Óbuda University

PhD thesis

Automatic roundoff error analysis of numerical algorithms

by
Attila Gáti

Supervisors:

Prof. Dr. László Horváth
Prof. Dr. István Krómer

Applied Informatics Doctoral School

Budapest, 2013

Contents

1 Introduction

The idea of automatic error analysis of algorithms and/or arithmetic expressions is as old as the scientific computing itself and originates from Wilkinson (see Higham [39]), who also developed the theory of floating point error analysis [80], which is the basic of today's floating point arithmetic standards ([65], [62]). There are various forms of automatic error analysis with usually partial solutions to the problem (see, e.g. Higham [39],[21]). The most impressive approach is the interval analysis, although its use is limited by the technique itself (see. e.g. [59], [60], [61], [43], [44], [76], [39]). Most of the scientific algorithms however are implemented in floating point arithmetic and the users are interested in the numerical stability or robustness of these implementations. The theoretical investigations are usually difficult, require special knowledge and they do not always result in conclusions that are acceptable in practice (for the reasons of this, see, e.g. [79], [80], [11], [39], [21]). The common approach is to make a thorough testing on selected test problems and to draw conclusions upon the test results (see, e.g. [69], [70], [77]). Clearly, these conclusions depend on the "lucky" selection of the test problems and in any case require a huge amount of extra work. The idea of some kind of automatization arised quite early.

The effective tool of linearizing the effects of roundoff errors has been widely applied since the middle of the 70's ([52], [53], [54], [55], [56], [57], [58], [73], [47], [72]). Given a numerical method the computed value is a function $R(d,\delta)$ ($R : \mathbb{R}^{n+m} \to \mathbb{R}^k$), where $d \in \mathbb{R}^n$ is the input vector, and δ is the vector of individual relative rounding errors on the m arithmetic operations ($\delta \in \mathbb{R}^m$, $\|\delta\|_\infty \leq u$, where u is the machine rounding unit). Considering the first order Taylor expansion of R at $\delta = 0$ we have for all $j = 1, 2, \ldots, k$

$$R_j(d,\delta) = R_j(d,0) + \sum_{i=1}^{m} \frac{\partial R_j}{\partial \delta_i}(d,0) + o(\|\delta\|) \tag{1}$$

From (1) an approximate error bound for the forward error immediately follows:

$$|R_j(d,\delta) - R_j(d,0)| \lessapprox \sigma(d)\, u \qquad (j = 1, 2, \ldots, k) \tag{2}$$

where

$$\sigma(d) = \max_j \sum_{i=1}^{m} \left| \frac{\partial R_j}{\partial \delta_i}(d,0) \right| \tag{3}$$

The need for computing the partial derivatives accurately and effectively motivated the design of new automatic differentiation techniques. Since R is given as an algorithm, it is decomposed to basic arithmetic operations and elmentary functions. By automatic differentiation we apply systematically the chain rule of calculus according to the relation "being an operand of" among the operations and elementary

functions to compute the derivatives of the composition function R. In contrast to numerical differentiation, automatic differentiation is free of truncation errors. On the other hand it has the drawback, that we can not treat numerical methods as black boxes, because we need the structural knowledge of their decomposition to basic operations and elementary function. The data structure commonly used to represent the structure of a numerical method is the computational graph.

Let us consider a simple example from [53]:

$$\begin{aligned} v &\leftarrow d \times d, \\ w &\leftarrow d + v, \\ x &\leftarrow d \times v, \\ y &\leftarrow w + x, \\ z &\leftarrow y - v. \end{aligned}$$

with a single input d, and a single output z. The corresponding computational graph is shown in Figure 1.

Concerning the computational graph related error analysis Chaitin-Chatelin and Frayssé [11] gave the following summary.

"The stability analysis of an algorithm $x = G(y)$ with the implementation $G_{alg} = \Pi G^{(i)}$ depends on the partial derivatives $\frac{\partial G^{(i)}}{\partial y_j}$ computed at the various nodes of the computational graph (see § 2.6). The partial derivatives show how inaccuracies in data and rounding errors at each step are propagated through the computation. The analysis can be conducted in a forward (or bottom-up) mode or a backward (or top-down) mode. There have been several efforts to automate this derivation. One can cite the following:

1. *Miller and Spooner (1978);*
2. *the B-analysis: Larson, Pasternak, and Wisniewski (1983), Larson and Sameh (1980);*
3. *the functional stability analysis: Rowan (1990);*
4. *the automatic differentiation (AD) techniques:*
   Rall (1981), Griewank (1989), Iri (1991)"

Miller developed his approach in several papers ([49], [50], [51], [52], [53], [54], [55], [56], [57], [58]). The basic idea of Miller's method is the following. 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

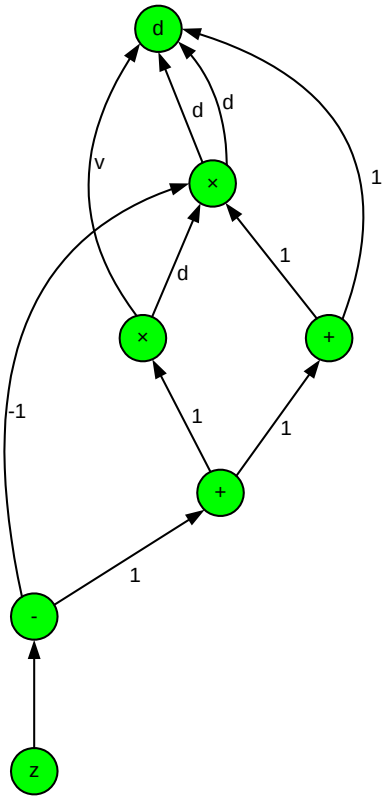

Figure 1: A computational graph.

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.

The Miller algorithm was implemented in Fortran language (actually in FORTRAN IV) by Webb Miller and David Spooner [56], [57] in 1978. More information on the use of the software by Miller and its theoretical background can be found in [52], [53], [54] and [56]. The software is in the ACM TOMS library with serial number 532 [57].

In the book of Miller and Wrathall [58], 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

- inversion of triangular matrices
- the linear least-squares problem by solving the normal equations
- Gaussian elimination without pivoting and with partial pivoting,
- the Gauss-Jordan elimination,
- the Gaussian elimination with iterative improvement,
- the Cholesky factorization,
- Cholesky factorization after rank-one modification,
- the classical and modified Gram-Schmidt methods,
- the application of normal equations and Householder reflections for linear least squares problem,
- rational QR method, downdating of the QR factorization,
- the characteristic polynomial computation by the Faddeev method,
- symmetric matrix representations.

Miller's approach was further developed for relative error analysis by Larson, Pasternak, and Wisniewski [47]. Larson et al. [47] say (pp. 125–126) the following.

"To perform the error analysis, the algorithm being analyzed is represented as a directed graph with each numeric quantity being a node in this graph. Directed arcs lead from operands to their results. Each node's value is subject to error, called local relative error. This error affects those subsequent nodes that are computed using the contamined node as an operand. All of the local relative errors together are used to produce a total relative error for each node. From the directed graph, a system of equations relating to the local and total relative errors is generated.

Our software is related to that of Miller [4] and Miller and Spooner [5], which analyzes the numerical stability of algorithms using absolute error analyses. In particular, the input formats are similar, and the minicompiler, a program for translating a FORTRAN-like language into the data format for the code [5], can be used to specify the computational graph of the algorithm being analyzed."

Rowan [72] considers the numerical algorithms as black boxes. The two main elements of his approach is a function that estimates a lower bound on the backward error, and a new optimization method (called subplex and based on the Nelder-Mead simplex method) that maximizes the function. A numerical method is considered unstable if the maximization gives a large backward error. The FORTRAN software requires two user-supplied Fortran subprograms; one implements the algorithm to be tested in single precision, and the other provides a more accurate solution, typically by executing the same algorithm in double precision.

Bliss [5], Bliss et al [6] developed Fortran preprocessor tools for implementing the local relative error approach of Larson and Sameh. They also used some statistical techniques as well. The structure of their "instrumentation environment" is shown in Figure 2 taken from Bliss et al [6].

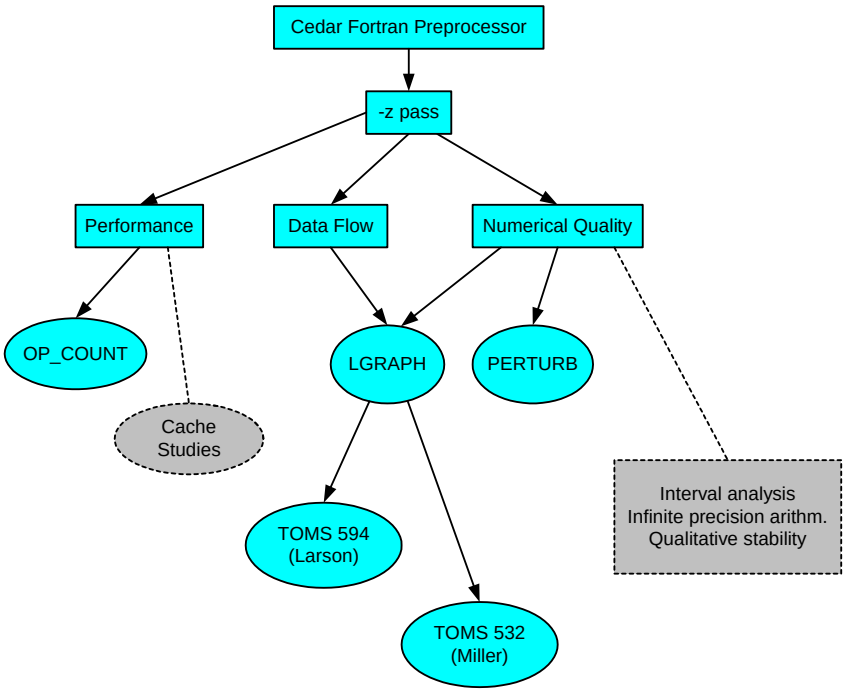

Figure 2: The "instrumentation environment" of Bliss et al [6].

Error analysis tools based on the differential error model have been developed not only for numerical but also for symbolic computing environments. Stoutemyer [73] was the first to use computer algebra for error-analysis software. He used the accumulated error expressions from the differential error model to determine a bound on the error and proposes analyzing the condition expressions to find points where instability is a problem. For other authors and approaches, see Mutrie et

al. [63], Krämer [43], [44], or [39].

Finally, we have to mention the PRECISE toolbox of Chaitin–Chatelin et al. [11], [21] that embodies an entirely different strategy (somewhat similar to the naive testing, but much more sophisticated). "*PRECISE allows experimentation about stability by a straightforward randomisation of selected data, then lets the computer produce a sample of perturbed solutions and associated residuals or a sample of perturbed spectra.*" ([11], p. 104).

Upon the basis of corresponding literature, the Miller approach seems to be the most advanced although Miller's method has several setbacks. 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 contains 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 [39] points out that the special language and its restrictions as the greatest disadvantage of the software:

"*... 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*".

Unfortunately, this can be said more or less on the other developments that followed the Miller approach.

The aim of my research work is to improve Miller's method to a level that meet today's requirements. I improved and upgraded Miller's method in two main steps. The first step was a F77 version usable in PC environment with standard F77 compilers such as the GNU and Watcom compilers [24], [25], [26]. This version was able to handle algorithms with a maximum of 3000 inputs, and a maximum of 1000 outputs, and operations of a maximum of 50000 while the original Miller program was able to handle a maximum of 30 inputs, 20 outputs and 300 operations [24], [25], [26]. However even this new version used the simplified Fortran like language of Miller which is considered as a major problem by Higham and others. Using the recently available techniques such as automatic differentiation, object oriented programming and the widespread use of MATLAB, I have eliminated the above mentioned drawbacks of Miller's method by creating a Matlab interface. Applying

the operator overloading based implementation technique of automatic differentiation Griewank [37] and Bischof etal [4] we have provided 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 [30], [28], [29], [35]: the well known simplex method of Nelder and Mead, and the so called multidirectional search method developed by Torczon [75].

In the thesis we present a significantly improved and partially reconstructed Miller method by designing and developing a new Matlab package for automatic roundoff error analysis. 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. Since the analyzed numerical algorithm can be given in the form of a Matlab m-file, our software is easy to use.

We used the software package to examine the stability of some ABS methods [1], [2], namely the implicit LU methods and several variants of the Huang method [25]. The obtained computational results agreed with the already known facts about the numerical stability of the ABS 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 and the famous MGS (modified Gram-Schmidt) method (see [58], [2] or [36]).

We also tested three fast matrix multiplication algorithms with the following results:

1. The classical Winograd scalar product based matrix multiplication algorithm of $O(n^3)$ operation cost is highly unstable in accordance with the common belief, that has never been published.

2. Both the Strassen and Winograd recursive matrix multiplication algorithms of $O(n^{2.81})$ operation costs are numerically unstable.

3. The comparative testing indicates that the numerical stability of Strassen's algorithm is somewhat better than those of Winograd.

Upon the basis of our testing, we may think that the new software called Miller Analyzer for Matlab will be useful for numerical people or algorithm developers to analyze the effects of rounding errors. The results of my research were published in the works [24], [25], [26], [27], [30], [28], [29], [31], [32], [33], [34], [35].

2 The original Miller-Spooner software and its limitations

Using the software, the numerical method to analyse must be expressed in a special and simplified Fortran-like language. The language allows the usage of integer and real types. The variables can be scalars, or one or two dimensional arrays of both types. We can construct for-loops and if tests, but only that are not based on the values of real variables. This means, that we can only define algorithms, in which the flow of control does not depend on the value of floating point variables. To analyse methods that contain branching based on the value of real variables the possible pathes through comparisions have to be treated separately. This can be realized by constrained optimization. We constrain the search for maximum to such input vectors, by which the required path of control is realized. The constraints can be specified through a user-defined Fortran subroutine, called POSITV.

The software package consists of three programs, a minicompiler and two error analyser programs. The minicompiler takes as data the numerical method to analyse, written in a special programming language, and produces as output a translation of that program into a series of assignment statements. The output is presented in a readable form for the user, and in a coded form for use as input to the error analyser programs. One of the error analyser programs (i) is for deciding numerical stability of a single algorithm, and the other (ii) is for comparing two algorithms. The input for the error-testing programs is arranged as follows: (1) the output of the minicompiler (For program (i), it is the compiled version of the algorithm to analyse. In the case of program (ii) we must provide the compiled version of the two algorithms to compare), (2) a list of initial data for the numerical maximizer, (3) the code of the chosen error-measuring value, (4) target value for the maximizer. The programs return with the final set of data found by the maximizer routine and with the value of the chosen error-measuring number at this set of data. If program (i) diagnoses instability (the value of the error-measuring number at the final set of data exceeds the target value) then the condition number is also computed at the final set of data.

The minicompiler does its job in two phases: the compilation phase and the interpreter-like execution phase. In the compilation phase the program makes syntactical and semantical analysis, and generates an intermediate code (a coded representation of the algorithm to analyse) for the interpreter routine. The interpreter executes this intermediate code in a special way. All integer expressions are actually evaluated in order to perform the correct number of iterations of for-loops, and to interpretively perform if-then tests. In contrast no actual real arithmetic computation is done. The interpreter does not evaluate the floating point operations, only registers them in the form of assignment statements. Throughout all

real variables are treated symbolically as being the n-th input value, intermediate value, or real constant. The restrictions of the language ensures, that the produced series of assignment statements, which is actually a representation of the algorithms computational graph, is independent from the input data. Based on the computational graph generated by the minicompiler, the error analyser programs compute the partial derivatives of the output of the analyzed numerical method with respect to the inputs and individual rounding errors in every data required by the maximizer routine. The differentiation is realized through graph algorithms, so the applied method is not numerical derivation. The computation of stability measuring function is based on these partial derivatives. The direct search method used for maximizing is the Rosenbrock method.

The Miller-Spooner roundoff error analyser software was written in Fortran IV for the IBM 360/370 series, and its final version were issued in 1979, which can be downloaded from the NETLIB repository. Without any modifications, it can not be used in a PC environment with the most widely used compilers based on the Fortran 77 standard (GNU compiler, Watcom, Lahey, etc.), because there are several non-standard solutions in the source. By compilation we get several error messages, but even if we are able to correctly eliminate these errors we get a program, which simply does not work. The semantical problems are arising from two facts. First, the software uses EBCDIC character encoding instead of ASCII. Second, the correct working of the program requires, that some local variables be static i.e., they have to keep their values between two function or procedure calls. Unlike in older Fortran dialects this is not guaranteed by the Fortran 77 standard.

In addition, the original version bounds the user to extraordinary strict limitations in the size of the algorithm to analyse. The number of inputs cannot exceed 30, the number of outputs must be smaller than 20, and the total number of inputs and real arithmetic operations may not exceed 300. The limits are due to small values of constants determining the sizes of static arrays used in the software. Another weak point of the sofware is maximizing, because it can be done only by direct search, as the function ω is not differentiable. Direct search methods are heuristic, so sometimes the program can provide misleading results. Failure of the maximizer to find large values of ω does not guarentee that none exists. The method tends to be optimistic: unstable methods may appear to be stable.

Finally, we have to note that the software is rather inconvenient to use. The input stream for the error analyser programs must be made out by hand. If we want to perform constrained optimization through the POSITV routine, we have to relink the program, and the method of analysing algorithms containing branching based on the values of real variables by constrained optimization can be rather complicated in some cases.

The original Miller program can handle a maximum of 30 inputs, 20 outputs

and 300 operations. My F77 variant of Miller's algorithm can handle algorithms with a maximum of 3000 inputs, 1000 outputs, and 50000 arithmetic operations [25].

3 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 (see, e.g. Muller et al. [62]). Unfortunately it is not true for the exponential, trigonometric, 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 Castano et al. [10] 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 $X = \{x_1, x_2, \ldots, x_n\}$, $V = \{v_1, v_2, \ldots, v_m\}$ and $\{\leftarrow, +, -, \times, \div, \mathrm{sqrt}\}$ disjoint sets of n, m and six symbols, respectively, and let $S = \{s_1, s_2, \ldots, s_t\}$ 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 $v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda$ or $v_\lambda \leftarrow -v'_\lambda$ ($1 \leq \lambda \leq m$, $\circ_\lambda \in \{+, -, \times, \div\}$), 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 $1 \leq \lambda \leq m$ the identity

$$J(v_\lambda) = \begin{cases} J(v'_\lambda) \circ_\lambda J(v''_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda \\ \sqrt{J(v'_\lambda)} & \text{if } c_\lambda = v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda \\ -J(v'_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow -v'_\lambda \end{cases} \tag{4}$$

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 $p_1 = w_{j_1}$, $p_2 = w_{j_2}$, ..., $p_k = w_{j_k}$. 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 all 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 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

$$\widehat{J}(v_\lambda) = \begin{cases} \left(\widehat{J}(v'_\lambda) \circ_\lambda \widehat{J}(v''_\lambda)\right) \cdot (1 + \delta_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow v'_\lambda \circ_\lambda v''_\lambda \\ \sqrt{\widehat{J}(v'_\lambda)} \cdot (1 + \delta_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow \operatorname{sqrt} v'_\lambda \\ -\widehat{J}(v'_\lambda) & \text{if } c_\lambda = v_\lambda \leftarrow -v'_\lambda \end{cases} \tag{5}$$

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 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 (5), 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:

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

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 (4) and (5) (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 $J(v'_\lambda) > 0$.

4 Numerical stability measures

The numerical stability measures used in Miller's algorithm were developed by Miller in a sequence of papers [49], [51], [54], [56] and [58]. The purpose of this chapter is to give an understanding of how the software measures the effects of rounding errors, and how the error measuring function is formulated. Actually we can perform analysis based on several error measuring numbers (various ways of assigning ω), and beside analysing the propagation of rounding errors in a single algorithm we can also compare the numerical stability of two competiting numerical methods, which neglecting rounding errors compute the same values. The method 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 rounding errors. The applied model is 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 allowed arithmetic operations are addition, substraction, multiplication, division, square root and unary minus, but unary minus is always considered to be error free.

First let us consider the case of analysing a single algorithm. The software provides four stability-measuring numbers, $JW_E(d)$, $JW_L(d)$, $WK_E(d)$, $WK_L(d)$, which measure the minimum problem perturbation equivalent to rounding error in a given program at data d. Other numbers $ER_E(d)$ and $ER_L(d)$ use weaker comparison of computed and exact solutions. To give the precise definition of these error measuring functions we shall need some notations. Suppose that $S \subseteq \mathbb{R}^n$, $T \subseteq S$ and $f : S \to \mathbb{R}^m$. Notation $f(T)$ will denote the set: $f(T) = \{f(t) \in \mathbb{R}^m : t \in T\}$. We extend some operations on vectors to sets of vectors: let $S, S_1, S_2 \subseteq \mathbb{R}^n$, $v \in \mathbb{R}^n$ be a vector, $\alpha \in \mathbb{R}$ a scalar and $A \in \mathbb{R}^{m \times n}$ an

m by n matrix, then the sets αS, AS, $v+S$, S_1+S_2 will be defined as:

$$\begin{aligned} \alpha S &= \{\alpha x : x \in S\} \\ AS &= \{Ax : x \in S\} \\ v+S &= \{v+x : x \in S\} \\ S_1+S_2 &= \{x+y : x \in S_1, y \in S_2\}. \end{aligned}$$

Suppose R is a numerical algorithm applying only the allowed arithmetic operations mentioned above. If the operations are performed in floating point arithmetic, with the assumption of the standard model, then the computed value is a function $R(d,\delta)$ ($R : \mathbb{R}^{n+m} \to \mathbb{R}^k$), where $d \in \mathbb{R}^n$ is the input vector, and δ is the vector of individual relative rounding errors on the m arithmetic operations ($\delta \in \mathbb{R}^m$, $\|\delta\|_\infty \leq u$, where u is the machine rounding unit). Let $f = R(d,0)$ be the exact solution. $K^{(n)} = \{x \in \mathbb{R}^n : \|x\|_\infty \leq 1\}$ will denote the maximum norm unit ball. For all $d \in S$ $D_E(d) = diag(d)$ will be a diagonal matrix with d_i as its i-th diagonal entry, and $D_L(d) = diag(\|d\|_\infty)$ will stand for a diagonal matrix with the largest entry of the input vector as all of its diagonal elements. We similarly define diagonal matrices for the exact result: $F_E = diag(f(d))$, and $F_L = diag(\|f(d)\|_\infty)$. Using these notations we can define the stability measuring numbers as follows:

$$\begin{aligned} JW_E(d) &= \inf\left\{\alpha \geq 0 : R(d, uK^{(m)}) \subseteq f(d + \alpha u D_E K^{(n)})\right\} \\ JW_L(d) &= \inf\left\{\alpha \geq 0 : R(d, uK^{(m)}) \subseteq f(d + \alpha u D_L K^{(n)})\right\} \\ WK_E(d) &= \inf\left\{\alpha \geq 0 : R(d, uK^{(m)}) \subseteq f(d + \alpha u D_E K^{(n)}) + \alpha u F_E u K^{(k)}\right\} \\ WK_L(d) &= \inf\left\{\alpha \geq 0 : R(d, uK^{(m)}) \subseteq f(d + \alpha D_L u K^{(n)}) + \alpha u F_L K^{(k)}\right\} \\ ER_E(d) &= \inf\left\{\alpha \geq 0 : \left\|R(d, uK^{(m)}) - f(d)\right\|_\infty \subseteq \left\|f(d + \alpha u D_E K^{(n)}) - f(d)\right\|_\infty\right\} \\ ER_L(d) &= \inf\left\{\alpha \geq 0 : \left\|R(d, uK^{(m)}) - f(d)\right\|_\infty \subseteq \left\|f(d + \alpha u D_L K^{(n)}) - f(d)\right\|_\infty\right\}. \end{aligned}$$

In the definitions $R(d, uK^{(m)})$ is the set of possible computed values under the assumption of the standard model of floating point arithmetic. On the right-hand side of the subset sign in the definitions we have compact sets, which implies that the infimums are achieved. So we can say that for any $\delta \in \mathbb{R}^m$, $\|\delta\|_\infty \leq u$, there exists a vector $\pi \in \mathbb{R}^n$, $|\pi_i| \leq JW_E(d) \cdot |d_i| \cdot u$ $(i = 1, 2, \ldots, n)$ for which $R(d,\delta) = f(d+\pi)$ holds, and $JW_E(d)$ is the smallest such number. This means that $JW_E(d) \cdot u$ is the least upper bound for the componentwise relative backward error. Analogous fact holds for $JW_L(d)$ with the relation $|\pi_i| \leq JW_L(d) \cdot \|d\|_\infty \cdot u$ $(i = 1, 2, \ldots, n)$ for the vector π, so $JW_L(d) \cdot u$ is the least upper bound for the normwise relative backword error. In the case of the WK_X measures ($X = E$, or L) we also allow perturbations in the output space. For any $\delta \in \mathbb{R}^m$, $\|\delta\|_\infty \leq u$, there exist a vector $\pi \in \mathbb{R}^n$, $|\pi_i| \leq WK_E(d) \cdot |d_i| \cdot u$ $(i = 1, 2, \ldots, n)$ and a vector

$\varphi \in \mathbb{R}^k$ $|\varphi_i| \leq WK_E(d) \cdot |f_i(d)| \cdot u$ $(i = 1, 2, \ldots, k)$ for which $R(d, \delta) = f(d + \pi) + \varphi$ holds and $WK_E(d)$ is the least such number. So $WK_E(d) \cdot u$ can be interpreted as the least upper bound for the relative mixed forward-backward error measured componentwise. As the same is true for $WK_L(d)$ with $|\pi_i| \leq WK_L(d) \cdot \|d\|_\infty \cdot u$ $(i = 1, 2, \ldots, n)$ and $|\varphi_i| \leq WK_L(d) \cdot \|f(d)\|_\infty \cdot u$ $(i = 1, 2, \ldots, k)$, $WK_L(d)$ is the least upper bound for the normwise relative mixed forward-backward error. Notice that the relations $WK_L(d) \leq WK_E(d) \leq JW_E(d)$ and $WK_L(d) \leq JW_L(d) \leq JW_E(d)$ will hold.

The numbers JW_X and WK_X require that the computed solution be obtained at (or close to) exact solution. ER_X measures how much the data must be perturbed to create an error as large as that in the computed solution. In the case of ER_E it is true, that for any $\delta \in \mathbb{R}^m$, $\|\delta\|_\infty \leq u$, there exists a vector $\pi \in \mathbb{R}^n$, $|\pi_i| \leq ER_E(d) \cdot |d_i| \cdot u$ $(i = 1, 2, \ldots, n)$ for which $\|R(d, \delta) - f(d)\|_\infty = \|f(d + \pi) - f(d)\|_\infty$ holds, and $ER_E(d)$ is the least such number. The same is true for $ER_L(d)$ with the relation $|\pi_i| \leq ER_L(d) \cdot \|d\|_\infty \cdot u$ $(i = 1, 2, \ldots, n)$ for the vector π. The inequalities $ER_L(d) \leq ER_E(d)$ and $ER_X(d) \leq JW_X(d)$ $(X = E,$ or $L)$ are obvious.

Now let us take a glance at the error-measuring numbers used to compare the stability properties of two algorithms. Let Q and R be two algorithms for evaluating the function f, and the computed values at data d, with rounding errors $\gamma \in \mathbb{R}^l$ and $\delta \in \mathbb{R}^m$ be denoted by $Q(d, \gamma)$ and $R(d, \delta)$. The error measuring functions are defined as:

$$\begin{aligned} JW_{R/Q} &= \inf\left\{\alpha \geq 0 : R(d, uK^{(m)}) \subseteq Q(d, \alpha u K^{(l)})\right\} \\ ER_{R/Q} &= \inf\left\{\alpha \geq 0 : \left\|R(d, uK^{(m)}) - f(d)\right\|_\infty \subseteq \left\|Q(d, \alpha u K^{(l)}) - f(d)\right\|_\infty\right\}. \end{aligned}$$

We can say, that for any $\delta \in \mathbb{R}^m$, $\|\delta\|_\infty \leq u$, there exists a vector $\gamma \in \mathbb{R}^l$, $|\gamma_i| \leq JW_{R/Q}(d) \cdot u$ $(i = 1, 2, \ldots, n)$ for which $R(d, \delta) = Q(d, \gamma)$ holds, and $JW_{R/Q}(d)$ is the smallest such number. Regarding the standard model of machine arithmetic the computed values given by R can be achived with rounding errors in Q bounded by $JW_{R/Q}(d) \cdot u$. In the case of $ER_{R/Q}$ it is true, that for any $\delta \in \mathbb{R}^m$, $\|\delta\|_\infty \leq u$, there exists a vector $\gamma \in \mathbb{R}^l$, $|\gamma_i| \leq ER_{R/Q}(d) \cdot u$ $(i = 1, 2, \ldots, n)$ for which $\|R(d, \delta) - f(d)\|_\infty = \|Q(d, \gamma) - f(d)\|_\infty$ holds, and $ER_{R/Q}(d)$ is the smallest such number. $ER_{R/Q} \cdot u$ can be interpreted as the least upper bound for the relative roundoff errors in Q, by which Q produces output with as large forward error as does R in floating point arithmetic with machine rounding unit u. The inequalities $ER_{Q/R}(d) \leq JW_{Q/R}(d)$ and $\frac{JW_X^Q}{JW_X^R} \leq JW_{Q/R}(d)$ $(X = E,$ or $L)$ will also hold.

The software gives first order approximation of the above error measuring numbers based on the partial derivatives of R with respect to the input vector d and the individual rounding errors δ, or in the case of comparing two algorithms the

derivatives of Q and R with respect to rounding errors. The desired derivatives are obtained by automatic differentiation on the computational graph of the given numerical algorithm(s).

5 The optimization algorithms

In order to achieve global error bounds Miller's algorithm applies a numerical maximizer routine. Since in general the error measuring functions are not differentiable, a direct search method, the method of Rosenbrock [71] is used. Naturally, it is not guaranteed that global maximum is reached, since the maximizer routine may terminate in a local extremum, or it is also possible that it fails to converge at all. So sometimes we may get misleading results: unstable methods may appear to be stable. On the other hand the various error measuring numbers have the adventage, that the relations holding among them usually prevents the user to accept missleading results. Since the publication of Rosenbrock's method, the derivative-free optimization has also developed a lot (see, e.g. [64], [75], [45], [9], [48], [42], [13], [23]). In order to improve the performance of the program I selected, programmed and tested two extra methods. These are the famous Nelder-Mead simplex method ([64], [45], [9], [48], [42], [13], [23]) and Torczon's multidirectional search ([75], [48], [42], [13]). The two selected algorithms are the following:

The Nelder-Mead method

Initialization: Choose an initial simplex of vertices $Y_0 = \{y_0^0, y_0^1, \ldots, y_0^n\}$. Evaluate f at the points in Y_0. Choose constants:

$$0 < y^s < 1, \quad -1 < \delta^{ic} < 0 < \delta^{oc} < \delta^r < \delta^e.$$

For $k = 0, 1, 2, \ldots$

0. Set $Y = Y_k$.
1. **Order:** Order the $n+1$ vertices of $Y = \{y^0, y^1, \ldots, y^n\}$ so that

$$f^0 = f\left(y^0\right) \leq f^1 = f\left(y^1\right) \leq \cdots \leq f^n = f\left(y^n\right).$$

2. **Reflect:** Reflect the worst vertex y^n over the centroid $y^c = \sum_{i=0}^{n-1} y^i/n$ of the remaining n vertices:

$$y^r = y^c + \delta^r \left(y^c - y^n\right).$$

   Evaluate $f^r = f(y^r)$. If $f^0 \leq f^r < f^{n-1}$, then replace y^n by the reflected point y^r and terminate the iteration: $Y_{k+1} = \{y^0, y^1, ..., y^{n-1}, y^r\}$.
3. **Expand:** If $f^r < f^0$, then calculate the expansion point

$$y^e = y^c + \delta^e \left(y^c - y^n\right)$$

and evaluate $f^e = f(y^e)$. If $f^e \leq f^r$, replace y^n by the expansion point y^e and terminate the iteration: $Y_{k+1} = \{y^0, y^1, ..., y^{n-1}, y^e\}$. Otherwise, replace y^n by the reflected point y^r and terminate the iteration:

$$Y_{k+1} = \{y^0, y^1, ..., y^{n-1}, y^r\}.$$

4. **Contract:** If $f^r \geq f^{n-1}$, then a contraction is performed between the best of y^r and y^n.
   (a) **Outside contraction:** If $f^r < f^n$, perform an outside contraction

$$y^{oc} = y^c + \delta^{oc}(y^c - y^n)$$

   and evaluate $f^{oc} = f(y^{oc})$. If $f^{oc} \leq f^r$, then replace y^n by the outside contraction point y_k^{oc} and terminate the iteration:

$$Y_{k+1} = \{y^0, y^1, ..., y^{n-1}, y^{oc}\}.$$

   Otherwise, perform a shrink.
   (b) **Inside contraction:** If $f^r \geq f^n$, perform an inside contraction

$$y^{ic} = y^c + \delta^{ic}(y^c - y^n)$$

   and evaluate $f^{ic} = f(y^{ic})$. If $f^{ic} < f^n$, then replace y^n by the inside contraction point y^{ic} and terminate the iteration:

$$Y_{k+1} = \{y^0, y^1, ..., y^{n-1}, y^{ic}\}.$$

   Otherwise, perform a shrink.
5. **Shrink:** Evaluate f at the n points $y^0 + \gamma^s(y^i - y^0)$, $i = 1, \ldots, n$, and replace $y^1, ..., y^n$ by these points, terminating the iteration:

$$Y_{k+1} = \{y^0 + \gamma^s(y^i - y^0), i = 0, ..., n\}.$$

The MDS method

Initialization: Choose an initial simplex of vertices $Y_0 = \{y_0^0, y_0^1, \ldots, y_0^n\}$. Evaluate f at the points in Y_0. Choose constants:

$$0 < \gamma^s < 1 < \gamma^e.$$

For $k = 0, 1, 2, ...$

0. Set $Y = Y_k$.
1. **Find best vertex:** Order the $n + 1$ vertices of $Y = \{y^0, y^1, ..., y^n\}$ so that $f^0 = f(y^0) \leq f(y^i)$, $i = 1, ..., n$.

2. **Rotate:** Rotate the simplex around the best vertex y^0:

$$y_i^r = y^0 - (y^i - y^0), \quad i = 1, ..., n.$$

Evaluate $f(y_i^r)$, $i = 1, ..., n$, and set $f^r = \min\{f(y_i^r) : i = 1, ..., n\}$. If $f^r < f^0$, then attempt an expansion (and then take the best of the rotated or expanded simplices). Otherwise, contract the simplex.

3. **Expand:** Expand the rotated simplex:

$$y_i^e = y^0 - \gamma^e(y^i - y^0), \quad i = 1, ..., n.$$

Evaluate $f(y_i^e)$, $i = 1, ..., n$, and set $f^e = \min\{f(y_i^e) : i = 1, ..., n\}$. If $f^e < f^r$, then accept the expanded simplex and terminate the iteration: $Y_{k+1} = \{y^0, y_1^e, ..., y_n^e\}$. Otherwise, accept the rotated simplex and terminate the iteration: $Y_{k+1} = \{y^0, y_1^r, ..., y_n^r\}$.

4. **Shrink:** Evaluate f at the n points $y^0 + \gamma^s(y^i - y^0)$, $i = 1, ..., n$, and replace $y^1, ..., y^n$ by these points, terminating the iteration:

$$Y_{k+1} = \{y^0 + \gamma^s(y^i - y^0), i = 0, ..., n\}.$$

Details on the convergence and implementation may be found in the literature [71], [64], [75], [45], [9], [48], [42], [13], [23].

5.1 Comparative testing of the applied direct search methods

In the following, I give some test results about the use of the applied direct search methods. The tests concern three algorithms, also discussed by Miller in his case studies [58]: inversion of triangular matrices using back substitution, Gaussian elimination without pivoting and Gauss-Jordan elimination with partial pivoting. We consider such error functions in each test case that are known to be unbounded (see Miller [58]), and we investigate the number of function evaluations required by the direct search methods to reach the target value of 10000. The size parameter for the matrix to invert and the system of equitions to solve by the Gauss and Gauss-Jordan elimination methods will be set to $n = 4$, $n = 8$ and $n = 16$ in each case.

5.1.1 Triangular matrix inversion

Table 1 shows the results for the analysis of inverting triangular matrices using back substitution. We started the maximizer routines from the diagonal matrix

$$D = \begin{bmatrix} 1 & & & & \\ & 2 & & & \\ & & 3 & & \\ & & & \ddots & \\ & & & & n \end{bmatrix}.$$

In this case, the Multidirectional search method was the only method that could

n	Rosenbrock	Nelder-Mead	Multidirectional
4	–	–	232
8	–	–	184
16	–	–	680

Table 1: Number of function evaluations required to maximize the normwise backward error for triangular matrix inversion.

find a value greater than 10000 for the normwise backward error (JW_L).

5.1.2 Gaussian elimination

The results for Gaussian elimination are shown in Table 2. The error function to maximize was the normwise mixed forward-backward error (WK_L). The maximization was started from the system:

$$A = \begin{bmatrix} 3 & 1 & 1 & \cdots & 1 \\ 1 & 4 & 1 & \cdots & 1 \\ 1 & 1 & 5 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & 1 & \cdots & n+2 \end{bmatrix}, \qquad b = \begin{bmatrix} n+2 \\ n+3 \\ n+4 \\ \vdots \\ 2n+1 \end{bmatrix}. \tag{6}$$

Table 3 shows the results of a test that was the same as the previous except we constrained the maximization to symmetric matrices. The search was started fom (6).

5.1.3 Gauss-Jordan elimination with partial pivoting

Finally, I have tested the search methods on the normwise backward error (JW_L) for Gauss-Jordan elimination with partial pivoting. The results are shown in Table 4. The Nelder-Mead simplex method failed for all sizes.

n	Rosenbrock	Nelder-Mead	Multidirectional
4	209	–	555
8	692	461	1397
16	1139	2508	9828

Table 2: Number of function evaluations required to maximize the normwise mixed forward-backward error for Gaussian elimination.

n	Rosenbrock	Nelder-Mead	Multidirectional
4	–	162	324
8	226	–	1631
16	1843	759	3195

Table 3: Number of function evaluations required to maximize the normwise mixed forward-backward error for Gaussian elimination. The search was constrained to symmetric matrices.

From the above examples one could deduce that the less robust method is the Nelder-Mead simplex method. As the multidirectional search method found the target value in all cases it is our strongest method, but it requires much more function evaluations and so computational time, than the other two methods. A good practice could be if we start an analysis using the Rosenbrock method, and if it fails we can exploit multidirectional search.

6 The new operator overloading based approach of 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 Rall [67] or Griewank [37].

The original 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

n	Rosenbrock	Nelder-Mead	Multidirectional
4	–	–	1248
8	475	–	8052
16	5822	–	17958

Table 4: Number of function evaluations required to maximize the normwise backward error for Gauss-Jordan elimination.

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 fixed 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 original 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 [12], Verma [78] and MAD (by Shaun A. Forth [22]). 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.

7 The Miller Analyzer for Matlab

Miller Analyzer for Matlab is a mixed-language software. We kept several routines from the work of Miller et al. [57], 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 [68]), 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 then the original compiler (the minicompiler of Miller) 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 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

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)

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.

7.1 Defining the numerical method to analyse by m-file programming

The numerical method to analyse must be implemented in a special way in the form of m-functions. The numerical algorithm can be given either as a single m-function, or it can be organized into a main m-function and one or more subfunctions. The purpose of these m-files is to build the computational graph corresponding to the floating point operations performed when the numerical algorithm is executed upon a given input data. Instead of the built-in double precision MATLAB array, we use a special class called cfloating, for which the arithmetic operators and the function

sqrt for square root are defined (overloaded). When the error analyser calls the main m-function, the MATLAB interpreter executes it. Upon performing the operations on the variables of type cfloating, beside computing the floating point result of the operation, the appropriate node is also added to the computational graph. The cfloating class contains two fields (data members): the actual floating point value, as in the case of ordinary variables, and a node identifier, which identifies the node in the graph corresponding to the given floating point value.

As every MATLAB variable, cfloating is also an array, and every element of the array contain the two fields: value and node identifier. It can be a matrix (two dimensional array) or a multidimensional array (array with more than two dimension). Scalars (1-by-1 array) and vectors (1-by-n or n-by-1 array) are special matrices in MATLAB. The MATLAB operators: $+, -, *, /, .*, ./, .\backslash$ and the function sqrt can be applied, scalar, vector, and matrix operations are also supported. The cfloating type substitutes the real, double precision MATLAB type, the complex arithmetic is not supported directly. By algorithms involving complex computation, the user must decompose the complex operations to real arithmetic by hand. We are planning to add direct support of complex arithmetic in the future.

7.1.1 The main m-function

Algorithm 3 Main function

```
1:  function main( identifier )
2:      identifier=miller_ptr(identifier);

3:      Initializing the input as cfloating arrays
        and adding the input nodes to the graph

4:      Run the algorithm using cfloating type
        to add the arithmetic nodes
        and compute the output as cfloating array

5:      Add the output nodes to the graph
    end
```

The main m-function is the m-function that is dedicated to be called by the error analyser, when the computational graph has to be built. Algorithm 3 shows the general form of the main m-function. As in line 1, the main m-function must have one input argument and it must not have any output arguments. To make

MATLAB able to find our main m-function, it has to be reside in an m-file with the same name, and the m-file must be on the MATLAB path or in the working directory. Line 2 fulfills a formal requirement, all such m-functions have to be begun with that statement. Actually it creates a handle to the computational graph being built and we will use it for several purposes (instead of `'identifier'` any valid variable name can be used).

In our model the analysed numerical method computes a function $P(d)$ ($P : \mathbb{R}^n \to \mathbb{R}^k$, $d \in \mathbb{R}^n$ is the input vector) The length n of the input vector d is always fixed for analysis, error maximization is performed in the n dimensional space $\mathbb{R}^n$.

In many cases P is not defined at every point in $\mathbb{R}^n$, since no division by zero may occur and no square root of a negative number may be taken. If the MATLAB interpreter encounters such an operation, it signals the error condition by throwing an exception. The maximizer catches the error, so the maximization process is not terminated, but continues at other data d.

We say that 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 all unrollable taxative loops. In the case of such programs a unique computational graph represents the algorithm (assuming that the number of inputs is fixed), so it is enough to call the main m-function and build the computational graph only once[1]. On the other hand, if the flow of control depends on the input values, we regenerate the graph by calling the main function at every d input data, upon which the error measuring number is to be computed. In such cases, the number of arithmetic operations may also depend on d.

In a computational graph, there can be four kinds of node. First we add the *input nodes* that correspond to the n entries of the input vector d (see line 3). In the next step (line 4) we run the algorithm on d. Beside evaluating the m operations, we also add m *arithmetic nodes* to the graph. We distinguish six kinds of arithmetic nodes: four correspond to the binary operations (+, -, *, /) and two to square root and unary minus. A constant value may also appear as operand in an operation. In the graph, *constant nodes* corresponds to the constant values used in the algorithm. Finally, some of the arithmetic nodes are designated as *output nodes* meaning that the result of the given operation is one of the output values of the algorithm (line 5). In order to evaluate the error measuring number at d, the partial derivatives of the values corresponding to the output nodes with respect to the values corresponding to the input nodes and the relative rounding errors hitting the arithmetic nodes will be computed.

[1] In some cases we run the algorithm at the first time in order to count the operations and determine the amount of memory to allocate, and make an additional call to build the graph.

7.1.2 The input nodes

Assume that the main m-function is called, and the MATLAB interpreter is about to execute line 3. At that point, the number n of inputs is fixed and the actual floating point values of the inputs, the entries of d are set. Our task is to create and initialize input variables of type cfloating with the values of d, and add the corresponding n input nodes to the computational graph. Three routines will help us: input_size, input, parameter.

input_size Syntax:

n = input_size(miller)

The function returns the number of inputs into n. Here and in the following, the parameter miller is the same variable as in algorithm 3 line 2.

input Syntax:

B = input(miller)
B = input(miller,n)
B = input(miller,m,n)
B = input(miller,[m n])
B = input(miller,m,n,p,...)
B = input(miller,[m n p ...])

Using the variants of function input, we can create variables of cfloating type.

The input vector d can be read as a sequential file. At the beginning a pointer points to the first entry, and after reading the current entry, the pointer is incremented to point the next element of d. Unless we read exactly n elements applying one or more times the input statement during the execution of the main m-function and its subfunctions, we get an error message. Another restriction is that we cannot read an entry more than once, since there is not 'rewind' or 'seek' routines.

1. B = input(miller)
   reads one floating point value from d, adds an input node to the computational graph, and returns a scalar of type cfloating initialized with the value and the identifier of the node currently added.

2. B = input(miller,n)
   reads n^2 elements from the input vector, adds the corresponding input nodes, and returns an n-by-n cfloating matrix initialized with the value - node identifier pairs in column major order. It has the same effect as:

```
for j = 1:n
    for i = 1:n
        B(i,j) = input(miller);
    end
end
```

3. B = input(miller,m,n) or B = input(miller,[m n])
   returns an m-by-n cfloating matrix with elements initialized just as above, but with $m \cdot n$ elements instead of n^2:

```
for j = 1:n
    for i = 1:m
        B(i,j) = input(miller);
    end
end
```

4. B = input(miller,m,n,p,...) or B = input(miller,[m n p...])
   returns an m-by-n-by-p-by-... cfloating array initialized with $m \cdot n \cdot p \cdot \ldots$ currently read and added value - node identifier pairs in column major order.

parameter In most cases all the size parameters of a numerical algorithm cannot be deduced from the number of inputs. For example, assume that we would like to analyse an algorithm for least square solution of an overdetermined system $\|Ax - b\| \to \min$. In that case the function input_size will return the number of the elements in the extended matrix $\begin{bmatrix} A & b \end{bmatrix}$, but it does not determine uniquely the number of equations and unknowns. For such cases the user can pass a vector of parameters to the main m-function while calling the error analyser routines. The parameter vector is a double precision array. It is the users decision what parameters to use by implementing the input algorithm, and how to arrange it into a single vector, or to use parameters at all.
Syntax:

```
b = parameter(miller,i)
```

returns the i-th entry of the parameter vector served by the user.

Using global variables is another approach to passing parameters, and would also work fine. However, using the parameter statement is safer than global variables, because the software monitors the parameter vector for changes. We have mentioned that by straight-line programs the computational graph is generated only once. If the parameter vector changes, it is guaranteed that the main m-function will be called and the graph will be regenerated. If parameters are passed by global variables, it is up to the user to ensure that the graph is regenerated.

7.1.3 The arithmetic nodes

The desired arithmetic nodes can be added to the computational graph by executing arithmetic operations on cfloating arrays. MATLAB has two different types of arithmetic operations. Matrix arithmetic operations are defined by the rules of linear algebra. Array arithmetic operations are carried out element by element, and can be used also with multidimensional arrays. The period character (.) distinguishes the array operations from the matrix operations. The cfloating array supports both types with the restriction, that matrix division, elementwise power and matrix power are not allowed. Note that the cfloating array supports only real arithmetic, it does not store imaginary part and cannot perform complex operations. The following operators can be used with cfloating arrays:

1. Addition or unary plus[2]. $A+B$ adds A and B. A and B must have the same size, unless one is a scalar. A scalar can be added to a matrix of any size.

2. Subtraction or unary minus[3]. $A-B$ subtracts B from A. A and B must have the same size, unless one is a scalar. A scalar can be subtracted from a matrix of any size.

3. Matrix multiplication. $C = A * B$ is the linear algebraic product of the matrices A and B. For nonscalar A and B, the number of columns of A must equal the number of rows of B. A scalar can multiply a matrix of any size. If both A and B are matrices $C = A * B$ has the same effect as:

```
[n,k] = size( A );
[l,m] = size( B );
assert( k == l, 'Inner dimensions must agree!' );
for i = 1 : n
    for j = 1 : m
        C(i,j) = 0.0;
        for k = 1 : l
            C(i,j) = C(i,j) + A(i,k) * B(k,j);
        end
    end
end
```

4. Array multiplication. $A .* B$ is the element-by-element product of the arrays A and B. A and B must have the same size, unless one of them is a scalar.

[2]Unary plus does not add a node to the graph, just returns the same cfloating array.
[3]Unary minus always considered to be error free.

5. Division by a scalar. B/a is the matrix with elements $B(i,j)/a$ (a is a scalar). Note that this definition differs from the built-in version, where B/A is the matrix division $B * \mathrm{inv}(A)$.

6. Array right division. $A./B$ is the matrix with elements $A(i,j)/B(i,j)$. A and B must have the same size, unless one of them is a scalar.

7. Array left division. $A.\backslash B$ is the matrix with elements $B(i,j)/A(i,j)$. A and B must have the same size, unless one of them is a scalar.

8. Square root. $\mathrm{sqrt}(A)$ is the element-by-element square root of the array A.

The above cfloating operations computes the floating point values of the entries of the resulting arrays and adds the arithmetic nodes corresponding to the elementary operations evaluated. The results will be cfloating arrays with elements equal to the resulting value - node identifier pairs.

Combining cfloating with built-in data type *double* The cfloating version of the above operators will be called, if at least one of the operands is of type cfloating. In the mixed cases, when one operand is a cfloating array and the other is a double precision MATLAB array[4], the entries of the built-in typed array are considered to be constants. The operation is only executed after the corresponding constant nodes have been added. For a particular value a constant node is added only once for the whole execution. For example if B is a cfloating matrix, then $C = \mathrm{zeros}(\mathrm{size}(B)) + B$ will add only one constant node for the value 0.0, and every entry will refer to that node. Furthermore all additional occurrences of the constant value zero, will refer to that previously added node.

The function cfloating() Sometimes it is necessary to explicitly convert a built-in double precision array to cfloating type.
Syntax:

$$C = \mathrm{cfloating}(B)$$

returns a cfloating array, which have the same size as B. The value part of the entries of C will be initialized with the corresponding elements of B, but no nodes will be added to the graph, and the node identifier part will be set to zero. The addition of the constant node corresponding to an entry of C will be postponed until the first occurrence of the particular entry in an arithmetic operation. In the following we shall refer to a value without corresponding graph node as an unregistered value. If B is already a cfloating array, the function has no effect.

[4] cfloating can be combined only with double in binary operations

Programming with cfloating MATLAB is a matrix-based computing environment with sophisticated matrix and array manipulation methods. The following functions works with arrays of any type without explicitly defining (overloading) them, so these methods have the same behavior in conjunction with cfloating as with built-in types. For detailed description see the MATLAB help.

1. **Matrix concatenation functions.** cat, horzcat, vertcat, repmat, blkdiag. Function horzcat(A, B, C,...) is a synonym for [A, B, C,...], and vertcat(A,B,C,..) for [A; B; C;...]. In the case of cat, horzcat, vertcat and blkdiag combination of cfloating and built-in types is also allowed. If at least one of the arguments is a cfloating array, the arrays of built-in type will be converted to cfloating arrays with unregistered values, and then concatenated. So $D = [A, B, C]$ has the same effect as $D = [\text{cfloating}\,(A)\,, \text{cfloating}\,(B)\,, \text{cfloating}\,(C)]$.

2. **Matrix indexing.** The various indexing schemes of MATLAB can also be applied to cfloating matrices and multidimensional arrays on both sides of the assignment operator, but the cfloating array itself must not be used as an index. A submatrix resulted from a cfloating array by indexing will be also of type cfloating.

3. **Getting information about a matrix.** The functions: length, ndims, numel, size, isempty, isscalar, isvector can be used in the same way as for built-in types.

4. **Reshaping a matrix.** The functions: reshape, rot90, fliplr, flipud, flipdim, transpose, permute, ipermute, circshift, shiftdim can also be used. (transpose(A) is the same as A').

A good m-file programming practice is preallocating arrays before loops to avoid their growing inside the loop. Preallocating leads typically to a situation, where explicit conversion to cfloating is necessary. Consider the case of matrix multiplication. Assume that algorithm 4 called with cfloating matrices. If we omit line 6, then an error occurs at line 10. Without line 6 the right-hand side: C(i,j) + A(i,k) * B(k,j) at line 10 results a cfloating scalar, but C is still a double matrix. By such indexed assignment with different types, as in line 10, MATLAB tries to convert the right-hand side value to the type of the left-hand side. However, automatic conversion from cfloating to double is not allowed, which yields an error.

Algorithms that are not straight-line programs In this section we will see, how we can make the flow of control depend on the value of cfloating arrays.

1. **The function value.**
   Syntax:

Algorithm 4 Matrix multiplication

```
 1:  function C = mtimes(A,B)
 2:  [n,k] = size( A );
 3:  [l,m] = size( B );
 4:  assert( k == l, 'Inner dimensions must agree!' );
 5:  C = zeros(n,m);
 6:  C = cfloating(C);
 7:  for i = 1 : n
 8:      for j = 1 : m
 9:          for k = 1 : l
10:              C(i,j) = C(i,j) + A(i,k) * B(k,j);
11:          end
12:      end
13:  end
```

$$C = \text{value}\,(B)$$

If B is a cfloating array, it returns with the value part of B. C will be a built-in typed double array with the same size as B. We have mentioned, that 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 value of cfloating variables, based on them, we can construct conditional expressions for if tests and while loops.

2. **Relational operators.** For convenience we have defined the relational operators (<, >, <=, >=, ==, ~=) on cfloating arrays. Hence, cfloating arrays can directly (without the value function) be operands of relational expressions. For instance $a > 0.0$ has the same effect as $\text{value}\,(a) > 0.0$.

If either a value function or a relational operator in conjunction with a cfloating value occurs in the m-file implementation of the input algorithm, the main m-function will be called and the computational graph will be regenerated at every set of input data, upon which the error measuring number is to be computed.

Constrained error maximization Actually the stability analysis by Miller Analyser for MATLAB is the maximization of an error function $\omega\,(d)$. Beside unconstrained optimization, we can also perform constrained optimization. The function constraint is used to define constraints for the search for large values of the error measuring quantity.
Syntax:

constraint(miller,V)

Here, V can be either a cfloating or a double array. If V is cfloating typed, then it is first converted to built-in type double by calling value(V). The entries of V is then added to the vector of constraints $C(d)$. The i-th element of $C(d)$ represents the constraint $C_i(d) \geq 0$. The constrained optimization is realized through penalizing the value of $\omega(d)$ at inputs for which any $C_i(d)$ is near to or less than zero. The error measuring value is simply multiplied by $\min(1, C_1, C_2, \ldots, C_n)$ $(n \geq 0)$, and the maximization is performed on that penalized error measuring value.

The weak composition model Syntax:

composition(miller)

instructs the error analyser to apply the weak composition model of error propagation. Suppose F is a program for computing $f(d)$. Calling function composition separates F into two subprograms H and G: H consists of the arithmetic operations performed before the invoking of composition and G consists of the later operations. The weak composition model assumes that the operations are exact but intermediate values are rounded as they passed from H to G. At most one composition statement can be executed.

Error handling By executing the m-files implementing the input algorithm many kinds of error condition may arise. We distinguish terminating and non-terminating errors. If a terminating error occurs, the process of error maximization is aborted and control returns to the MATLAB prompt with an error message. When we execute the input algorithm upon the initial input vector, all the errors are terminating errors.

In the case of non-straight-line programs the main m-function is called at every set of data, upon which the error measuring quantity is to be evaluated. By these further executions a non-terminating error may also occur. A non-terminating error aborts only the execution of the input algorithm, but maximization is continued by evaluating the error measuring quantity upon other input vectors. If it is not the initial execution, division by a non-constant cfloating variable with value zero, or taking the square root of a negative number (non-constant, cfloating) causes a non-terminating error. The user can also trigger a non-terminating error by calling alg_error:
Syntax:

alg_error(miller,message)

The function also prints an error message to the MATLAB prompt. All other error conditions terminate the maximization of roundoff errors.

7.1.4 The output nodes

The final step of building a computational graph is choosing the output nodes of the algorithm from its arithmetic nodes. The values corresponding to the output nodes are those, whose numerical stability will be analysed.
Syntax:

output(B)

The node identifiers corresponding to the elements of the cfloating array B are added to the vector of outputs. Every element of B must be a computed value, so only arithmetic node may become an output node. Several output statements may be executed, but a single node must not be added more than once. The vector of outputs is written by the output function, as a sequential file: A pointer is maintained to designate the index where the next element is to be put. If B has n entries, the pointer is incremented by n.

7.2 Doing the analyses

In this section we will see how we can analyse the numerical stability of algorithms defined according to the rules given in the previous section.

7.2.1 Creating a handle to the error analyser

To perform error analysis, an object of class *miller* have to be created. By creation we must provide the name of the main m-function of the input algorithm. For comparing numerical stability of two algorithms solving the same problem the user must provide the names of the two main m-files of the algorithms to be compared.
Syntax:

m = miller(mfunction)

Analysing a single algorithm m-function is a string containing the name of the function defining the method to analyse. By comparing, mfunction contains two names delimited by '/'. Note that the names have to be given without the '.m' extension. In the following m will denote a properly created object of type *miller*.

7.2.2 Setting the parameters of maximization

By the set method parameters can be set, which determine how the analysis will be performed.

1. We can analyse stability according to several error measuring quantities.
   Syntax:

m = set(m, 'error_measure',errorstr)

Sets the error measuring quantity to maximize. `errorstr` is a string containing the name of the desired error measuring quantity. Analysing a single algorithm the values of `errorstr` can be: 'wkl', 'wke', 'jwl', 'jwe', 'erl', 'ere' for the appropriate error comparing value. To compute condition number `errorstr` is set to 'cnl' or 'cne' for normwise and elementwise condition number. Assume that we are comparing two methods and '`method1/method2`' was the m_function argument by constructing of the miller object. The value 'jw1/2' will set the error comparing value to $\mathrm{JW}_{\texttt{method1/method2}}$. Similarly 'jw2/1', 'er1/2', 'er2/1' will set $\mathrm{JW}_{\texttt{method2/method1}}$, $\mathrm{ER}_{\texttt{method1/method2}}$ and $\mathrm{ER}_{\texttt{method2/method1}}$ respectively. For details about the error measuring numbers see pages 89-94 in the book by Miller and Wrathall [58].

2. The stopping criterion of maximization can also be set.
Syntax:

m = set(m, 'stop_crit',v)

Sets the stopping criterion for the given value v. v must be a scalar. The maximization terminates, if this value is reached. Zero turns off testing on reaching a stopping value.

7.2.3 Error analysis

1. For testing purposes we can omit computing error measuring quantities and just run the input algorithm.
Syntax:

output = run(m,d)
output = run(m,d,p)

Returns the output vector of the input algorithm. d is a double precision MATLAB array with the input data. If d is a matrix it will be vectorized in column major order. The entries of the vector d will be read by the input statement (see section 7.1.2). p is the parameter vector. Its entries can be reached in the input algorithm by the parameter statement as in section 7.1.2 described. If p has more than one dimensions, it is also vectorized in column major order.

2. Computing error measuring numbers at a given set of data d:
Syntax:

rho = <errormeasure>(m,d)
rho = <errormeasure>(m,d,p)

d and p are the same as above. <`errormeasure`> can be substituted with: wkl, wke, jwl, jwe, erl, ere, cnl, cne, jw1vs2, jw2vs1, er1vs2, er2vs1 for the desired error measuring number. For example jw1vs2 will compute $\mathrm{JW}_{\texttt{method1/method2}}$. The calling also sets the 'error_measure' argument to the error measuring quantity being computed. So calling maxsearch after one of these function will maximize the error value has just been computed.

3. For performing maximization the function maxsearch have to be used:
Syntax:

[rho, dfinal] = maxsearch (m, dinit, methodcode)
[rho, dfinal] = maxsearch (m, dinit, methodcode, p)

`dinit` is the input data vector from which the maximization starts, `methodcode` is a string: 'ros', 'nms', 'mds' to perform optimization using the Rosenbrock, the Nelder-Mead simplex, or the Multidirectional Search by Torczon respectively. p is the parameter array as above.

Error handling We have mentioned in section 7.1.3, that terminating and nonterminating errors may occur during execution of the input algorithm. Further nonterminating errors may arise during the computation of the error measuring quantity. The nonterminating errors do not abort the error maximization process. The evaluation of the error measuring quantity fails at the given set of data, but maximization continues. The nonterminating errors are counted, and at the end of maximization we can get a report about the errors encountered. As in section 7.1.3 described: by the first evaluation of the error measuring value all errors are terminating errors. So, the computation must be error-free at `dinit` to perform maximization. The nonterminating errors are the following:

1. Division by zero or taking the square root of negative number during the execution of the input algorithm.

2. By computing wkl, wke, jwl, jwe, jw1vs2 or jw2vs1 we may get the error message 'OMEGA failure'. This arise, if either the number of operations that are not error free, or the number of inputs is less than the number of outputs.

3. By the same error measures as above 'DIAGON failure' arises if the error measuring number cannot be computed accurately because of rank deficiency.

4. By computing erl, ere, er1vs2, er2vs1 we can get 'GETER failure'. These error measuring quantitie are the quotient of the norms of two matrices. The error is encountered, if the divisor is zero.

5. If computing the condition number fails we get 'CONDIT failure'.

Functions reset and resetcounter The miller object uses dynamic memory allocation: it grows for the needs, but automatically it does not free up memory. If we would like to free up the memory owned by the object, reset must be called.
Syntax:

reset(m)

Frees up the memory allocated by m. The object will be the same state as if it were created right now.

By maxsearch beside the errors the evaluations of the error measure is also counted. The counters can be reset calling resetcounter.
Syntax:

resetcounter(m)

Before calling maxsearch again, reset or resetcounter need to be called.
Syntax:

destroy(m)

Frees up all the memory allocated by m. Referencing to m after calling `destroy` causes segmentation violation!

The display function The display function is also defined for the miller class. This function called for built in MATLAB types, if their values are printed when the semicolon omitted. Display returns several information on the actual object.

8 Applications

8.1 Gaussian elimination, an important example

The numerical stability of the Gauss method and its variants is in question, since von Neumann, Goldstine, Turing, Fox, Huskey and Wilkinson advocated it as an efficient sequential computer solver for linear algebraic systems of the form $Ax = b$. Due to mainly Wilkinson [79], [80], [39] we know a lot about the numerical stability

of the Gaussian elimination method. Hence it is a good and quite significant algorithm to test the efficiency of the Miller method.

Using the original version of the method [57], 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 [58]). 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 [34].

We consider first the procedure without pivoting. Algorithm 5 shows an m-

Algorithm 5 Gaussian elimination

```
 1:  function b = gauss( A, b )
 2:  % Gaussian elimination
 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:      for i = k + 1 : n
11:          amult = A(i,k) / A(k,k);
12:          A(i,k+1:n) = A(i,k+1:n) - amult * A(k,k+1:n);
13:          b(i) = b(i) - amult * b(k);
14:      end
15:  end
16:  %
17:  % Back substitution
18:  for i = n : -1 : 1
19:      b(i) = ( b(i) - A(i,i+1:n) * b(i+1:n) ) / A(i,i);
20:  end
```

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: $\mathrm{cond}(A) \approx 5.6179$, so Gaussian elimination without pivoting can be unstable at very well-conditioned data.

Algorithm 6 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
```

Consider Algorithm 6 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 to work with variables of `cfloating` type. If B is a `cfloating` array, $C = \text{value}\,(B)$ 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 6 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.

8.2 The analysis of the implicit LU and Huang methods

Using the software package we have analyzed the numerical stability of solving the $Ax = b$ linear system ($A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$) with the implicit LU and Huang methods. These methods are special variants of the ABS methods [1], [2]. The linear ABS methods are projection methods of the form:

Algorithm 7 Linear ABS method

$x_1 \in \mathbb{R}^n$, $H_1 = I$, $V = [v_1, ..., v_n] \in \mathbb{R}^{n\times n}$,
$W = [w_1, ..., w_n] \in \mathbb{R}^{n\times n}$, $Z = [z_1, ..., z_n] \in \mathbb{R}^{n\times n}$
for $k = 1, ..., n$
$\quad p_k = H_k^T z_k \qquad (z_k \in \mathbb{R}^n,\ p_k^T A^T v_k \neq 0)$
$\quad x_{k+1} = x_k - \frac{p_k v_k^T (Ax_k - b)}{v_k^T A p_k}$
$\quad H_{k+1} = H_k - \frac{H_k A^T v_k w_k^T H_k}{w_k^T H_k A^T v_k} \qquad (w_k \in \mathbb{R}^n,\ w_k^T H_k A^T v_k \neq 0)$
end
$x_{n+1} = A^{-1} b$

8.2.1 The implicit LU method

The implicit LU algorithm is given by the choice $W = Z = I$. We consider the special case, when $V = I$ and $x_1 = 0$. Algorithm 8 shows the method to analyze. Let the initial data for maximizing be:

$$A_0 = \begin{bmatrix} 3 & 1 & 1 & 1 \\ 1 & 4 & 1 & 1 \\ 1 & 1 & 5 & 1 \\ 1 & 1 & 1 & 6 \end{bmatrix}, \qquad b_0 = \begin{bmatrix} 6 \\ 7 \\ 8 \\ 9 \end{bmatrix}.$$

Note that in the case of Algorithm 8 the maximization is implicitly constrained to strongly non-singular systems. We have chosen 10000 as target value. The

Algorithm 8 Implicit LU method

$x_1 = 0,\ H_1 = I$
for $k = 1, ..., n$
$\quad x_{k+1} = x_k - \frac{H_k^T e_k e_k^T (Ax_k - b)}{e_k^T A H_k^T e_k}$
$\quad H_{k+1} = H_k - \frac{H_k A^T e_k e_k^T H_k}{e_k^T H_k A^T e_k}$
end

maximizer finds values greater than the target value for both WK_L and ER_L error-measuring numbers. Because of the inequalities hold between the error-measuring numbers, a set of data exists for all error-measuring numbers, where their values exceed 10000. In addition at the final set of data the condition number is extremly low. In the case of WK_L it is 12.3, and in the case of ER_L it is 8.1. So the implicit LU algorithm is unstable even at very well conditioned data.

Compared the algorithm with Gaussian elimination without pivoting the results are: $JW_{LU/Gauss} > 10000$, $JW_{Gauss/LU} > 10000$, $ER_{LU/Gauss} > 10000$, $ER_{Gauss/LU} < 29$. So the implicit LU has worse stability properties, than Gaussian elimination without pivoting.

8.2.2 The Huang method

The algorithm is obtained by the choice $w_i = z_i = A^T v_i$ we have analysed four variants of the method. With the given substitutions applied to the general ABS algorithm we get the first variant:

Algorithm 9 Huang method, first variant

$x_1 \in \mathbb{R}^n,\ H_1 = I$
for $k = 1, ..., n$
$\quad p_k = H_k^T A^T e_k$
$\quad x_{k+1} = x_k - \frac{p_k e_k^T (Ax_k - b)}{e_k^T A p_k}$
$\quad H_{k+1} = H_k - \frac{H_k A^T e_k e_k^T A H_k}{e_k^T A H_k A^T e_k}$
end

In the case of the Huang method the maximization is implicitly constrained to non-singular sytems, because otherwise division by zero would occur. We have started the maximizer from the same initial data A_0, b_0 as in the case of the implicit LU method. Applying the maximizer ER_L and WK_L have exceeded 10000, so the method is unstable. Unlike the case of implicit LU large values were found at ill-conditioned data (61710 for WK_L, and 32365 for ER_L). So the algorithm is more sensitive to ill-conditioning than it is reasonable. Comparing the algorithm with

the modified Gram-Schmidt method, we get the results: $JW_{Huang/MGS} > 10000$, $JW_{MGS/Huang} < 3400$, $ER_{Huang/MGS} > 10000$, $ER_{MGS/Huang} < 2050$. So there are both sets of data where the Huang, and sets of data where the MGS method gives much poorer results.

If we consider that $H_i = H_i^T$, then the algorithm can be modified as follows:

Algorithm 10 Huang method, second variant

$x_1 \in \mathbb{R}^n$, $H_1 = I$
for $k = 1, ..., n$
$p_k = H_k^T A^T e_k$
$x_{k+1} = x_k - \frac{p_k e_k^T (Ax_k - b)}{e_k^T A p_k}$
$H_{k+1} = H_k - \frac{p_k p_k^T}{e_k^T A p_k}$
end

In this case we have got very similar result to the previous algorithm. Comparing the two method all the error-measuring numbers are bounded by 6.1. Thus the two variants have the same stability properties.

The following variant is based on the fact that $H_i^2 = H_i$:

Algorithm 11 Huang method, third variant

$x_1 \in \mathbb{R}^n$, $H_1 = I$
for $k = 1, ..., n$
$p_k = H_k^T A^T e_k$
$x_{k+1} = x_k - \frac{p_k e_k^T (Ax_k - b)}{e_k^T A p_k}$
$H_{k+1} = H_k - \frac{p_k p_k^T}{p_k^T p_k}$
end

We have got values for ER_L and WK_L exceeding 10000 also at ill-conditioned sets of data. Comparing the algorithm with the MGS method we get similar results to the previus two cases. If we compare the method with the second variant, we get that all error-measuring numbers are not bounded by 10000 except that $ER_{Huang3/Huang2} < 1.5$. So the third variant is more stable than the previous ones.

The fourth analysed algorithm was the so called modified Huang method. See Algorithm 12. In this case the maximizer cannot find value for JW_L greather than 4.3, so the algorithm is backward stable. Comparing with the MGS method the results are: $JW_{Huang/MGS} < 561$, $JW_{MGS/Huang} > 10000$, $ER_{Huang/MGS} < 20$, $ER_{MGS/Huang} < 4038$. So the modified Huang has better stability properties than the MGS method. If we compare the method with the previous (third) variant, we have that: $ER_{Huang4/Huang3} < 1.23$, $JW_{Huang4/Huang3} < 1.6$, the other two

Algorithm 12 Modified Huang method

$x_1 \in \mathbb{R}^n$, $H_1 = I$
for $k = 1, ..., n$
$p_k = H_k \left(H_k^T A^T e_k \right)$
$x_{k+1} = x_k - \frac{p_k e_k^T (Ax_k - b)}{e_k^T A p_k}$
$H_{k+1} = H_k - \frac{p_k p_k^T}{p_k^T p_k}$
end

numbers are not bounded by 10000. So the method is much more stable, than the previous variants of the Huang method.

8.3 An automatic error analysis of fast matrix multiplication procedures

The multiplication of two $n \times n$ matrices requires $O\left(n^3\right)$ arithmetic operations by the standard Cayley definition. For large n, faster or cheaper matrix multiplication methods are clearly important for the applications since the 1950's. The first such method was constructed by Winograd [81]. He observed that for matrices $A, B \in \mathbb{R}^{n \times n}$ (n is even), the entries of $C = AB$ can be written as

$$c_{ik} = \sum_{j=1}^{n/2} \left(a_{i,2j-1} + b_{2j,k} \right) \left(a_{i,2j} + b_{2j-1,k} \right) - \sum_{j=1}^{n/2} a_{i,2j-1} a_{i,2j} - \sum_{j=1}^{n/2} b_{2j-1,k} y_{2j,k}. \quad (7)$$

The second sum depends only on index i, while the third one depends only on index k. Since the last two sums can be precomputed and then used in the computation of the entries of C, we can save approximately half of the multiplications at the expense of extra additions. The method requires $(1/2)\, n^3 + n^2$ multiplications and $(3/2)\, n^3 + O\left(n^2\right)$ additions. If a multiplication takes much longer time than an addition, the method has a clear advantage.

The next step of developing fast methods is due to Strassen [74], who constructed a recursive matrix multiplication algorithm that requires $O\left(n^{\log_2 7}\right)$ arithmetic operations. For simplicity, assume that $n = 2^\ell$. Partition matrices A and B such that

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad \left(A_{ij}, B_{ij} \in \mathbb{F}^{\frac{n}{2} \times \frac{n}{2}} \right).$$

Strassen' method is based on the observation that

$$C = AB = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

can be computed in the form

$$\begin{array}{ll}
M_1 = (A_{12} - A_{22})(B_{21} + B_{22}), & C_{11} = M_1 + M_2 - M_4 + M_6, \\
M_2 = (A_{11} + A_{22})(B_{11} + B_{22}), & C_{12} = M_4 + M_5, \\
M_3 = (A_{11} - A_{21})(B_{11} + B_{12}), & C_{21} = M_6 + M_7, \\
M_4 = (A_{11} + A_{12}) B_{22}, & C_{22} = M_2 - M_3 + M_5 - M_7. \\
M_5 = A_{11}(B_{12} - B_{22}), & \\
M_6 = A_{22}(B_{21} - B_{11}), & \\
M_7 = (A_{21} + A_{22}) B_{11}, &
\end{array} \tag{8}$$

This arrangement requires 7 multiplications and 18 additions of two $\frac{n}{2} \times \frac{n}{2}$ matrices. Using recursion for the matrix multiplications one finds that the total cost of the method is $O\left(n^{\log_2 7}\right)$ arithmetic operations (see also [15].

Somewhat later Winograd [82] gave the following variant of Strassen's algorithm with the same $O\left(n^{\log_2 7}\right)$ arithmetic cost.

$$\begin{array}{lll}
S_1 = A_{21} + A_{22}, & M_1 = S_2 S_6, & T_1 = M_1 + M_2, \\
S_2 = S_1 - A_{11}, & M_2 = A_{11} B_{11}, & T_2 = T_1 + M_4, \\
S_3 = A_{11} - A_{21}, & M_3 = A_{12} B_{21}, & \\
S_4 = A_{12} - S_2, & M_4 = S_3 S_7, & C_{11} = M_2 + M_3, \\
S_5 = B_{12} - B_{11}, & M_5 = S_1 S_5, & C_{12} = T_1 + M_5 + M_6, \\
S_6 = B_{22} - S_5, & M_6 = S_4 B_{22}, & C_{21} = T_2 - M_7, \\
S_7 = B_{22} - B_{12}, & M_7 = A_{22} S_8, & C_{22} = T_2 + M_5. \\
S_8 = S_6 - B_{21}, & &
\end{array} \tag{9}$$

Winograd's variant requires 7 multiplications and 15 additions of two $\frac{n}{2} \times \frac{n}{2}$ matrices.

These results initiated an intensive research on the development of faster matrix multiplication methods and there are several ones (see, e.g. [66], [18], [40], [41]). Today's fastest method is due to Coppersmith and Winograd [14] and its arithmetic cost is $O(n^{2.376})$. For practical purpose, however only Strassen's method and its Winograd variant are advocated. Most of the related papers are concerned with the complexity and/or implementation issues on computer architectures of different types. As for the implementations, we refer only to the papers by D'Alberto and Nicolau [16], [17]. Only a few papers deal with the numerical stability of these fast matrix multiplication algorithms and it is a common belief that such methods are numerically unstable (see, e.g. Higham [39]). Brent [7], [8] gave the first formal error analysis of Winograd's first method and the method of Strassen. For the latter, Brent [8] proved the following perturbation result.

Theorem 1 *Let $A, B \in \mathbb{R}^{n \times n}$, where $n = 2^\ell$. Suppose that $C = AB$ is computed by Strassen's method and that $n_0 = 2^r$ is the threshold at which conventional*

multiplication is used. The computed product $\widehat{C}$ *satisfies*

$$\left\|C-\widehat{C}\right\| \leq \left[\left(\frac{n}{n_0}\right)^{\log_2 12}\left(n_0^2+5n_0\right)-5n\right] u\,\|A\|\,\|B\|+O\left(u^2\right). \tag{10}$$

Here u is the unit roundoff, that is $u=\frac{1}{2}\beta^{1-t}$, where β is the base or radix and t is the precision of the floating point number system (see, e.g. Wilkinson [80], Higham [39] or Muller et al. [62]).

Higham [38] obtained a similar result and the above formulation of Brent's original result is given by him. The numerical stability of Strassen's method was also investigated by Dumitrescu [20]. The numerical stability of the Winograd-Strassen algorithm was investigated by Higham [39].

Theorem 2 *Let* $A, B \in \mathbb{R}^{n\times n}$*, where* $n=2^{\ell}$*. Suppose that* $C=AB$ *is computed by the Winograd-Strassen method (9) and that* $n_0=2^r$ *is the threshold at which conventional multiplication is used. The computed product* $\widehat{C}$ *satisfies*

$$\left\|C-\widehat{C}\right\| \leq \left[\left(\frac{n}{n_0}\right)^{\log_2 18}\left(n_0^2+6n_0\right)-6n\right] u\,\|A\|\,\|B\|+O\left(u^2\right). \tag{11}$$

The Winograd-Strassen algorithm is built in IBM's ESSL software package (see, e.g. [19]) and there is also a United States Patent No. 7209939 for the precision improvement of the algorithm.

For the standard computation of $C=AB$ $(A, B \in \mathbb{R}^{n\times n})$, we have the error bound

$$\left|C-\widehat{C}\right| \leq \gamma_n u\,|A|\,|B|+O\left(u^2\right), \tag{12}$$

where $\gamma_n = n$ and $|A|=\left[|a_{ij}|\right]_{i,j=1}^{n}$ (see, e.g. [80], [36], [39]). Miller [50], [51] showed that if a matrix multiplication algorithm satisfies an error bound of the type (12) and uses only scalar addition, subtraction, and multiplication, then it must perform at least n^3 multiplications. Consequently, algorithms (8) and (9) cannot satisfy the bound (12) that is considered optimal in a sense (see, e.g. [46]).

For further stability analysis of fast matrix multiplication methods, we also mention the papers of Bini and Lotti [3], and Demmel et al. [18]. An excellent survey of the matter is given in Higham [39].

The gaps between bounds (10), (11) and (12) indicate the possibility of numerical instability already observed in some cases (see, e.g. [38], [39]. Here we make a systematic error analysis of Winograd's first method, the Strassen and Winograd-Strassen methods using an improved version of Miller's automatic error analyzer.

8.3.1 Numerical testing

Here we analyze the traditional matrix multiplication, Winograd's first algorithm (7), the Srassen algorithm (8) and the Winograd-Strassen algorithm (9).

The Matlab programs of the above algorithms can be found in Algorithm 13-17:

Algorithm 13 Winograd's method

```
function C=Winograd(A,B)
[m,k] = size( A );
[l,n] = size( B );
assert( k == l, 'Inner matrix dimensions must agrree!' );
assert( mod(k, 2) == 0, 'Inner matrix dimensions must be even!' );
%
rowFactor = feval( class(A), zeros( m, 1 ) );
columnFactor = feval( class(A), zeros( 1, n ) );
for i=1:m
    rowFactor(i) = A(i,1:2:k) * A(i,2:2:k)';
end
for j=1:n
    columnFactor(j) = B(1:2:k,j)' * B(2:2:k,j);
end
C = -repmat( rowFactor, 1, n );
%
C = C - repmat( columnFactor, m, 1 );
%
for i=1:m
    for j=1:n
        C(i,j)=C(i,j)+(A(i,1:2:k)+B(2:2:k,j)')*(A(i,2:2:k)'+B(1:2:k,j));
    end
end
```

There have been a comparison between the standard matrix multiplication and the three faster algorithms. Furthermore there was a comparison between the Strassen and the recursive Winograd methods. The initial matrices A and B were

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \; B = \begin{bmatrix} 4 & 1 & 1 & 1 \\ 1 & 4 & 1 & 1 \\ 1 & 1 & 4 & 1 \\ 1 & 1 & 1 & 4 \end{bmatrix}$$

The obtained result are the following.

Algorithm 14 Strassen's method

```
 1:  function C=Strassen(A,B)
 2:  [m,k] = size( A );
 3:  [l,n] = size( B );
 4:  assert( m == k , 'A must be a squere matrix!' );
 5:  assert( k == l , 'B must be a squere matrix!' );
 6:  assert( l == n, 'A and B must be the same order!' );
 7:  k = uint32( m );
 8:  assert( bitand( k, k - 1 ) == 0, 'n must be the form of 2^k' );
 9:  %
10:  if n == 1
11:      C = A * B;
12:      return;
13:  end
14:  d = n / 2;
15:  d1 = d + 1;
16:  %
17:  A11=A(1:d,1:d);
18:  A12=A(1:d,d1:n);
19:  A21=A(d1:n,1:d);
20:  A22=A(d1:n,d1:n);
21:  %
22:  B11=B(1:d,1:d);
23:  B12=B(1:d,d1:n);
24:  B21=B(d1:n,1:d);
25:  B22=B(d1:n,d1:n);
26:  %
27:  [C11,C12,C21,C22]=Str2x2(A11,A12,A21,A22,B11,B12,B21,B22);
28:  %
29:  C(1:d,1:d)=C11;
30:  C(1:d,d1:n)=C12;
31:  C(d1:n,1:d)=C21;
32:  C(d1:n,d1:n)=C22;
```

Algorithm 15 Strassen's 2 x 2 matrix product

```
 1:  function [C11,C12,C21,C22]=Str2x2(A11,A12,A21,A22,B11,B12,B21,B22)
 2:  % Strassen 2 x 2 matrix product
 3:  M1=Strassen( A12-A22, B21+B22 );
 4:  M2=Strassen( A11+A22, B11+B22 );
 5:  M3=Strassen( A11-A21, B11+B12 );
 6:  M4=Strassen( A11+A12, B22 );
 7:  M5=Strassen( A11, B12-B22 );
 8:  M6=Strassen( A22, B21-B11 );
 9:  M7=Strassen( A21+A22, B11 );
10:  %
11:  C11=M1+M2-M4+M6;
12:  C12=M4+M5;
13:  C21=M6+M7;
14:  C22=M2-M3+M5-M7;
```

First Winograd multiplication:

```
[A, B] = maxsearchcmp( @Winograd, @nmult,...
 AI, BI, @er1vs2, 'mds', 1.0e7 )
The chosen error measuring number is er.
The error measuring number at the initial data: 2.48485
There are no constraints
The stopping value is: 1e+007
The choosen search method is MULTIDIRECTIONAL SEARCH method.
Starting the maximizer...
 Column 1 gives the number of evaluations,
 column 2 gives the current error measuring value.
 160 5.100921e+000
 320 5.907460e+001
 480 2.411424e+002
 640 2.289525e+003
 800 7.975595e+003
 960 7.265662e+004
 1120 2.546605e+005
 1280 2.325295e+006
 !!!Instability located!!!
 After 1396 evaluations the error measuring number: 1.046806e+007
```

The output matrices are

Algorithm 16 Winograd's recursive method

```
 1:  function C=WStrassen(A,B)
 2:  [m,k] = size( A );
 3:  [l,n] = size( B );
 4:  assert( m == k , 'A must be a squere matrix!' );
 5:  assert( k == l , 'B must be a squere matrix!' );
 6:  assert( l == n, 'A and B must be the same order!' );
 7:  k = uint32( m );
 8:  assert( bitand( k, k - 1 ) == 0, 'n must be the form of 2^k' );
 9:  %
10:  if n == 1
11:      C = A * B;
12:      return;
13:  end
14:  d = n / 2;
15:  d1 = d + 1;
16:  A11=A(1:d,1:d);
17:  A12=A(1:d,d1:n);
18:  A21=A(d1:n,1:d);
19:  A22=A(d1:n,d1:n);
20:  %
21:  B11=B(1:d,1:d);
22:  B12=B(1:d,d1:n);
23:  B21=B(d1:n,1:d);
24:  B22=B(d1:n,d1:n);
25:  %
26:  [C11,C12,C21,C22]=WStr2x2(A11,A12,A21,A22,B11,B12,B21,B22);
27:  C(1:d,1:d)=C11;
28:  C(1:d,d1:n)=C12;
29:  C(d1:n,1:d)=C21;
30:  C(d1:n,d1:n)=C22;
```

Algorithm 17 Winograd variant of Strassen's 2 x 2 matrix product

```
 1:  function [C11,C12,C21,C22]=WStr2x2(A11,A12,A21,A22,B11,B12,B21,B22)
 2:  % Winograd variant of Strassen 2 x 2 matrix product
 3:  S1=A21+A22;
 4:  S2=S1-A11;
 5:  S3=A11-A21;
 6:  S4=A12-S2;
 7:  S5=B12-B11;
 8:  S6=B22-S5;
 9:  S7=B22-B12;
10:  S8=S6-B21;
11:  %
12:  M1=WStrassen( S2, S6 );
13:  M2=WStrassen( A11, B11 );
14:  M3=WStrassen( A12, B21 );
15:  M4=WStrassen( S3, S7 );
16:  M5=WStrassen( S1, S5 );
17:  M6=WStrassen( S4, B22 );
18:  M7=WStrassen( A22, S8 );
19:  %
20:  T1=M1+M2;
21:  T2=T1+M4;
22:  %
23:  C11=M2+M3;
24:  C12=T1+M5+M6;
25:  C21=T2-M7;
26:  C22=T2+M5;
```

```
A =

   0.5806   0.5806   0.5806   0.5806
   0.5806   0.5806   0.5806   0.5806
   0.5806   0.5806   0.5806   0.5806
   0.5806   0.5806   0.5806   0.5806


B =

  1.0e+009 *

   4.5555   0.0000   0.0000   0.0000
  -3.0370   0.0000   0.0000   0.0000
  -1.5185   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000
```

The result shows the great instability of this method, which has been known since the 70s.

Analysing Strassen's algorithm, we obtained:

```
>> [A, B] = maxsearchcmp( @Strassen, @nmult,...
 AI, BI, @er1vs2, 'mds', 1.0e10 )
The chosen error measuring number is er.

The error measuring number at the initial data: 9.46275
There are no constraints
The stopping value is: 1e+010

The choosen search method is MULTIDIRECTIONAL SEARCH method.
Starting the maximizer...

 Column 1 gives the number of evaluations,
 column 2 gives the current error measuring value.

            160   1.460319e+001
            320   4.830682e+001
            480   1.215640e+002
            640   2.706130e+002
            800   4.342814e+002
            960   1.672311e+003
           1120   3.132267e+003
```

```
1280   9.683686e+003
1440   1.456509e+004
1600   3.567706e+004
1760   4.432362e+005
1920   4.434299e+005
2080   4.443088e+005
2240   4.497564e+005
2400   4.735137e+005
2560   6.715336e+005
2720   1.404269e+006
2880   5.442865e+006
3040   1.978842e+007
3200   1.998272e+008
3360   5.313679e+008
3520   5.144812e+009


 !!!Instability located!!!

 After 3571 evaluations the error measuring number: 1.139389e+010
```

The maximizer stopped at the input matrices:

```
A =

  1.0e+005 *

   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000  -0.0000
   0.0000   0.0000   0.0000   4.0779


B =

  1.0e+005 *

   1.4831   0.0000   0.0000   0.0000
  -5.5611   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000
```

According to the result the method of Strassen has much worse stability properties then the standard matrix multiplication.

Testing Winograd's recursive algorithm, we could find instability as well:

```
>> [A, B] = maxsearchcmp( @WStrassen, @nmult,...
 AI, BI, @er1vs2, 'mds', 1.0e10 )
The chosen error measuring number is er.

The error measuring number at the initial data: 4.82675
There are no constraints
The stopping value is: 1e+010

The choosen search method is MULTIDIRECTIONAL SEARCH method.
Starting the maximizer...

 Column 1 gives the number of evaluations,
 column 2 gives the current error measuring value.

          160   2.145500e+001
          320   8.222295e+001
          480   7.551987e+001
          640   1.844033e+002
          800   3.575004e+002
          960   8.844130e+002
         1120   1.522865e+003
         1280   7.545635e+003
         1440   9.260712e+003
         1600   2.968749e+004
         1760   5.642213e+004
         1920   9.391043e+004
         2080   4.484037e+005
         2240   6.285645e+005
         2400   5.545539e+005
         2560   4.675970e+005
         2720   3.825951e+006
         2880   9.186001e+006
         3040   1.219495e+007
         3200   3.139133e+007
         3360   3.983461e+007
         3520   8.281908e+007
         3680   1.748969e+008
```

```
          3840   6.305491e+008
          4000   8.937694e+008
          4160   3.681177e+009
          4320   5.593516e+009
          4480   7.274456e+009


 !!!Instability located!!!

 After 4543 evaluations the error measuring number: 1.011707e+010
```

The enormous value of the error measuring number was revealed at:

```
A =

   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000  -0.0166   0.0000
  -0.0000   0.0000   7.7561   8.4865


B =

   3.0000   0.0000  -0.0000   0.0000
   0.0000   3.0000   0.0000 -16.9706
   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000  -0.0000
```

We have also compared the recursive algorithm of Winograd with Strassen's method:

```
>> [A, B] = maxsearchcmp( @WStrassen, @Strassen,...
 AI, BI, @er1vs2, 'mds', 1.0e10 )
The chosen error measuring number is er.

The error measuring number at the initial data: 0.510079
There are no constraints
The stopping value is: 1e+010

The choosen search method is MULTIDIRECTIONAL SEARCH method.
Starting the maximizer...
```

```
 Column 1 gives the number of evaluations,
 column 2 gives the current error measuring value.

           160   1.283606e+000
           320   2.531995e+001
           480   9.834933e+001
           640   8.074625e+002
           800   2.873184e+003
           960   2.777216e+004
          1120   1.428901e+005
          1280   2.184651e+005
          1440   4.762304e+006
          1600   1.642535e+007
          1760   1.579126e+008
          1920   1.323460e+008
          2080   4.544618e+008


 !!!Instability located!!!

 After 2144 evaluations the error measuring number: 1.010641e+010
```

The result shows that Winograd's method has worse stability properties than the algorithm of Strassen. The great value was found at:

```
A =

  1.0e+010 *

   0.0000   0.0000   0.0000   0.0000
   0.0000   0.9256   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000
   0.0000  -2.5453   0.0000   0.0000


B =

  1.0e+010 *

   0.0000  -0.0000   0.0000   3.9336
   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000  -2.3139
```

```
0.0000    0.0000    0.0000    0.0000
```

According to the presented numerical results and many others not shown here we can firmly establish the following conclusions:

1. The classical Winograd scalar product based matrix multiplication algorithm of $O(n^3)$ operation cost is highly unstable in accordance with the common belief but never justified in a formal way.

2. Both the Strassen and Winograd recursive matrix multiplication algorithms of $O(n^{2.81})$ operation costs are numerically unstable.

3. The comparative testing indicates that Strassen's algorithm has somewhat better numerical stability than those of Winograd.

The obtained results support the common but disputed opinion that these fast matrix multiplication methods are numerically unstable (for reference, see, e.g. Higham [39]).

9 Summary of the results

The numerical stability of computational algorithms is a very important issue. The classic error analysis techniques very rarely give computationaly feasible results for the practitioner. The classical numerical testing on selected test examples often misleading depending on the selection and properties of the test problems. This thesis joins a long line of research that aims an automatic error analysis based on compiler techniques, the computational graph techniques, the automatic differentiation techniqes, object oriented programming. Here the aim is that by simply using the written program of an algorithm under consideration the average or any other user can have a reliable estimate of the numerical stability without blind test problem selection. This line of research was initiated by W. Miller, who developed the most advanced program system and its theory. Although many researcher developed similar or partly similar systems none of them achieved the high level of Miller's solution. Miller's solution however was limited in use by the computer technique of his age. In this thesis I analyzed, improved, upgraded and reimplemented his method. The results of my research can summarized as follows.

Thesis 1

I replaced the minicompiler and its simplified programming language of the Miller method to object oriented Matlab.

Thesis 2

Upon the bases of computer testing and theory I added two new optimization methods to the system that improved the performance of the software.

Thesis 3

I reprogrammed and tested the system in Matlab. The new 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. The analyzed numerical algorithm can be given in the form of a Matlab m-file. Hence our software is easy to use. The program consists of about 10000 lines and it is downloadable from the site

http://phd.uni-obuda.hu/images/milleranalyzer.zip

together with a detailed user guide.

Thesis 4

I applied the new Matlab version to investigate the numerical stability of some ABS methods (implicit LU, Huang and its variants), and three fast matrix multiplication algorithms. The obtained results indicate numerical instability of various scale and in the case of fast matrix multiplication algorithms give a definite yes for the suspected numerical instability of these methods.

References

[1] Abaffy, J., Broyden, C. G., Spedicato, E., A class of direct methods for linear systems, Numerische Mathematik, 45, 1984, 361–376

[2] Abaffy J., Spedicato, E.: ABS Projection Algorithms: Mathematical Techniques for Linear and Nonlinear Equations, Ellis Horwood, 1989

[3] Bini, D., Lotti, G.: Stability of fast algorithms for matrix multiplication, Numerische Mathematik, 36, 1980, 63–72

[4] Bischof, C.H., Bücker, H.M., Hovland, P., Naumann, U., Utke, J.(eds.): Advances in Automatic Differentiation, Springer, 2008

[5] Bliss, B.: Instrumentation of FORTRAN programs for automatic roundoff error analysis and performance evaluation, MSc thesis, University of Illinois at Urbana-Champaign, 1990

[6] Bliss, B., Brunet, M.-C., Gallopoulos, E.: Automatic parallel program instrumentation with applications in performance and error analysis, In Expert Systems for Scientific Computing, E. N. Houstis, J. R. Rice, and R. Vichnevetsky, editors, North-Holland, Amsterdam, The Netherlands, 1992, 235–260

[7] Brent, R.P.: Algorithms for matrix multiplication, Technical Report STAN-CS-70-157, Computer Science Department, Stanford University, 1970

[8] Brent, R.P.: Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity, Numerische Mathematik, 16, 1970, 145–156

[9] Byatt, D.: Convergent variants of the Melder-Nead algorithm, MSc thesis, University of Canterbury, 2000

[10] Castano, B., Heintz, J., Llovet, J., Martinez, R.: On the data structure straight-line program and its implementation in symbolic computation, Mathematics and Computers in Simulation, Volume 51, Number 5, February 2000 , pp. 497-528(32)

[11] Chaitin-Chatelin, F., Frayssé, V.: Lectures on Finite Precision Computations, SIAM, Philadelphia, 1996

[12] Coleman, T.F., Verma, A.: ADMAT: An Automatic Differentiation Toolbox for MATLAB, Computer Science Department, Cornell University, 1998

[13] Conn, A.R., Scheinberg, K. Vicente, L.N.: Introduction to derivative-free optimization, SIAM, Philadelphia, 2008

[14] Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions, Journal of Symbolic Computation, 9, 1990, 251–280

[15] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edition, The MIT Press, McGraw Hill, Cambridge, 2001

[16] D'Alberto, P., Nicolau, A.: Adaptive Strassen's matrix multiplication, In Proceedings of the 21st Annual International Conference on Supercomputing, ACM, New York, NY, 2007, 284–292.

[17] D'Alberto, P., Nicolau, A.: Adaptive Winograd's matrix multiplications, ACM Transactions on Mathematical Software, Vol. 36, No. 1, Article 3, DOI 10.1145/1486525.1486528 http://doi.acm.org/10.1145/1486525.1486528

[18] Demmel, J., Dumitriu, I., Holtz, O., Kleinberg, R.: Fast matrix multiplication is stable, Numerische Mathematik, 106, 2007, 199–224, DOI 10.1007/s00211-007-0061-6

[19] Douglas, C., , Heroux, M., Slishman, G., Smith, R.M.: GEMMW: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm, J. Comp. Phys., 110, 1994, 1–10

[20] Dumitrescu, B.: Improving and estimating the accuracy of Strassen's algorithm, Numerische Mathematik, 79, 1998, 485–499

[21] Einarsson, B. (ed): Accuracy and reliability in scientific computing, SIAM, Philadelphia, 2005

[22] Forth, S.A.: An efficient overloaded implementation of forward mode automatic differentiation in MATLAB, ACM Trans. Math. Softw., 32, 2006, 195–222

[23] Gao, F., Han, L.: Implementing the Nelder-Mead simplex algorithm with adaptive parameters, Comput. Optim. Appl., DOI 10.1007/s10589-010-9329-3

[24] Gáti Attila: Numerikus algoritmusok automatikus hibaanalízise Miller módszerével, Doktoranduszok Fóruma, Gépészmérnöki Kar Szekciókiadványa, Miskolci Egyetem 2003, 70–77

[25] Gáti, A.: Automatic error analysis with Miller's method. Miskolc Math. Notes, 5, 1, 2004, 25–32.

[26] Gáti, A.: Automatic roundoff error analysis with Miller's method, MicroCAD 2004, International Scientific Conference, Applied Mechanics, Modern Numerical Methods, University of Miskolc 2004, 29–34

[27] Gáti Attila: A Miller-Spooner-féle automatikus hibaelemző program továbbfejlesztése, Doktoranduszok Fóruma, Gépészmérnöki Kar Szekciókiadványa, Miskolci Egyetem 2004, 83–87

[28] Gáti, A.: The upgrading of the Miller-Spooner roundoff analyser software, MicroCAD 2005, International Scientific Conference, Applied Mechanics, Modern Numerical Methods, Miskolci Egyetem 2005, 49–54, ISBN 963 661 653 1

[29] Gáti, A.: The upgrading of the Miller-Spooner roundoff analyser software, 5th International Conference of PhD Students, Engineering Sciences II, Miskolci Egyetem 2005, 43–48. ISBN 963 661 679 5

[30] Gáti, A.: A Miller-Spooner-féle automatikus hibaelemző program továbbfejlesztése (The upgrading of the Miller-Spooner roundoff analyzer 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), ISBN 963 472 909 6

[31] Gáti Attila: Hibaelemzés és automatikus differenciálás, Doktoranduszok Fóruma, Gépészmérnöki Kar Szekciókiadványa, Miskolci Egyetem 2005, 56–62

[32] Gáti, A.: Roundoff error analysis and automatic differentiation, MicroCAD 2006, International Science Conference, Mathematics and Computer Science, Miskolci Egyetem 2006, 29–35. ISBN 963 661 707 4

[33] Gáti Attila: Hibaelemzés és automatikus differenciálás, Tavaszi Szél 2006 Konferenciakiadvány, Kaposvár 2006, 287–290. ISBN 963 229 773 3

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

[35] Gáti, A.: Miller Analyzer for Matlab: A Matlab Package for Automatic Roundoff Analysis, Computing and Informatics, 31, 2012, 713–726

[36] Golub, G.H., Van Loan, C.F.: Matrix Computations, 2nd ed., Johns Hopkins University Press, 1989

[37] Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, SIAM, 2000

[38] Higham, N. J.: Exploiting Fast Matrix Multiplication Within the Level 3 BLAS, ACM Transactions on Mathematical Software, 16, 4, 1990, 352–368

[39] Higham, N. J.: Accuracy and Stability of Numerical Algorithms, SIAM, 1996

[40] Kaporin, I.: A practical algorithm for faster matrix multiplication, Numer. Linear Algebra Appl., 6, 1999, 687–700

[41] Kaporin, I.: The aggregation and cancellation techniques as a practical tool for faster matrix multiplication, Theoretical Computer Science 315, 2004, 469–510

[42] Kolda, T.G., Lewis, R.M., Torczon, V.: Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods, SIAM Review, 45, 2003, 385–482

[43] Krämer, W.: Constructive Error Analysis, Journal of Universal Computer Science, vol. 4, no. 2 (1998), 147-163

[44] Krämer, W., Bantle, A.: Automatic Forward Error Analysis for Floating Point Algorithms, Reliable Computing 7, 2001, 321–340

[45] Lagarias, J.C., Reeds, J.A., Wright, M.H., Wright, P. E.: Convergence Properties of the Nelder–Mead Simplex Method in Low Dimensions, SIAM J. Optim., 9 (1999) 112–147

[46] Lakshmivaran, S., Dhall, S.K.: Analysis and Design of Parallel Algorithms, McGraw-Hill, 1990

[47] Larson, J.L., Pasternak, M.E.,Wisniewski, J.A.: Algorithm 594: Software for Relative Error Analysis, ACM Trans. Math. Softw., 9, 1983, 125–130

[48] Lewis, R.M., Torczon, V., Trosset, M.W.: Direct search methods: then and now, Journal of Computational and Applied Mathematics 124 (2000) 191–207

[49] Miller, W.: On the stability of finite numerical procedures, Numerische Mathematik, 19, 1972, 425–432

[50] Miller, W.: Computational complexity and numerical stability, in Proceeding STOC '74 Proceedings of the sixth annual ACM symposium on Theory of Computing, 1974, 317–322

[51] Miller, W.: Computational complexity and numerical stability, SIAM J. Comput., 4, 2, 1975, 97–107

[52] Miller, W.: Computer search for numerical instability, JACM, 22, 4, 1975, 512–521

[53] Miller, W.: Software for roundoff analysis, ACM Trans. Math. Softw., 1, 2, 1975, 108–128

[54] Miller, W.: Roundoff analysis by direct comparison of two algorithms, SIAM Journal on Numerical Analysis, 13, 3, 1976, 382–392

[55] Miller, W.: Roundoff Analysis and Sparse Data, Numerische Mathematik, 29, 1977, 37–44

[56] Miller, W., Spooner, D.: Software for roundoff analysis, II, ACM Trans. Math. Softw., 4, 4, 1978, 369–387

[57] Miller, W., Spooner, D.: Algorithm 532: software for roundoff analysis [Z], ACM Trans. Math. Softw., 4, 4, 1978, 388–390

[58] Miller, W., Wrathall, C.: Software for roundoff analysis of matrix algorithms, Academic Press, New York, 1980

[59] Monte Leon, V.J.: Automatic error analysis in finite digital computations, using range arithmetic, MSc thesis, U.S. Naval Postgraduate School, Monterey, California, 1966

[60] Moore, R.E.: Automatic Error Analysis in Digital Computation, Technical report, LMSD-48421, Lockheed, 28 January, 1959

[61] Moore, R.E.: Methods and Applications of Interval Analysis, SIAM, Philadelphia, 1979

[62] Muller, J.-M., et al.: Handbook of Floating-Point Arithmetic, Birkhäuser, 2010

[63] Mutrie, M.P.W., Bartels, R.H., Char, B.W.: An approach for floating-point error analysis using computer algebra, Proceeding ISSAC '92 Papers from the international symposium on Symbolic and algebraic computation, ACM New York, NY, USA, 1992, 284–293

[64] Nelder, J.A., Mead, R.: A simplex method for function minimization, Computer Journal, 7 (1965) 308–313

[65] Overton, M.L.: Numerical computing with IEEE floating point arithmetic, SIAM, Philadelphia, 2001

[66] Pan, V.: How can we speed up matrix multiplication?, SIAM Review, 26, 3, 1984, 393–415

[67] Rall. L.B.: Automatic Differentiation: Techniques and Applications, Lecture Notes in Computer Science, 120, Springer, 1981

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

[69] Rice, J.R. : Numerical Methods, Software, and Analysis, McGraw-Hill, 1983

[70] Rice, J.R.: Matrix Computations and Mathematical Software, McGraw-Hill, 1983

[71] Rosenbrock, H.H.: An automatic method for finding the greatest or least value of a function, Comput. J., 3, 1960, 175–184

[72] Rowan, T.H.: Functional Stability Analysis of Numerical Algorithms. Ph.D. thesis, University of Texas at Austin, Austin, 1990

[73] Stoutemyer, D.R.: Automatic error analysis using computer algebraic manipulation. ACM Trans. Math. Software, 3, 1977, 26–43

[74] Strassen, V.: Gaussian elimination is not optimal, Numerische Mathematik, 13, 1969, 354–356

[75] Torczon, V.: On the convergence of the multidirectional search algorithm, SIAM Journal on Optimization, 1, 1, 1991, 123–145

[76] Tucker, W.: Validated numerics: a short introduction to rigorous computations, Princeton University Press, 2011

[77] Ueberhuber, C.W.: Numerical Computation 1-2 (Methods, Software, and Analysis), Springer, 1997

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

[79] Wilkinson, J.H.: The Algebraic Eigenvalue Problem, Oxford University Press, 1965

[80] Wilkinson, J.H.: Rounding Errors in Algebraic Processes, Dover, 1994

[81] Winograd, S.: A new algorithm for inner product, IEEE Trans. C-17, 1968, 693–694

[82] Winograd, S.: On multiplication of 2×2 matrices, Linear Algebra Appl., 4, 1971, 381–388