

A System to Define Derivatives for a Large Number of Functions Built into MATLAB

Martin Bücker, Andre Vehreschild
[buecker,vehreschild]@sc.rwth-aachen.de
Institute for Scientific Computing
RWTH Aachen University
D-52056 Aachen
Germany

1 Motivation

Most MATLAB projects extensively use the numerous library functions—also called toolbox functions—or functions built into the language’s core. The source code of these functions is usually not available or is formulated in a different language for which an appropriate AD-tool might not be available. MATLAB extensively resorts to LAPACK [1], BLAS [3, 4, 7], ARPACK [8], and many other libraries to implement its toolbox and built-in functions.

Several approaches are feasible to formulate derivatives for toolbox functions that are implemented in different languages or for which only binary versions are available:

1. Use of a language-specific AD-tool. This approach is applicable only, if the source code as well as an AD-tool for the implementation language of the function are available. In many cases one of the two conditions fails so that this approach might not be applicable.
2. Re-implementation of the functionality in MATLAB and use of an AD-tool for MATLAB. This approach only applies to small functions because the effort to re-implement large functions may be significant and the probability of errors increases with the size and complexity of the function. The advantage of this approach is that it avoids any problems concerned with mixed language programming.
3. Implement the computation of the derivative of a function by hand. That is, for each function whose derivatives are needed, a file has to be created which solely implements the derivative computation. Several challenges have to be mastered here: (i) most parts of the computation of the original function are repeated in the derivative function because of the need for temporary values; (ii) an additional file has to be maintained for each toolbox function that is to be differentiated; (iii) the file has to be distributed with each project in which the derivative is needed; (iv) a derivative has to be implemented by hand. The latter requires the most human effort and is also error-prone. Furthermore, many of the toolbox functions in MATLAB are “polymorph” in the way that they can be parameterized executing similar, yet different algorithms. For instance, the `norm()`-function computes different vector and matrix norms, depending on its second argument.
4. Insert the derivative directly into the code using high-level functions (possibly using the original function [5]). This is a powerful, but sophisticated approach. The AD-tool

has to identify the correct version of the function called, depending on the number of arguments and results, the types and dimensions of the inputs and optionally some flags used in the call. The aim of the present study is to sketch possible solutions to this problem.

The number of functions in the basic installation of MATLAB is around 1,200 including such functions as `help` for which no derivative code is to be generated. However, the number of functions requiring derivatives is still around 300. Furthermore, many of these functions implement a whole class of functions. Recall that the `norm()`-function implements the Euclidean, the infinity, the Frobenius, and some other norms, depending on its second argument. All of these calls have distinct derivative terms.

2 The macro language of the ADiMat system

The source-to-source transformation AD-tool ADiMat [2] generates derivative code for MATLAB programs and provides a way to specify the derivative of toolbox functions by providing a macro language. In this section, the main part of the macro language is defined and an illustrating example is given.

A derivative of a function is given by a statement which satisfies a pseudo grammar formulated in the EBNF version used in [6]. Terminals representing numbers or identifiers are printed in bold as in **NUM**. Terminals are enclosed in double quotes, if they have to appear in the input in exactly this way. Non-terminals are given in lower case. The following pseudo grammar is a simplified fragment of the overall grammar:

```

stat := "BMFUNC" signature "DIFFSTO" derivative
signature := result "=" ID "(" (paramlist)? ")"
result := "$$"
paramlist := parameter ("," parameter)*
parameter := (( "$" NUM (":" TYPE)? ("=" (NUM | ID)))? | NUM | ID)
derivative := {any valid MATLAB expression with $NUM tokens in it}

```

A statement recognized by this grammar using the starting symbol `stat` associates a derivative with a function described by its signature. The signature represents a MATLAB function returning exactly one result and accepting an arbitrary number (including zero) of arguments. The result is denoted by the token `$$`. Parameters occur in three alternatives. The first option is a so-called positional parameter which is given by a `$NUM` expressions denoting the position of the parameter in the list. A positional parameter may optionally have a type or a default value. A parameter may also be a number or an identifier. This specializes the function so that the call has to be done with exactly this constant parameter to use the given derivative.

The `norm()`-function is taken as an example which is recognized by the language defined by this grammar. The `norm()`-function can be used to compute several vector and matrix norms. Given a vector v of dimension n , the arguments of the `norm()`-function may be used to compute the Euclidean vector norm $\|v\|_2$, the p -vector-norm

$$\|v\|_p = \left(\sum_{i=1}^n |v_i|^p \right)^{1/p},$$

or the Frobenius norm $\|M\|_F$ of a matrix M . The derivative of the p -vector-norm is given by

$$\|v\|_p^{1-p} \sum_{i=1}^n \left(|v_i|^{p-1} \frac{\partial}{\partial v_i} |v_i| \right). \quad (1)$$

The first part of translating this derivative to the macro language consists of the signature of the function given by `$$=norm($1:vector, $2)`. This signature introduces to ADiMat the function `norm` using exactly two parameters, where the first one has to be a vector. The second part of the derivative definition is the MATLAB implementation of (1), where the vector v is replaced by the placeholder `$1` and the scalar p is replaced by `$2`. Furthermore, the derivative of `$1` is denoted by `$@1` and the mapper function `call()` of ADiMat is used. The mapper function applies the function given as its first parameter to all directional derivatives of the derivative object given in its second parameter. As an example, consider the expression `call(@sum, g_x)` where `g_x` is a derivative object with a certain number of directional derivatives, say `n`. Then, the mapper function executes the following pseudo MATLAB code:

```
for c=1:n
    g_r{c}= sum(g_x{c});
end
```

Now, using the macro language, the association of the p -vector-norm with its derivative is given by:

```
BMFUNC $$=norm($1:vector, $2) DIFFSTO (($$.^(1-$2)*
    call(@sum, abs($1).^($2-1)*g_abs($@1, $1)))
```

Here, `g_abs()` computes the derivative of the `abs()`-function. The signature can be extended to compute the Euclidean norm if the second parameter is omitted to mimic the default behaviour of the MATLAB function. The extended signature is given by `$$=norm($1:vector, $2=2)`, where the second parameter gets the default value of 2.

In an analogous way, the Frobenius norm for matrices, characterized by the string `'fro'` as the second parameter, can be associated with its derivative as follows:

```
BMFUNC $$=norm($1:matrix, 'fro') DIFFSTO
    (call(@sum, call(@diag, $@1'*$1+$1'*$@1))./(2*$$))
```

The decision which derivative code is inserted is made by examining the actual static call of the function. Consider a function call of `norm(v,3)`, where `v` is assumed to be vector. This call is analyzed and the list of all derivative definitions for the identifier `norm` is fetched from the database storing the derivative definitions. The list is filtered to match the actual call with respect to the number of parameters. That is, in the example of `norm(v,3)` only entries with two parameters remain in the list. The filter is actually more complicated, because it treats parameters having a default value attached differently. After that, the types of the arguments and the ones specified in the parameter list are compared and only the exact matching types or more generic types remain in the list. Finally, constants are analyzed and taken into account, to select the correct derivative. A constant in the function call has to exactly match the one specified in the derivative definition. If none matches, then the next more generic derivative definition is checked for compliance, until one signature satisfies the function call or the list is empty. The latter case is considered to be an error.

3 Concluding remarks

The macro language used by ADiMat enables the definition of derivatives of toolbox functions. It may also be used for user-defined functions, to efficiently specify their derivative. The latter will be shown more precisely in the presentation or paper. Using the macro language, derivatives can be given tailored exactly to the call of the function to be differentiated. Hence, there is no need for additional files. The code is integrated into the differentiated function. Furthermore the macro language enables the specification of derivatives with respect to flags or constants given in the function call. This yields a performance increase in comparison with a more general derivative code, which had to be used otherwise. The systems is currently migrated from the old built-in system of ADiMat, which uses lists to store the derivative definitions, to a database which cares about the data management.

References

- [1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.
- [2] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 65–72, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
- [3] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990.
- [4] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.
- [5] R. Giering, T. Kaminski, and T. Slawig. Generating efficient derivative code with TAF: Adjoint and tangent linear Euler flow around an airfoil. 2003.
- [6] D. Grune, H. E. Bal, C. Jacobs, K. G. Langendoen, Koen Langendoen, and Henri Bal. *Modern Compiler Design*. John Wiley & Sons, Inc., 2000.
- [7] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.
- [8] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK USERS GUIDE: Solution of Large-Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, 1998.