in D$, $(d, 0) \in \widehat{D}$ also holds, and $P(d) = R(d, 0)$.

The analysis of the effects of rounding errors on the evaluation of P at $d_0 \in D$ in floating point arithmetic according to the computational sequence C is based on the derivatives of R at $(d_0, 0) \in \widehat{D}$ with respect to the entries of d and δ. According to the straight-line program representation and Equation (2), the function R is decomposed into the composition of φ_λ $(1 \leq \lambda \leq m)$ elementary functions. For all $1 \leq \lambda \leq m$ φ_λ has the form:

$$\begin{aligned} \varphi_\lambda(x, y, \varepsilon) &= (x \circ_\lambda y)(1 + \varepsilon) && \text{if } c_\lambda = v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda \\ \varphi_\lambda(z, \varepsilon) &= \sqrt{z}(1 + \varepsilon) && \text{if } c_\lambda = v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda \\ \varphi_\lambda(x) &= -x && \text{if } c_\lambda = v_\lambda \leftarrow -v'_\lambda \end{aligned}$$

where $x, y, z, \varepsilon \in \mathbb{R}$, $z \geq 0$, $|\varepsilon| \leq u$. The required derivatives of R are evaluated by automatic (sometimes also called algorithmic) differentiation techniques, i.e., knowing the elementary functions and their derivatives, we apply systematically the chain rule of calculus according to the dependence relation given by the computational graph to build the derivatives of the composition function R. The applied automatic differentiation algorithm requires the differentiability of the elementary functions. Thus we have to restrict the domains of the functions P and R to ensure that no square root of zero will be encountered in (1) and (2) (except the case of v'_λ being a constant, which is not a practical implementation of P). Let $J : X \to \mathbb{R}$ be a consistent interpretation of the straight-line program Π in the domain $\mathbb{R}$. We call the interpretation differentiable if for every entry of the computational sequence of C with the form $v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda$ the corresponding identity $J(v_\lambda) = \sqrt{J(v'_\lambda)}$ holds with $J(v'_\lambda) > 0$.

3 THE OPERATOR OVERLOADING BASED DIFFERENTIATION

Now, we briefly discuss the technique of automatic differentiation that we have applied in our software. To fully understand this section, the reader should be familiar with the object oriented concept of operator overloading and the way it can be used to implement automatic differentiation tools. For the required information please consult [8, Chapter 5].

The software by Miller has its own programming language. The analyzed numerical algorithm must be expressed in that simplified, Fortran-like language. The restrictions of the language guarantee that the defined program can be converted to an equivalent formal straight-line program. A software module called the minicompiler compiles the given algorithm into a straight-line program as the first step of analysis.

The problem is that programs containing iterative loops that may be traversed variable number of times and branches that modify calculation according to various criteria cannot be handled by the Miller's minicompiler. On the other hand the straight-line program and the computational graph is still an accurate model of such a program as it is executed upon a given d input vector. Loops can be unrolled, and only certain branches of the program are actually taken in each given case. By executing any numerical program, we can record the arithmetic operations occurred in the form of a computational sequence as an execution trace of all the operations and their arguments. With the nomination of the input and output variables, we get a straight-line program, for which the derivatives can be calculated in the same way as by Miller's approach. Of course, for different input data we may get different straight-line programs by tracing the execution.

Let $d \in \mathbb{R}^n$ be a vector of input data upon which a given numerical algorithm can be executed without any arithmetic exceptions and run-time errors. Tracing the execution we get a straight-line program $\Pi_d = (S_d, X, V_d, T_d, C_d)$ with program functions P_d in exact arithmetic and R_d in the presence of rounding error. Under our assumptions the interpretation $J(x_i) = d_i$, $(i = 1, 2, \ldots n)$ will be consistent, and if it is also differentiable, then we can calculate the Jacobian of function R_d at $(d, 0) \in \mathbb{R}^{n+m_d}$.

The basic idea of operator overloading approach of automatic differentiation is that we use a special user defined class instead of the built-in floating point type, for which all the arithmetic operators and the square root function are defined (overloaded). Upon performing the operations on the variables of that special type, in addition to computing the floating point result of the operation, the appropriate entry (node) is also added to the computational sequence (graph). Such a class must contain at least two fields (data members): the actual floating point value as in the case of ordinary variables and an identifier that identifies the entry (node) in the computational sequence (graph) corresponding to the given floating point value.

We have developed a Matlab interface for automatic differentiation, which overloads the arithmetic operators and the function `sqrt` for Matlab vectors and matrices of class real (complex arithmetic is not supported). By executing the m-file code of the analyzed numerical algorithm using our special class instead of class real, we get the required computational sequence as a trace of execution. Our approach is much the same as the overloaded automatic differentiation libraries ADMAT (developed by Coleman and Verma [3, 18]) and MAD (by Shaun A. Forth [4]). The main difference is that unlike these toolboxes we also calculate the partial derivatives with respect to the rounding errors in addition to the derivatives with respect to the inputs.

4 MEASURING THE EFFECTS OF ROUNDING ERRORS

The reasonable use of our software requires a basic understanding of how we can measure the effects of rounding errors based on the Jacobian. We compute the error function ω in the same way as Miller did, which can be summarized as follows.

As stated earlier, let $P : D \to \mathbb{R}^k$ ($D \subseteq \mathbb{R}^n$) be the program function of a straight-line program on domain $\mathbb{R}$ and $R : \widehat{D} \to \mathbb{R}^k$ ($\widehat{D} \subseteq \mathbb{R}^n \times \mathbb{R}^m$) the program function with presence of rounding errors. In our case the straight-line program arises from an execution trace upon a specified $d \in \mathbb{R}^n$ input vector. Assume that P is continuously differentiable at d and R is continuously differentiable at $(d, 0)$; we can measure the effects of rounding errors occurred by executing the corresponding computational sequence at d in floating point arithmetic as follows.

Our software can compute several error measuring numbers, but for brevity we shall consider only the approximation of the normwise backward error, which will be denoted by $\omega(d)$. For a deeper insight on the error measuring methods, the reader should consult Miller [14, Section 2, 10, 12] or Miller and Wrathall [15, Chapter 4]. By backward error analysis we compare the effects of rounding errors with the effects of perturbing the input data. $\omega(d)$ is the smallest number for which $R(d,\delta) = P(d+\pi)$ holds for some $\pi \in \mathbb{R}^n$ satisfying $\|\pi\|_\infty \leq \omega(d) \cdot \|d\|_\infty \cdot u$ whenever $\|\delta\|_\infty \leq u$. As δ varies over the m-dimensional cube $K_\infty^m = \{\delta \in \mathbb{R}^m : \|\delta\|_\infty \leq u\}$, the computed value $R(d,\delta)$ varies over the set $\{R(d,\delta) : \|\delta\|_\infty \leq u\}$. For the set of data perturbations

$$K_\infty^n = \{\pi \in \mathbb{R}^n : \|\pi\|_\infty \leq \omega(d) \cdot \|d\|_\infty u\}$$

the exact outputs form the set $\{P(d+\pi) : \pi \in K_\infty^n\}$. For the number $\omega(d)$

$$\{R(d,\delta) : \|\delta\|_\infty \leq u\} \subseteq \{P(d+\pi) : \|\pi\|_\infty \leq \omega(d) \cdot \|d\|_\infty \cdot u\}, \qquad (3)$$

but the two sets have common boundary points. To make ω easy to compute, we apply linear approximation to P and R: $R(d,\delta) \approx R(d,0) + A\delta$, $P(d+\pi) \approx P(d) + B\pi$, where $A = [\partial R_i/\partial \delta_j(d,0)]_{i=1,j=1}^{k,m}$ and $B = [\partial P_i/\partial \pi_j(d)]_{i=1,j=1}^{k,n}$ are the Jacobians. Since $R(d,0) = P(d)$, we can approximate $\omega(d)$ by applying the first order approximations to (3) as the smallest positive number for which

$$\{A\delta : \|\delta\|_\infty \leq 1\} \subseteq \{B\pi : \|\pi\|_\infty \leq \omega(d) \cdot \|d\|_\infty\} \qquad (4)$$

holds. The use of maximum norm makes the value $\omega(d)$ difficult to evaluate, so we work with Euclidean norm instead. Let $D_L = \operatorname{diag}(\|d\|_\infty)$ be the n-by-n diagonal matrix with diagonal elements $\|d\|_\infty$. Then

$$\{B\pi : \|\pi\|_\infty \leq \omega(d) \cdot \|d\|_\infty\} = \{BD_L\pi : \|\pi\|_\infty \leq \omega(d)\}$$

and we calculate $\omega(d)$ as the smallest positive number that makes

$$\{A\delta : \|\delta\|_2 \leq 1\} \subseteq \{BD_L\pi : \|\pi\|_2 \leq \omega(d)\} \tag{5}$$

true. According to (5) $\omega(d)$ can be found by solving the k-by-k generalized eigenvalue problem: $AA^Tx = \lambda BD_LD_L^TB^Tx$, and $\omega(d)$ will be the largest such eigenvalue.

We apply direct search methods to maximize the function $\omega(d)$. Finding large values can be interpreted that the analyzed numerical method is numerically unstable. In addition to the Rosenbrock method originally applied by Miller in [13], we extended the software so that it can also use the multidirectional search method by Torczon [17] and the well known simplex method of Nelder and Mead.

5 MILLER ANALYZER FOR MATLAB

Miller Analyzer for Matlab is a mixed-language software. We kept several routines from the work of Miller et al. [13], which was written in Fortran. These routines perform automatic differentiation using graph techniques on the computational graph, compute error measuring numbers from the derivatives and do the maximization of the error function. The interface between Matlab and the Fortran routines is implemented in C++. The source has to be compiled into a Matlab MEX file, and it is to be called from the command prompt of Matlab. The integration into the Matlab environment makes the use of the program convenient. Matlab provides an easy way of interchanging vectors and matrices with the error analyzer software, and we can immediately verify the results either by testing the analyzed numerical method or by applying some kind of a posteriori roundoff analysis upon the final set of data returned by the maximizer.

Applying the operator overloading technology of Matlab (version 5.0 and above, for details see Register [16]), we have provided a much more flexible way of defining the numerical method to analyze, than the minicompiler did. This new way is based on a user-defined Matlab class called `cfloating`, on which we have defined all the arithmetic operators and the function `sqrt`. The functions defining these operators compute the given arithmetic operations and create an execution trace of the operations as a computational sequence. To analyze a numerical method, we can implement it in the form of Matlab m-file using `cfloating` type instead of the built-in floating point type. However, the `cfloating` class can do more than the original compiler since it does not only register the floating point operations, but also computes their results. During execution the value of real variables are available, which through the overloading of relational operators makes it possible to define numerical methods containing branches based on values of real variables and iterative loops (i.e., algorithms that are not straight-line).

Still, this is not yet enough to analyze the numerical stability of such algorithms, because unlike the minicompiler the generated computational graph may depend on the input data.

Algorithm 1 The original algorithm

```
1: Compilation
2: Generating the computational graph
3: repeat
4:    for data d required by the maximizer
5:    Computing partial derivatives
6:    Evaluating of $\omega(d)$
7: until (stopping criterion of the maximizer)
```

Algorithm 2 The new approach

```
1: repeat
2:    for data d required by the maximizer
3:    Generating the computational graph
4:    Computing partial derivatives
5:    Evaluating of $\omega(d)$
6: until (stopping criterion of the maximizer)
```

Algorithm 1 gives the high level pseudocode of the original program of Miller. Statements (1) and (2) are performed by the minicompiler. As the analyzed method is guaranteed to be straight-line, the generated computational graph is independent from the floating point input vector. The loop given in statements (3)–(7) is executed by the error analyzer program. The program computes the partial derivatives and the stability measuring number for every d input set of data required by the maximizer. The program terminates if the stopping criterion of the numerical maximizer is fulfilled. Algorithm 2 illustrates our new approach. In this case the compilation phase is omitted since the Matlab interpreter executes the m-file directly. The problem is that the generated computational graph is not necessarily independent from the input data. Therefore, the process that builds the computational graph had to be inserted into the main loop (Algorithm 2, statement (3)). In this way our program is able to analyze the numerical stability of algorithms that are not straight-line.

6 GAUSSIAN ELIMINATION: AN EXAMPLE

Using the original version of the method [13], Miller analyzed the numerical stability of Gaussian elimination solving the linear system $Ax = b$ ($A \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$)

without pivoting and with partial pivoting (for details see [14, Section 4] and [15, Chapter 5.2]). Our software, which is available on `http://phd.uni-obuda.hu/images/milleranalyzer.zip`, can easily reproduce the results obtained by Miller. For details about the use of the software see our User's Manual [5].

We consider first the procedure without pivoting.

Algorithm 3 Gaussian elimination

```
function b = gauss(A, b)
% Gaussian elimination
$[n, m]$ = size(A);
assert($n$==$m$, 'A is not a square matrix!');
$m$ = numel(b);
assert($n$==$m$, 'b must have as many elements as the columns of A!');
%
% Elimination
for $k = 1{:}n - 1$
   for $i = k + 1{:}n$
      amult = A($i, k$)/A($k, k$);
      A($i, k + 1{:}n$) = A($i, k + 1{:}n$) − amult ∗ A($k, k + 1{:}n$);
      b($i$) = b($i$) − amult ∗ b($k$);
   end
end
%
% Back substitution
for i = n : -1 : 1
   b($i$) = (b($i$) − A($i, i + 1{:}n$) ∗ b($i + 1{:}n$))/$A(i, i)$;
end
```

Algorithm 3 shows an m-file implementation appropriate for analysis. The software can easily find linear systems for which ω is extremely large. We fixed the size of the problem at $n = 4$. Started from a randomly chosen data set, the Rosenbrock method located:

$$A \approx \begin{bmatrix} 0.7447 & 0.1774 & 0.5546 & -0.0404 \\ 0.7136 & 0.1681 & 0.5303 & 0.9408 \\ 0.7440 & 0.8149 & 0.9112 & 0.5309 \\ 1.0416 & 0.1674 & -0.6000 & 0.7108 \end{bmatrix}, \qquad b \approx \begin{bmatrix} 0.8414 \\ -0.4787 \\ 0.3505 \\ -0.2878 \end{bmatrix},$$

where $\omega(A, b) \approx 2.1283e{+}011$. Matlab's condition estimation function gives $\operatorname{cond}(A) \approx 5.6179$, so Gaussian elimination without pivoting can be unstable at very well-conditioned data.

Consider Algorithm 4 implementing Gaussian elimination with row interchanges (partial pivoting). Partial pivoting is performed from line (10) to (13). In line (10) we find the pivoting element with maximal absolute value using the built-in Matlab functions `max` and `abs`. On the other hand, the function `value` is designed especially

Algorithm 4 Gaussian elimination with partial pivoting

```
 1: function b = gpp(A, b)
 2: % Gaussian elimination with partial pivoting
 3: $[n, m]$ = size(A);
 4: assert($n$==$m$, ‘A is not a square matrix!’);
 5: $m$ = numel(b);
 6: assert($n$==$m$, ‘b must have as many elements as the columns of A!’ );
 7: %
 8: % Elimination
 9: for $k = 1{:}n - 1$
10:     [maxval, maxi] = max(abs(value(A($k{:}n, k$))));
11:     maxi = maxi + $k$ − 1;
12:     A([$k$, maxi] , $k{:}n$) = A([maxi, $k$] , $k{:}n$);
13:     b([$k$, maxi]) = b([maxi, $k$]);
14:     for $i = k + 1{:}n$
15:         amult = A($i, k$)/$A(k, k)$;
16:         A($i, k + 1{:}n$) = A($i, k + 1{:}n$) − amult ∗ A($k, k + 1{:}n$);
17:         b($i$) = b($i$) − amult ∗ b($k$);
18:     end
19: end
20: %
21: % Back substitution
22: for $i = n{:} - 1{:}1$
23:     b($i$) = (b($i$) − A($i, i + 1{:}n$) ∗ b($i + 1{:}n$))/A($i, i$);
24: end
```

to work with variables of `cfloating` type. If B is a `cfloating` array, $C = \mathrm{value}\,(B)$ returns the floating point value of B, and C will be a built-in typed double array with the same size as B. Automatic (implicit) conversion of `cfloating` to double is not allowed, but with value we can make explicit conversion. After we have gained access to the floating point values, we can use the function `abs`, which is not defined on `cfloating` type. Being the row index of the pivoting element determined, we interchange the appropriate rows in lines (12) and (13). For Algorithm 4 and $n = 4$ the maximizer was not able to push ω above 6.0, which is in accordance with the well-known fact that the Gaussian elimination with partial pivoting is backward stable.

REFERENCES

[1] Abaffy, J.—Spedicato, E.: ABS Projection Algorithms: Mathematical Techniques for Linear and Nonlinear Equations. Prentice-Hall, Inc., Upper Saddle River, NJ USA, 1989.

[2] Castaño, B.—Heintz, J.—Llovet, J.—Martínez, R.: On the Data Structure Straight-Line Program and Its Implementation in Symbolic Computation. Math. Comput. Simul., Vol. 51, 2000, No. 5, pp. 497–528.

[3] Coleman, T. F.—Verma, A.: ADMAT: An Automatic Differentiation Toolbox for Matlab. Tech. Rep., Computer Science Department, Cornell University 1998.

[4] Forth, S. A.: An Efficient Overloaded Implementation of Forward Mode Automatic Differentiation in Matlab. ACM Trans. Math. Softw., Vol. 32, 2006, No. 2, pp. 195–222.

[5] Gáti, A.: Miller Analyser for Matlab, User's Manual. Available on: `http://phd.bmf.hu/images/milleranalyzer.zip`.

[6] Gáti, A.: Automatic Error Analysis With Miller's Method. Miskolc Math. Notes, Vol. 5, 2004, No. 1, pp. 25–32.

[7] Gáti, A.: The Upgrading of the Miller-Spooner Roundoff Analyser Software. In: A. Pethő and M. Herdon (Eds.): Proceedings of IF 2005, Conference on Informatics in Higher Education, Debrecen, 2005, University of Debrecen, Faculty of Informatics (in Hungarian).

[8] Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, USA, 2000.

[9] Higham, N. J.: Accuracy and Stability of Numerical Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[10] Miller, W.: Computer Search for Numerical Instability. J. ACM, Vol. 22, 1975, No. 4, pp. 512–521.

[11] Miller, W.: Software for Roundoff Analysis. ACM Trans. Math. Softw., Vol. 1, 1975, No. 2, pp. 108–128.

[12] Miller, W.: Roundoff Analysis by Direct Comparison of Two Algorithms. SIAM Journal on Numerical Analysis, Vol. 13, 1976, No. 3, pp. 382–392.

[13] Miller, W.—Spooner, D.: Algorithm 532: Software for Roundoff Analysis [z]. ACM Trans. Math. Softw., Vol. 4, 1978, No. 4, pp. 388–390.

[14] Miller, W.—Spooner, D.: Software for Roundoff Analysis, II. ACM Trans. Math. Softw., Vol. 4, 1978, No. 4, pp. 369–387.

[15] Miller, W.—Wrathall, C.: Software for Roundoff Analysis of Matrix Algorithms. Academic Press, New York 1980.

[16] Register, A. H.: A Guide to Matlab Object-Oriented Programming. Chapman & Hall/CRC, 2007.

[17] Torczon, V.: On the Convergence of the Multidirectional Search Algorithm. SIAM Journal on Optimization, Vol. 1, 1991, No. 1, pp. 123–145.

[18] Verma, A.: ADMAT: Automatic Differentiation in Matlab Using Object Oriented Methods. In: M. E. Henderson, C. R. Anderson, and S. L. Lyons (Eds.): Object Oriented Methods for Interoperable Scientific and Engineering Computing, Proceedings of the 1998 SIAM Workshop, Philadelphia, 1999, SIAM, pp. 174–183.

Attila Gáti received the M. Sc. degree in business economics and the B. Sc. (summa cum laude) degree in computer science from the University of Miskolc, Hungary, in 2000 and 2003, respectively. He is currently pursuing the Ph. D. degree at the Doctoral School for Applied Information Science of the University of Óbuda under the supervision of Aurél Galántai and László Horváth. From 2006 to 2008, he was an assistant lecturer in the Department of Analysis, University of Miskolc. Since 2008, he has been with the R & D division at ARH corporation, Hungary, as a researcher and algorithm developer working in the field of image processing and computer vision.