Miller Analyser for MATLAB
User's Manual

Attila Gáti[*]

July 12, 2010

1 Introduction

Miller Analyser for MATLAB is an automatic roundoff error analyser software, that extends the work of Miller et al. [1, 2, 3, 4, 5]. The software runs within the MATLAB environment, and can test the stability of numerical methods given as m-functions. Based on the algorithm of Miller, a number $\omega\left(d\right)$ is associated with each set d of input data. The function $\omega\left(d\right)$ measures rounding error, i.e. $\omega\left(d\right)$ is large exactly when the method applied to d produces results which are excessively sensitive to rounding errors. A numerical maximizer is applied to search for large values of ω. Finding large values of ω can be interpreted, that the given numerical method is suffering from a specific kind of instability.

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 competing numerical methods, which neglecting rounding errors compute the same values.

The analysis is based on the standard model of floating point arithmetic, which assumes that the individual relative rounding errors on arithmetic operations are bounded by the machine rounding unit. Practically, the computed result equals the correctly rounded exact result. The IEEE 754/1985 standard of floating point arithmetic guarantees that the standard model holds for addition, subtraction, multiplication, division and square root. Unfortunately, it is not true for the exponential, trigonometrical, hyperbolical functions and their inverses. Hence, we can analyse only numerical algorithms that can be decomposed to the above mentioned five basic operations and unary minus, which is considered error-free.

The first step of computing the error function $\omega\left(d\right)$ is building the computational graph of the analysed numerical method. Decomposition of a numerical method at a particular input $d = d_0$ to the allowed arithmetic operations give rise to a directed acyclic graph, the computational graph, with a node for each input value, output value and operation. There are arcs from each arithmetic node (ie., one corresponding to an operation) to the nodes for its operands and from each output node to the operation that computes its value.

According to the resulting computational graph the output computed as a function: $R_{d_0}\left(d, \delta\right)$ ($R_{d_0} : \mathbb{R}^{n+m} \rightarrow \mathbb{R}^k$), 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 ($\delta \in \mathbb{R}^m$, $\left\|\delta\right\|_\infty \leq u$, where u is the machine rounding unit). The computation of $\omega\left(d\right)$ is based on the partial derivatives of R_{d_0} with respect to the input and the rounding errors. We apply automatic differentiation on the graph in reverse order, ie. the chain rule is applied in the opposite direction as the basic operations were executed.

---

[*] Óbuda University, Bécsi út 96/b 1034 Budapest, Hungary, email: matgati@gmail.com

2 The contents of the package

The `miller/src` directory contains the C++ and Fortran language source files. The C++ source can be found in the `roundoff`, while the FORTRAN language source files in the `forround` directory. These source files must be compiled and linked into one MATLAB MEX file (see MATLAB External Interfaces Reference for detailed information on creating C-language mex files [6]). The resulting mex file must be named `roundoff.<mexext>`, where `<mexext>` is the platform dependent file extension of mex files (on 32 bit Windows platforms with MATLAB 7.1 or later the extension is `mexw32`).

A win32 version of the `roundoff` MEX file compiled for MATLAB 7.4.0 can be found in `miller/bin`. For other platforms or versions of MATLAB it may be required to be recompiled to work properly. Communication between MATLAB and `roundoff.<mexext>` is implemented through the interface of a MATLAB class called `miller`. The m-files defining class miller resides in `bin/@miller`. The analysed numerical method can be given in the form of MATLAB m-functions. To build the computational graph, a special class called `cfloating` has to be used instead of the built-in MATLAB arrays. The `bin/@cfloating` directory contains the corresponding m-files. In addition an object of class named `miller_ptr` must also be used in the m-file implementing the analysed algorithm. Actually this object represents the computational graph being built. `bin/@miller_ptr` is the implementation directory for that class. The `miller/examples` directory contains some examples with numerical methods capable for analysis.

3 Installation

The `miller/bin` directory has to be added to the MATLAB path. Start MATLAB, and run `setpath.m` to add the required directories to your MATLAB path.

4 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.

4.1 The main m-function

Algorithm 1 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 1 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

---

[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.

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.

4.2 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.

4.2.1 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 1 line 2.

4.2.2 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.

4.2.3 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 $\|Ax - b\| \rightarrow \min$. 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.

4.3 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.

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

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

8. Square root. $\mathrm{sqrt}\left(A\right)$ 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.

---

[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.

4.3.1 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 $C = \text{zeros}\left(\text{size}\left(B\right)\right) + B$ 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.

4.3.2 The function cfloating()

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

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

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.

4.3.3 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 $D = [\text{cfloating}\left(A\right), \text{cfloating}\left(B\right), \text{cfloating}\left(C\right)]$.

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').

---

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

Algorithm 2 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
```

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 2 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.

4.3.4 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:

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

   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.

4.3.5 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.

4.3.6 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.

4.3.7 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.

4.4 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.

5 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.

5.1 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*.

5.2 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.

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.

5.3 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 4.2.2). p is the parameter vector. Its entries can be reached in the input algorithm by the parameter statement as in section 4.2.3 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:

   $[\mathrm{rho}, \mathrm{dfinal}] = \mathrm{maxsearch}\,(m, \mathrm{dinit}, \mathrm{methodcode})$
   $[\mathrm{rho}, \mathrm{dfinal}] = \mathrm{maxsearch}\,(m, \mathrm{dinit}, \mathrm{methodcode}, p)$

   `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.

5.3.1 Error handling

We have mentioned in section 4.3.7, 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 4.3.7 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'.

5.3.2 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!

5.3.3 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.

6 Gaussian elimination, an example

It is a good practice to make a separate directory for a particular numerical problem, and place the m-files implementing algorithms solving that problem into the same directory. The miller/examples/LinearSystem is an example directory for analysing algorithms solving the linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ nonsingular matrix and $b \in \mathbb{R}^n$. We will see, how to analyse gaussian elimination without pivoting and with partial pivoting.

6.1 Gaussian elimination without pivoting

6.1.1 The main m-function

The main m-function for gaussian elimination without pivoting is gaussM.m (algorithm 3) Algorithm 3 follows the general scheme as described in section 4.1. By calling subroutines we firs initialize the input, matrix A, and vector b in line 3. With the cfloating variables we call the algorithm itself (line 4), and finally we register the output x (line 5).

Algorithm 3 gaussM

```
1: function gaussM(miller)
2: miller=miller_ptr(miller);
3: [A,b] = InitInput(miller);
4: x = gauss(A,b);
5: output(x);
```

6.1.2 Initializing the input

Algorithm 4 InitInput

```
function [A, b] = InitInput(miller)
n = parameter(miller,1);
if n * n + n ~= input_size(miller)
    error( 'Number of inputs does not match to the size parameters!' );
end
A = input(miller,n,n);
b = input(miller,n,1);
```

In algorithm 4 we initialize the input and add the input nodes to the computational graph. As parameter, it is reasonable to pass the size of the linear system n. We do not need other parameters. We access our parameter in line 2. The inputs of the algorithm are the entries of the extended matrix $\begin{bmatrix} A & b \end{bmatrix}$, which has $n\,(n+1)$ elements. In line 3 we check out if the parameter served by the user is inconsistent with the size of the input data, and if so we print an error message and terminate the m-file by calling error. Beside parameters, the other decision that a priori has to be made, is how to arrange the input into a single vector. In this case, we decided to vectorize the extended matrix in column major order. Accordantly, we initialize the input as cfloating variables, and add the $n\,(n+1)$ input nodes to the graph in line 6 and 7. The first statement reads n^2 elements from the input vector and initializes A in column major order, the second reads n elements and initializes the vector b with them.

6.1.3 The algorithm

Algorithm 5 gauss

```
 1:  function x = gauss( A, b )
 2:  [n,m] = size(A);
 3:  assert( n == m, 'A is not a square matrix!' );
 4:  m = numel( b );
 5:  assert( n == m, 'b must have as many elements as the columns of A!' );
 6:  x = zeros( n, 1 );
 7:  x = feval( class(A), x );
 8:  for k = 1 : n - 1
 9:      for i = k + 1 : n
10:          amult = A(i,k) / A(k,k);
11:          A(i,k+1:n) = A(i,k+1:n) - amult * A(k,k+1:n);
12:          b(i) = b(i) - amult * b(k);
13:      end
14:  end
15:  for i = n : -1 : 1
16:      x(i) = ( b(i) - A(i,i+1:n) * x(i+1:n) ) / A(i,i);
17:  end
```

Algorithm 5 implements gaussian elimination in such way, that it can be called both with cfloating and with built-in typed input arguments. From line 2 to 5 we assign the size of the system to n, and check the sizes of A and b for integrity. At line 6 we preallocate the output vector x. Line 7 converts x to the same type as A (see MATLAB help for functions feval and class). If A is of type double, it has no effect, but if it is of type cfloating, then it performs explicit conversion to cfloating as described in section 4.3.2. From line 8 to 17, we compute the solution of the system by Gaussian elimination without pivoting. If the function was called with cfloating arguments, the arithmetic nodes are also added to the computational graph.

In the following we will follow the naming convention, that an input algorithm implemented with a name 'algname' will have a corresponding main m-function 'algnameM'.

6.1.4 Error analysis

We have made two auxiliary routines VecInput and UnvecInput to make the analysis more convenient. Algorithm 6 produces the vectorized input d and the parameter p from a matrix A and a vector b of appropriate size. Algorithm 7 transforms back the parameter p and the input data d to the matrix A of coefficients and the vector b at the right-hand side.

Algorithm 6 VecInput

```
function [ d, p ] = VecInput( A, b )
[m,n] = size( A );
k = numel( b );
assert( m == n, 'A is not a square matrix!' );
assert( m == k, 'b must have as many elements as the columns of A!' );
d = [A, b];
d = d(:);
p = n;
```

Algorithm 7 UnvecInput

```
function [ A, b ] = UnvecInput( d, p )
n = p(1);
A = reshape( d(1:n * n), n, n );
b = reshape( d(n * n + 1 : end), n, 1 );
```

The function run is for computing the solution x (algorithm 8) of the system $Ax = b$ with algorithm Alg (Alg is a function handle). First we vectorize the input (line 2). In line 3 we create a miller object with the name of the main m-function as argument. According to our naming convention the name of the main m-function is the name corresponding to the function handle Alg with an 'M' at the end. We run the algorithm (line 4) and display some information (line 5). Finally at line 6 we free up all the memory allocated.

Algorithm 8 run

```
1: function x = run( Alg, A, b )
2: [d, p] = VecInput( A, b );
3: M = miller( [func2str(Alg), 'M'] );
4: x = run( M, d, p );
5: display( M );
6: destroy( M );
```

The function rho works very similarly to run, but instead computing the output vector it computes all the error measuring quantity for the given algorithm, at the given input A and b.

Algorithm 9 maxsearch

```
function [A, b, r] = maxsearch( Alg, A, b, emv, OptMthd, StopCrit )
[d, p] = VecInput( A, b );
M = miller( [func2str( Alg ), 'M'] );
emv( M, d, p );
set( M, 'stop_crit', StopCrit );
[r, d] = maxsearch( M, d, OptMthd, p );
[A,b] = UnvecInput( d, p );
display( M );
destroy( M );
```

The function maxsearch (algorithm 9) performs error maximization. The arguments:

1. Alg: function handle to the algorithm to analyse, just as above.
2. A and b: the input from which the maximization will start.
3. emv: The error measuring quantity in the form of a function handle to call a function, mentioned in section 5.3 at list item 2.
4. OptMthd: The method of optimization, that can be 'ros', 'nms', 'mds' to perform maximization using the Rosenbrock, the Nelder-Mead simplex, or the Multidirectional Search respectively.
5. StopCrit: The stopping criterion. The maximization terminates, if this value is reached. Zero turns off testing on reaching a stopping value.

The function returns:

1. A and b: the final set of input
2. r: the value of the maximum that has been found.

Now first lets test gauss.m. Go to the directory `examples\LinearSystem` and type:

```
[A,b] = testdata(4)

A =

     3     1     1     1
     1     4     1     1
     1     1     5     1
     1     1     1     6


b =
     6
     7
     8
     9
```

Let's call gauss with built in typed values values:

```
x=gauss(A,b)

x =

    1.0000
    1.0000
    1.0000
    1.0000
```

The result is correct.

Now let's test with run:

```
x=run(@gauss,A,b)
Object of type miller for analysing algorithm gaussM
The method for evaluating derivatives is: Miller
The error measuring number is: not set
The stopping value is: 0
During last search 1 calls were made
The number of inputs: 20
The number of operations: 62
The number of constants: 1
The number of outputs: 4
The number of constraints: 0
The penalty is: 1
Memory allocated: 5864
Memory used: 5616


x =

    1.0000
    1.0000
    1.0000
    1.0000
```

We get the same result.

Compute the error measuring quantities at A and b:

```
rho(@gauss,A,b)
Object of type miller for analysing algorithm gaussM
The method for evaluating derivatives is: Miller
The error measuring number is: cne = 3.17012
The stopping value is: 0
```

```
During last search 8 calls were made
The number of inputs: 20
The number of operations: 62
The number of constants: 1
The number of outputs: 4
The number of constraints: 0
The penalty is: 1
Memory allocated: 5864
Memory used: 5616


ans =

    wkl: 0.7551
    wke: 1.2732
    jwl: 0.7939
    jwe: 1.4753
    erl: 0.4303
    ere: 1.4763
    cnl: 8.7647
    cne: 3.1701
```

Perform a maximization:

```
[Af,bf,r]=maxsearch(@gauss,A,b,@wkl,'ros',10000)
The chosen error measuring number is wkl.

The error measuring number at the initial data: 0.75507
There are no constraints
The stopping value is: 10000

The choosen search method is ROSENBROCK method.
Starting the maximizer...

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

            100   1.853138e+000
            200   2.234214e+003


 !!!Instability located!!!

 After 209 evaluations the error measuring number: 3.916696e+004


The condition number at the final set of data is: 7.16683

Object of type miller for analysing algorithm gaussM
The method for evaluating derivatives is: Miller
The error measuring number is: wkl = 39167
The stopping value is: 10000
During last search 211 calls were made
The number of inputs: 20
The number of operations: 62
The number of constants: 1
The number of outputs: 4
The number of constraints: 0
The penalty is: 1
Memory allocated: 5864
Memory used: 5616


Stop flag set.
Please call reset or resetcounter before continue.


Af =

    2.9079   -2.2531    6.7751    3.9703
    1.2126    1.9950    2.2656    8.0076
    4.5516    5.6808    8.8495    1.3016
   -5.8092   -5.0303    0.0990    7.8322


bf =

    6.2908
    7.2191
    5.7301
    9.5742


r =

  3.9167e+004
```

So at Af, bf the error measure is more than 39000.

1. gppConstr and gppConstrM implements gaussian elimination with partial pivoting using constraints.
2. gppIf and gppIfM implements gaussian elimination with partial pivoting using row interchanges.

In the directory MatrixMult:

1. nmultM: ordinary matrix multiplication

2. Winograd and WinogradM implements the Winograd algorithm
3. Strassen, StrassenM, Str2x2 implements the strassen algorithm

Let us try the following in directory `examples/MatrixMult`:

```
run(@nmult,A,B)
Object of type miller for analysing algorithm nmultM
The method for evaluating derivatives is: Miller
The error measuring number is: not set
The stopping value is: 0
During last search 1 calls were made
The number of inputs: 32
The number of operations: 112
The number of constants: 1
The number of outputs: 16
The number of constraints: 0
The penalty is: 1
Memory allocated: 24000
Memory used: 23752


ans =

     7     7     7     7
     7     7     7     7
     7     7     7     7
     7     7     7     7

run(@Winograd,A,B)
Object of type miller for analysing algorithm WinogradM
The method for evaluating derivatives is: Miller
The error measuring number is: not set
The stopping value is: 0
During last search 1 calls were made
The number of inputs: 32
The number of operations: 184
The number of constants: 1
The number of outputs: 16
The number of constraints: 0
The penalty is: 1
Memory allocated: 35808
Memory used: 35560


ans =

     7     7     7     7
     7     7     7     7
     7     7     7     7
     7     7     7     7

run(@Strassen,A,B)
Object of type miller for analysing algorithm StrassenM
The method for evaluating derivatives is: Miller
The error measuring number is: not set
The stopping value is: 0
During last search 1 calls were made
The number of inputs: 32
The number of operations: 247
The number of constants: 0
The number of outputs: 16
The number of constraints: 0
The penalty is: 1
Memory allocated: 46140
Memory used: 45884


ans =

     7     7     7     7
     7     7     7     7
     7     7     7     7
     7     7     7     7
```

In all the three cases we got correct results.

```
[Af,Bf,r] = maxsearch(@Strassen,A,B,@wkl, 'ros', 10000)
The chosen error measuring number is wkl.

The error measuring number at the initial data: 18.7382
There are no constraints
The stopping value is: 10000

The choosen search method is ROSENBROCK method.
Starting the maximizer...

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

           160   2.438387e+001
           320   2.791163e+001
           480   3.568908e+001
           640   3.626861e+001
           800   3.631956e+001
           960   3.632914e+001
          1120   3.636743e+001
          1280   3.638377e+001
          1440   3.638796e+001
          1600   3.639103e+001
```

```
           1760   3.667554e+001
           1920   3.805665e+001
           2080   3.879064e+001
           2240   3.929862e+001
           2400   4.051778e+001
           2560   4.172274e+001
           2720   4.191887e+001
3 DIAGON failures.

Object of type miller for analysing algorithm StrassenM
The method for evaluating derivatives is: Miller
The error measuring number is: wkl = 42.2311
The stopping value is: 10000
During last search 2754 calls were made
The number of inputs: 32
The number of operations: 247
The number of constants: 0
The number of outputs: 16
The number of constraints: 0
The penalty is: 1
Memory allocated: 46140
Memory used: 45884

Reporting the number of errors:
DIAGON failure: 3

Af =

    4.6320    2.7085    2.7458    1.0841
    3.0614    5.1903    3.0586    2.4608
    0.4807    2.9094    5.5102    2.0005
   -2.7252    5.9879    3.0533    5.4673


Bf =

    4.7437    2.2323    1.9933   -0.5063
    2.2047    3.4582   -0.9976   -1.7825
   -0.7174   -2.1869    5.0484    0.2749
   -1.5117    2.5341   -0.7688    5.9973


r =

   42.2311
```

We can also compare two algorithms:

```
[AI, BI] = testdata( 4)
```

```
AI =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1


BI =

     4     1     1     1
     1     4     1     1
     1     1     4     1
     1     1     1     4

[A, B] = maxsearchcmp( @Winograd, @nmult, AI, BI, @er1vs2, 'mds', 1.0e15 )
The chosen error measuring number is er.

The error measuring number at the initial data: 2.12158
There are no constraints
The stopping value is: 1e+015

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

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

            160   4.115497e+000
            320   9.580887e+001
            480   3.667175e+002
            640   3.427610e+003
            800   1.204476e+004
            960   1.100727e+005
           1120   3.857973e+005
           1280   3.522651e+006
           1440   1.234582e+007
           1600   1.127252e+008
           1760   3.950666e+008
           1920   3.607208e+009
           2080   1.264213e+010
           2240   1.154306e+011
           2400   4.045482e+011
           2560   3.693781e+012
           2720   1.294554e+013
           2880   1.182010e+014
           3040   4.142574e+014
 !!!Instability located!!!

 After 3066 evaluations the error measuring number: 1.030452e+015


Object of type miller for comparing algorithm WinogradM with nmultM
The method for evaluating derivatives is: Miller
The error measuring number is: er = 1.03045e+015
The stopping value is: 1e+015
During last search 3067 calls were made
The number of inputs: 32
The number of operations: 184 112
The number of constants: 1 1
The number of outputs: 16
The number of constraints: 0
The penalty is: 1
The size of Ukkonen matrix: 0The size of Ukkonen matrix: 0
Memory allocated: 61056
Memory used: 48752


Stop flag set.
Please call reset or resetcounter before continue.


A =

   1.419364052641429   1.419364052641429   1.419364052641429   1.419364052641429
   1.419364052641429   1.419364052641429   1.419364052641429   1.419364052641429
   1.419364052641429   1.419364052641429   1.419364052641429   1.419364052641429
   1.419364052641429   1.419364052641429   1.419364052641429   1.419364052641429


B =

  1.0e+015 *

   0.000000000000004   0.000000000000001  -1.194197188598576   0.000000000000001
   0.000000000000001   0.000000000000004   0.796131459065712   0.000000000000001
   0.000000000000001   0.000000000000001   0.000000000000004   0.000000000000001
   0.000000000000001   0.000000000000001   0.398065729532871   0.000000000000004

Af =

    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
```

```
Bf =

  1.0e+014 *

    3.9807    0.0000    0.0000    0.0000
   -3.9807    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000
    0.0000    0.0000    0.0000    0.0000


r =

  1.2826e+015
```

References

[1] Miller, W. Computer search for numerical instability. *J. ACM 22*, 4 (1975), 512–521.

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

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

[4] Miller, W., and Spooner, D. Software for roundoff analysis, ii. *ACM Trans. Math. Softw. 4*, 4 (1978), 369–387.

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

[6] The MathWorks inc. Matlab external interfaces, version 7., 2004.