Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs *

C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, A. Vehreschild
Institute for Scientific Computing
Aachen University
52056 Aachen, Germany.

Abstract: Derivatives of mathematical functions play a key role in various areas of numerical and technical computing. Many of these computations are done in MATLAB, a popular environment for technical computing providing engineers and scientists with capabilities for mathematical computing, analysis, visualization, and algorithmic development. For functions written in the MATLAB language, a novel software tool is proposed to automatically transform a given MATLAB program into another MATLAB program capable of computing not only the original function but also user-specified derivatives of that function. That is, a program transformation known as automatic differentiation is performed to change the semantics of the program in a fashion based on the chain rule of differential calculus. The crucial ingredient of the tool is a combination of source-to-source transformation and operator overloading. The overall design of the tool is described and numerical experiments are reported demonstrating the efficiency of the resulting code for a sample problem.

Keywords: source transformation, operator overloading, automatic differentiation, derivatives, MATLAB, high-level language

1. Introduction

MATLAB[1] is a widely used environment for technical computing providing a language with high-level data types, operators, and a large set of mathematical functions. Typical applications include numerical simulations and rapid prototyping. In these applications derivatives often play a crucial role, for instance, sensitivity analysis, partial differential equations, and optimization. MATLAB supports the computation of derivatives by numerical differentiation based on divided differences, and also provides symbolic differentiation in one of its toolboxes [19]. Numerical differentiation gives approximations of derivatives, often of dubious quality, which are usually insufficient for sophisticated numerical algorithms relying on exact derivative values [14]. On the other hand, symbolic differentiation is only applicable to small-sized problems where the function to be differentiated is simple enough. If derivatives of functions given in the form of complicated computer programs are required, automatic differentiation offers a powerful alternative. Conceptually, automatic differentiation is a program transformation used to change the semantics of a given program in a fashion based on the chain rule of differential calculus to generate a program for the computation of a function and its derivatives.

Operator overloading has been used to implement automatic differentiation for functions written in MATLAB. Besides the early work reported in [17] another implementation based on operator overloading is the ADMIT/ADMAT [8] toolbox developed at Cornell University. In this note, we propose the first software tool implementing automatic differentiation for MATLAB programs based on source transformation. More precisely the novel AD tool ADiMat uses a hybrid approach by combining source transformation and operator overloading.

The structure of this note is as follows. In Sect. 2, the basics of automatic differentiation are briefly summarized and some specifics of the MATLAB environment are pointed out. The overall design of ADiMat is presented in Sect. 3. Numerical experiments illustrating the code transformation and the computational efficiency of the resulting code are reported in Sect. 4.

2. Automatic Differentiation

The term "Automatic Differentiation (AD)" comprises a set of techniques for automatically augmenting a given computer program with statements for the computation of derivatives. That is, given a computer program C that com-

*This research is partially supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB 401 "Modulation of flow and fluid–structure interaction at airplane wings," Aachen University, Germany.

[1] MATLAB is a trademark of *The Mathworks, Inc.*

putes a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, automatic differentiation generates another "differentiated" program C' that, for any input, not only evaluates f but additionally evaluates its Jacobian matrix $J \in \mathbb{R}^{m \times n}$ at the same input. Sophisticated AD tools offer an additional degree of freedom by allowing to compute the product $J \cdot S$, where S is an arbitrary $n \times k$ matrix provided by the user. This concept is referred to as *seeding* and may be critical in terms of performance under certain circumstances. The AD technology is applicable whenever derivatives of functions given in the form of a programming language, such as Fortran, C, or C++, are required. The reader is referred to the recent book by Griewank [14] and the proceedings of AD workshops [3, 9, 15] for details on this technique. In automatic differentiation the chain rule of differential calculus is applied over and over again to elementary operations such as binary addition or multiplication for which derivatives are known, combining these step-wise derivatives to yield the derivatives of the whole program.

One well-known strategy for applying the chain rule is the so-called forward mode of AD. Program output variables whose derivatives are to be computed by AD are called *dependent variables*; program input variables with respect to which one is differentiating are known as *independent variables*. If one is interested in obtaining n directional derivatives of f, a derivative object u^∇ is associated with every intermediate variable u involved in the evaluation of the function f. Note that u is not necessarily assumed to be a scalar variable. If u is a compound object, its associated derivative object u^∇ needs to store n scalar derivative values for every scalar entry of u. For every operation of the original code C involving a variable u, an additional operation is performed in the differentiated code C'. For instance, let the original code C perform an operation of the form $u = v * w$. Then, the differentiated code C' is given by

$$\begin{aligned} u &= v * w \\ u^\nabla &= v^\nabla \odot w + v \odot w^\nabla, \end{aligned}$$

where the symbol $\odot$ denotes an appropriate multiplication of an object and a derivative object.

Except for very simple cases, the function f cannot be evaluated within a single routine. Instead, evaluating f typically involves a large subtree of the whole program, the "top-level" routine of this tree invoking a multitude of lower-level routines and finally providing the function value. In such a case automatic differentiation must be applied to the whole subtree, often totaling several hundreds of routines and tens or even hundreds of thousands lines of code, in order to obtain the function's derivatives.

Another AD strategy is the so-called reverse mode. In contrast to the forward mode propagating derivatives along with the control flow of the original function, the reverse (or backward) mode of AD generates derivatives by reversing the control flow of the underlying function. The reverse mode is attractive in terms of computation time when the number of dependent variables is significantly less than the number of independent variables [14].

Implementing AD for a certain programming language is usually done using either the source transformation approach or the operator overloading approach. The latter is applicable only in object oriented languages that support overloading of operators. Each operator executes the derivative statements and the original code. Usually, this approach has a myopic view regarding the data dependencies, because an operator has only access to its own operands. The operators have no knowledge about the dependencies between variables, and computing the dependencies at runtime is complicated. This limits the inherent flexibility provided by the associativity of the chain rule and the potential performance gains thereby possible [6]. Several AD tools available for various widely used programming languages implement the operator overloading approach without a complete dependency analysis (e.g., ADOL-C [16] and FADBAD [2] for C++, ADOL-F [24] for Fortran90, and ADMIT/ADMAT [8] for MATLAB).

By contrast, a tool implementing the source transformation approach analyzes the whole code and can identify those variables that need derivative objects and the statements that need derivative computations associated to them. The source code is then augmented, potentially using AD rules whose scope extends beyond an operation, and optimizations may be applied. The result is a new code which computes the original function and its derivatives. AD tools based on source transformation include Adifor [4, 12], TAF (formerly TAMC) [13], and Odysée [22]/Tapenade [26] for Fortran code and Adic [7] for C/C++.

2.1. MATLAB-specific issues in AD

The popularity of MATLAB is to a large degree due to its high-level data types and the corresponding powerful operators. These data types include vectors, matrices,—either real- or complex-valued—and cell arrays, as well as structs and classes. The type system of MATLAB is somewhat different from those implemented in commonly used programming languages, where the types and shapes of all variables are recognized at compile time. MATLAB code is usually run in the interpreter, which implements a runtime type system storing information about the type, number of dimensions, and sizes of each object. The number of dimensions and the sizes of each dimension is also called *shape*. Both the type and the shape of an object may change during the program execution. The compilers available for MATLAB solve these issues by providing a static compile-time and a dynamic run-time type system [10, 18].

Taking into account that the type and shape of a variable may change, the AD tool has to provide a way to store an associated derivative object and change its type and shape accordingly. MATLAB provides several ways to store a collection of objects whose type and shape are dynamic. One of them is the cell array data type. It provides a matrix-like structure, but each entry may have any type and shape.

MATLAB's high-level objects come with powerful operators, which are implemented by overloading the mathematical operators. The multiplication operator executes the desired operation depending on the shape of its operands. In addition to the usual operators like addition, multiplication, or exponentiation, more powerful ones are available. The backslash operator, for example, solves a system of linear equations $Ax = b$, which is expressed by `x = A\b`. Furthermore, all operators exist in a "pointwise" version; for instance the pointwise vector multiplication $u = v.*w$ produces a vector u whose components are equal to the product of the corresponding components in v and w.

Note that the MATLAB grammar is context-sensitive. For example, while `0.23 +4.5` evaluates to the scalar 4.73 when used in an expression, the same term yields a 1×2 vector $(0.23, 4.5)$ when occurring in the matrix definition `[0.23 +4.5]`; and both the transpose operator and the string delimiter use the single quote.

Another context-sensitive issue is the use of identifiers. For example, suppose that the identifier `foo` was not used yet. Then the use of it in the code fragment

```
if cond
  x = foo(1, 2);
else
  foo(1, 2) = 42;
end
```

is ambiguous. If `cond` is false, `foo` will be treated as a variable because it occurs on the left-hand side of the assignment in line 4. If `cond` is true, the identifier `foo` will be treated as a function name because it was not defined to be used as a variable yet. In a successive run with `cond` being false again, the identifier will be redeclared to be a variable. The above code admittedly is bad style but perfectly legal in MATLAB. These ambiguities in the grammar of the language complicate the design of the parser and the algorithm that determines the use of an identifier.

Similar problems occur when other dynamic languages, such as LISP, are parsed [1]. Therefore we have employed techniques similar to the ones described in that context.

The classification algorithm needs to know about the many functions that are either built into MATLAB directly or are supplied in one of its various so-called *toolboxes*. A toolbox is a collection of functions that are tailored for a specific application area. They also may be implemented in languages other than MATLAB. Both the functions built into MATLAB and those supplied in the toolboxes are referred to as *built-ins* in this note. In order to enable efficient transformation of MATLAB codes, we thus needed to develop a concept to handle this large number of functions.

3. ADiMat

The AD tool ADiMat uses a synthesis of source transformation and operator overloading techniques to enable the application of the forward mode of AD to a MATLAB program. Our hybrid approach capitalizes on the benefits of both techniques. The source transformation approach has the benefit of analyzing the whole code, and therefore the ability to apply transformation rules and optimizations exceeding the scope of a single operation. Implementing the computation of the derivatives in a class of its own leads to the benefit of type independence. An additional advantage is the increased readability of the differentiated code, because the derivative expressions are short and the loops iterating over the directional derivatives are hidden within the class. In addition optimized class implementations can be developed without affecting the overall tool design.

3.1. The source transformation tool

The implementation of the source code analysis is done using the GNU tools Bison [11] and Flex [21]. These tools were chosen because they are available on a wide range of platforms and provide the flexibility needed in our application. When a code file has been read its identifiers are sorted into the sets "variables", "functions", and "*unknowns*." The latter stores all identifiers which are not yet assignable to one of the two other sets. For example, the code fragment shown in Fig. 1 uses the functions `foo` and `sin` to create a vector containing samples of a wave, the number of points being influenced by the parameter `par`. The function `fft` implements a fast Fourier transformation and returns the coefficients.

```
wave = foo(@sin, (-par * pi):(par * pi));
   % use the function sin to create
   % a vector containing a wave
coeff = fft(wave);
   % compute the Fourier coefficients of the wave
```

Figure 1. MATLAB code example

Analyzing the fragment, the two identifiers `wave` and `coeff` have to be variables because they occur on the left-hand side of an assignment. All other identifiers are not declared yet and are assigned to the set of unknowns. An iterative algorithm tries to determine the use of the identifiers marked "unknown." This algorithm uses a list of sources comprising the files in the search path and the built-in database, where these identifiers might be defined. The algorithm stops if the set of unknowns or the list of sources is empty. A similar algorithm is used by LISP and described in [1].

In the example in Fig. 1 the identifiers `foo`, `sin`, `par`, `pi`, and `fft` are not defined yet. It is assumed here that a file `foo.m` exists in the current directory containing an appropriate function definition. The iterative algorithm scans the current directory, analyzes the file `foo.m` and moves the identifier `foo` to the set of functions because the identifier is used to denote a user-defined function.

An extendable database is provided with ADiMat, which enables the declaration of built-ins. The identifiers `sin`, `pi` and `fft` are declared in this database, and the iterative algorithm searches the database for identifiers marked "unknown." The information provided by the database indicates the use of the identifiers. In addition to the definition of the use of an identifier, derivative information is also provided.

For the example in Fig. 1 the algorithm would finally stop and report an error because the use of the identifier `par` was not defined. For the following discussion let us assume that the identifier `par` is a variable, which was defined earlier and has already been identified.

The next step is the variable dependency analysis. The dependencies of the variables are needed to identify *active* variables. Variables are called active if they occur on a path from the independent to the dependent variables. The dependency analysis performs a breadth-first traversal of the dependency graph. For the example in Fig. 1, assuming that one is interested in the derivative of `coeff` with respect to `par`, the dependency graph is shown in Fig. 2. An active variable is denoted by a box, an inactive variable by a circle. The edges show the dependencies. For example, `coeff` depends on `wave`, because in the last line of the example `wave` occurs on the right-hand side of an assignment to `coeff`.

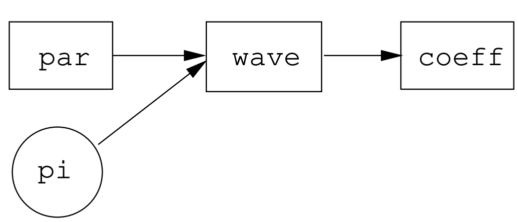

Figure 2. Dependency graph of the example from Fig. 1.

Every statement that contains active variables is called *active*. ADiMat currently supports only the so-called *strict* (or *single operation*) forward mode of AD. That is, every active statement which contains several operations is split up into several simpler statements that execute a single operation each. This *canonization* will enable the application of well known optimization strategies, e.g., common subexpression elimination [1], in future versions of our tool. We also plan the implementation of more sophisticated approaches, which do not need the splitting of the active statements.

All active statements within the code are differentiated with respect to all active variables they contain on the right-hand side. Operators, built-ins and user-defined functions are treated in different ways. The derivative expressions of the operators and the built-ins are constructed according to the well-known differentiation rules. The resulting new statement usually is inserted before the original statement. Calls of user-defined functions are modified 'in place.' The augmented user-defined function computes and returns its derivatives along with its results. Therefore the call has to be modified to provide the derivative objects of the active variables in the actual parameter list and to receive the derivative objects of the entries in the list of results. The augmentation of the program by adding new statements operates on abstract syntax trees (AST), thus enabling further optimizations at the AST level, e.g., eliminating subexpressions occurring in the original and in the differentiated code.

The last step of the AD tool is to write the augmented program and additionally selected graphs to the hard disk. The selectable graphs are a function dependency tree and the variable dependency graph. Both graphs may be of use to the experienced MATLAB programmer for optimizing the original code. They are available both as ASCII text and in a format suitable for the publicly available graph visualizer VCG [23].

The user may specify a directory to which the output files (augmented program and graphs) are written. This feature provides a rather simple form of configuration management, e.g., for the case of different applications requiring the same code to be differentiated with respect to different variables.

3.2. The derivative class

All derivative objects used in a program augmented by ADiMat belong to a new class provided together with the source transformation tool. The class comprises the full set of basic mathematical operators (e.g., addition, subtraction, multiplication), as well as access operators (e.g., assignment, indexing, concatenation). The derivative objects use the powerful cell array class of MATLAB to store directional derivatives. The advantage is that a cell array can store data of different type and shape, but provides a matrix-like access mechanism using indices. Before a derivative object can be used a constructor has to be called with the original object and the number of directional derivatives of interest, n, as parameters. The constructor generates n copies of the original object with all entries set to zero and stores the copies in the cell array. By contrast, the operator overloading AD tool ADMIT/ADMAT [8] provides the space for the derivatives by concatenating several instances of the original objects into a matrix potentially leading to excessive memory moves.

An easy-to-use function is available to perform the task of creating derivative objects: the code fragment

```
n = 1;
g_par = createZeroGradients(n, par);
g_par{1} = 1;
```

creates the derivative object `g_par` associated with `par`. Here we have assumed that only one directional derivative is sought. The last line of the code shows the process called seeding, where the directional derivative is initialized.

In contrast to operator overloading tools like ADMIT/ADMAT, the overloaded mathematical operators of ADiMat do not implement the differentiation rules for the addition, multiplication, etc., operators. Instead, they only provide the corresponding operations for the derivative objects. For example, an assignment `x = y * z` is transformed into the two assignments `g_x = g_y * z + y * g_z` and `x = y * z` for the derivative and the original value, respectively, where the overloaded multiplication operator performs the multiplication of a derivative object with another object. By contrast, in a purely operator overloading-based approach only the assignment `x = y * z` would appear in the differentiated code, and the overloaded multiplication operator would update the value `x` *and*—following the product rule—its derivative `g_x`.

3.3. Benefits of the combination

The combination of the source transformation and the operator overloading approach yields some significant benefits. The source transformation part of the tool analyzes the whole code, computes data dependencies, augments the code and applies possible optimizations. The development of the ADiMat tool started recently and so far emphasized the building of the needed tool infrastructure while implementing the most basic AD algorithm.

The major advantage of the presented approach is the ability to specify the built-ins using a simple macro language. While operator overloading approaches need to provide the derivative of a function in an additional file, the source transformation tool inserts the derivative code directly into the augmented program. For example, the derivative of the expression `y = sin(x)` is `g_y = ((g_x) .* cos(x))`, where the cosine function calls the code built into MATLAB and not an overloaded version residing in an additional file.

4. Example and performance comparison

To illustrate the output of ADiMat and its computational efficiency, we consider the following example, which determines the coefficients of the degree-m polynomial $p(x) = p_1 + p_2 x + p_3 x^2 + \dots p_m x^{m-1}$ that best fits n points (x_i, d_i) in the least squares sense: $\sum_{i=1}^{n}(p(x_i) - d_i)^2 =$ min with $n > m$. This leads to an over-determined linear system

$$V \cdot p = d\,, \tag{1}$$

where

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{m-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{m-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{m-1} \end{bmatrix}$$

denotes an $n \times m$ Vandermonde matrix, $p \in \mathbb{R}^m$, and $d \in \mathbb{R}^n$. This approach is known to be numerically doubtful, but presents a good example for the powerful MATLAB operators. For further discussions of the problem the reader is referred to [25].

In lines 11–14 of the function `fit` shown in Fig. 3, the matrix `V` is constructed from the abscissas contained in the argument `x`. Starting with the first column containing all ones, `V` is repeatedly expanded by appending additional columns (note that `x.^count` raises each component of the vector to the `count`-th power). After the composition of the matrix, Equation (1) is solved in line 16 using the backslash operator.

```
function p= fit(x, d, m)
% FIT -- Given x and d, fit() returns p
% such that norm(V*p-d) = min, where
% V = [1, x, x.^2, ... x.^(m-1)].

dim_x = size(x, 1);
if dim_x < m
   error('x must have at least m entries');
end

V = ones(dim_x, 1);     % line 11
for count = 1 : (m-1)
   V = [V, x.^count];
end                     % line 14

p = V \ d;              % line 16
```

Figure 3. Example function: Least squares fit of a polynomial.

To assess the sensitivity of the polynomial's coefficients, p, with respect to the abscissas, x, the function fit() is differentiated with respect to the argument x. ADiMat is applied to the code and the result is shown in Figure 4. The interface of the function in line 1 is modified to supply an input derivative object g_x associated with the variable x, and the list of results is extended to return the derivative g_p of p. Note that V becomes active because of line 13 of the original code given in Fig. 3. The corresponding derivative object g_V is set to zero in line 11 of Fig. 4 because the initialization of an active object by a constructor, such as in line 11 of Fig. 3, is treated as an assignment of a constant object. ADiMat provides a "constructor" that performs this task; see line 11 in Fig. 4. The assignments in the lines 14 and 15 show the statements for the exponentiation and its derivative. In line 16 the matrix–vector concatenation to update the derivative object of V is shown. The lines 20 and 21 finally solve the linear equation, where the original result is reused in line 21 to compute the derivative of the linear equation solver.

```
function [g_p, p]= g_fit(g_x, x, d, m)  % line 1
   % FIT -- Given x and d, fit() returns p
   % such that norm(V*p-d) = min, where
   % V = [1, x, x.^2, ... x.^(m-1)].

   dim_x = size(x, 1);
   if dim_x < m
      error('x must have at least m entries');
   end

   g_V = g_zeros(dim_x, 1);             % line 11
   V = ones(dim_x, 1);
   for count = 1 : (m-1)
      g_t0 = count .* x.^ (count- 1).* g_x;
      t0 = x.^ count;                   % line 15
      g_V = [g_V, g_t0];                % line 16
      V = [V, t0];
   end

   t1 = V \ d;                          % line 20
   g_p = V \ (-g_V * t1);               % line 21
   p = t1;
```

Figure 4. Differentiated example function.

In this example, the variable d has been assumed to be constant. If both V and d were active then the derivative expression for V \ d would be slightly different: g_p = V \ (g_d - g_V * t1).

Figures 5 and 6 show the performance of derivative computations with ADiMat's source transformation approach, with the operator overloading approach of ADMIT/ADMAT, and using divided differences. The horizontal axis of each chart shows the problem size, while the vertical axis shows the factor by which the additional derivative computations slow down the code. The timing results in both charts are computed on a PC (Intel Pentium IV™, 1.7GHz, 1GB RAM) running RedHat™ Linux 7.2 using MATLAB version 6 R12. The chart in Fig. 5 is a zoom into the chart in Fig. 6 for small problem sizes ($n = 10 : 100$ with step size 10).

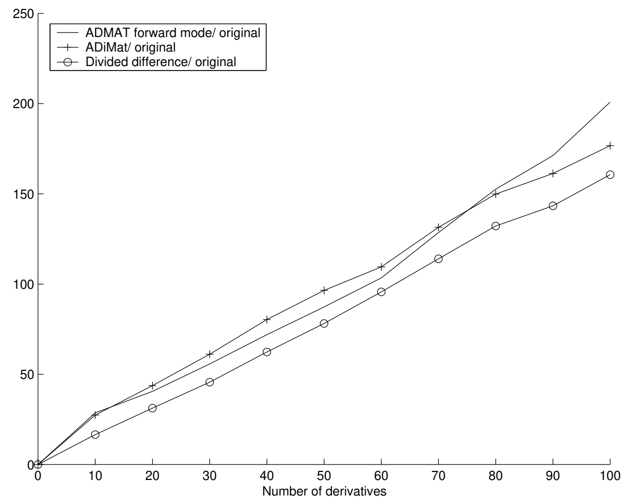

Figure 5. Timing ratio of programs computing derivatives and original program for small problem sizes ($m = 4$, $n = 10 : 100$).

The charts show that, for small problem sizes, the extra time needed to compute derivatives is nearly the same for both AD approaches. For larger problems, however, the source transformation approach is clearly superior. The quotient of the ADMIT/ADMAT tool seems to grow quadratically with the problem size, whereas the quotient of the ADiMat tool grows in a linear fashion. The operations both tools have to perform are approximately the same. Examining the implementation of both tools, the use of the cell array within the derivative class of ADiMat seems to be advantageous. The sub-indexing and concatenation operations for matrices implemented in MATLAB are reasonably fast. But using them excessively leads to a substantial performance loss. We believe that the implementation of the ADMIT/ADMAT toolbox suffers from these problems.

We also compared the forward mode of AD to the reverse mode, as provided in the ADMIT/ADMAT toolbox. We found that in all instances ADMIT/ADMAT's reverse mode was significantly slower than its forward mode.

Comparing the results with a divided differences approach to compute the derivatives shows that the divided differences are slightly faster for small problem sizes ($n = 10 : 200$). But we have no guarantee that the derivatives of the divided differences approach are accurate, whereas AD delivers exact derivatives up to machine precision [5, 14]. Thus, the slight performance penalty is a small price to pay

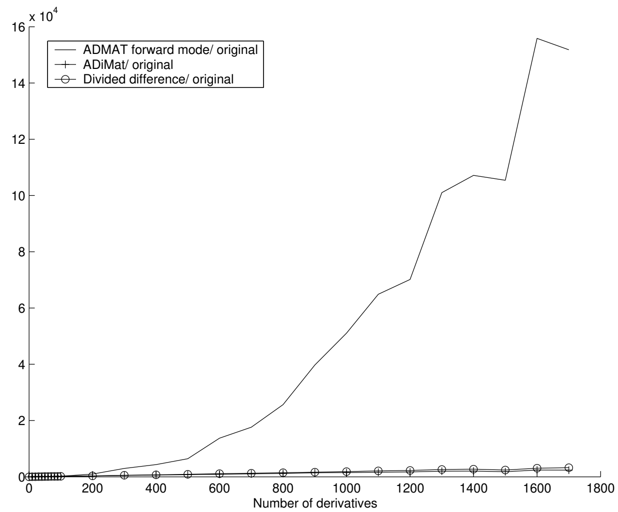

Figure 6. Timing ratio of programs computing derivatives and original program for larger problem sizes ($m = 4$, $n = 10 : 1700$).

to avoid potential numerical problems.

5. Concluding remarks

Many numerical simulations in a variety of scientific and engineering applications require sensitivity information for the computed results with respect to certain input parameters. This information can be obtained via partial derivatives. In technical computing, MATLAB is an extremely popular programming environment. In this paper a new tool called ADiMat is proposed. ADiMat transforms a MATLAB program such that the transformed program computes selected derivatives along with the results of the original program. This transformation is known as automatic differentiation providing derivative values accurate up to machine precision rather than approximations resulting from numerical differentiation. The crucial ingredient of ADiMat is the combination of AD implementation approaches based on operator overloading and source transformation. This hybrid approach enables ADiMat to cope with MATLAB's dynamic type system while allowing optimizations with a scope exceeding a single operation.

In its current version ADiMat supports the forward mode of automatic differentiation for programs written in MATLAB 6 (Release 12). Features that are currently not supported include complex numbers, classes, and private functions. First numerical experiments demonstrate that the code generated by ADiMat is competitive with the AD tool ADMIT/ADMAT, which is solely based on operator overloading. For larger numbers of directional derivatives the execution time of ADiMat-generated code is significantly less than the corresponding code executed by ADMIT/ADMAT.

Future research directions include optimization of the differentiated code, such as common subexpression elimination, vectorization and preallocation [20]. We also plan to integrate additional functionality by implementing further AD strategies like the statement–level reverse mode for the forward mode as implemented, for instance, in Adifor, as well as the global reverse mode.

A further issue of future research is the simplification of the computation of higher-order derivatives. Computing higher-order derivatives is currently possible, but not very intuitive, and needs a certain amount of knowledge about the differentiation process performed by ADiMat.

Some of the central concepts underlying the ADiMat tool, such as the iterative procedure for classifying the identifiers, the global dependency analysis, and the way operator overloading and source transformation are combined, carry over to other interpreted languages like Python. In other aspects the design was specifically tailored to MATLAB, one example being the use of cell arrays instead of ordinary arrays to hold the derivative objects. These decisions were made for performance as well as elegance and will have to be reconsidered when implementing AD for other languages.

Acknowledgment

The authors want to thank the unknown referees for their valuable comments, which helped to improve the presentation.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[2] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM-REP-1996-17, Technical University of Denmark, IMM, Departement of Mathematical Modeling, Lyngby, 1996.

[3] M. Berz, C. Bischof, G. Corliss, and A. Griewank. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, PA, 1996.

[4] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[5] C. H. Bischof, H. M. Bücker, and B. Lang. Automatic differentiation for computational finance. In E. J. Kontoghiorghes, editor, *Computational Methods in Decision-Making, Economics and Finance*, Boston, 2002. Kluwer Academic Publishers. To appear.

[6] C. H. Bischof and M. R. Haghighat. Hierarchical approaches to automatic differentiation. In Berz et al. [3], pages 83–94.

[7] C. H. Bischof, L. Roh, and A. Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software–Practice and Experience*, 27(12):1427–1456, 1997.

[8] T. F. Coleman and A. Verma. ADMIT-1: Automatic differentiation and MATLAB interface toolbox. *ACM Trans. Math. Softw.*, 26(1):150–175, 2000.

[9] G. Corliss, A. Griewank, C. Faure, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation 2000: From Simulation to Optimization*. Springer, New York, NY, 2002.

[10] L. De Rose, K. Gallivan, E. Gallopoulos, and B. Marsolf. FALCON: A MATLAB interactive restructuring compiler. In C.-H. Huang et al., editors, *Languages and compilers for parallel computing: 8th international workshop, LCPC 95, Columbus, Ohio, USA, August 10–12, 1995: Proceedings*, volume 1033 of *Lecture Notes in Computer Science*, pages 269–288, Berlin, Germany, 1996. Springer-Verlag.

[11] C. Donnelly and R. Stallman. *Bison, the YACC-compatible parser generator*. Free Software Foundation, 1999.

[12] M. Fagan and A. Carle. Adifor 3.0 overview. Technical Report CAAM–TR00–03, Rice University, Department of Computational and Applied Mathematics, 2000.

[13] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software*, 24(4):437–474, Dec. 1998.

[14] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, Philadelphia, PA, 2000.

[15] A. Griewank and G. Corliss. *Automatic Differentiation of Algorithms*. SIAM, Philadelphia, PA, 1991.

[16] A. Griewank, D. Juedes, and J. Utke. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, 22(2):131–167, 1996.

[17] D. R. Hill and L. C. Rich. Automatic differentiation in MATLAB. *Applied Numerical Mathematics*, 9:33–43, 1992.

[18] The Mathworks, Inc., Natick, MA. *Matlab Compiler—The Language of Technical Computing, User's Guide, Version 2*, 2001.

[19] The Mathworks, Inc., Natick, MA. *Symbolic Math Toolbox for use with Matlab, User's Guide, Version 2*, 2001.

[20] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 53–66, Berkeley, CA, Oct. 3–5 1999. USENIX Association.

[21] V. Paxson. *Flex, version 2.5, A fast scanner generator*. University of California, 1990.

[22] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in Odyssée. *Tellus*, 45A(5), 1993.

[23] G. Sander. Graph Layout through the VCG Tool. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing, DIMACS International Workshop GD'94*, Lecture Notes in Computer Science, Vol. 894, pages 194–205, Berlin, 1995. Springer.

[24] D. Shiriaev. ADOL–F automatic differentiation of Fortran codes. In Berz et al. [3], pages 375–384.

[25] C. F. Van Loan. *Introduction to Scientific Computing: A Matrix–Vector Approach Using MATLAB*. The MATLAB curriculum series. Prentice-Hall, Upper Saddle River, NJ, 2000.

[26] The TAPENADE tutorial. `http://www-sop.inria.fr/tropics/`.