Algorithms and Design for a Second-Order Automatic Differentation Module

Jason Abate
Christian Bischof
Lucas Roh
Alan Carle

CRPC-TR97691-S
May 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

submitted May 1997

Algorithms and Design for a Second-Order Automatic Differentiation Module* (to be presented at ISSAC 97)

Jason Abate†
Texas Institute for Computational Mathematics
University of Texas at Austin
abate@ticam.utexas.edu

Christian Bischof and Lucas Roh
Mathematics and Computer Science Division
Argonne National Laboratory
{bischof,roh}@mcs.anl.gov

Alan Carle
Center for Research on Parallel Computation
Rice University
carle@cs.rice.edu

CRPC-TR97691-S
May 1997

Abstract

This paper describes approaches to computing second-order derivatives with automatic differentiation (AD) based on the forward mode and the propagation of univariate Taylor series. Performance results are given which show the speedup possible with these techniques. We also describe a new source transformation AD module for computing second-order derivatives of C and Fortran codes and the underlying infrastructure used to create a language-independent translation tool.

1 Introduction

Automatic differentiation provides an efficient and accurate method to obtain derivatives for use in sensitivity analysis, parameter identification and optimization. Current tools are primarily targeted at computing first derivatives, i.e. gradients and Jacobians. Prior to AD, first derivative values were obtained through divided difference methods, symbolic manipulation or hand-coding, all of which have drawbacks when compared to AD. Accurate second-order derivatives are even harder to obtain; it is possible to end up with no accurate digits in the derivative value when using a divided difference scheme.

It is possible to repeatedly apply first derivative tools to obtain higher order derivatives, but this approach is complicated and ignores structural information about higher-order derivatives such as symmetry. Additionally, in cases where a full Hessian, H, is not required, such as with Hessian-vector products ($H \cdot v$ for a vector v), and projected Hessians ($v^T \cdot H \cdot v$ for a matrix v with many fewer columns than rows), it is possible to compute the desired values much more efficiently than with the repeated differentiation approach.

There is no "best" approach to computing Hessians – the most efficient approach to computing second-order derivatives depends on the specifics of the code and, to a lesser extent, the target platform on which the code will be run [4, 8]. In all cases, however, derivative values are computed to machine precision, without the roundoff errors inherent in divided difference techniques.

AD via source transformation provides great flexibility in implementing sophisticated algorithms which exploit the associativity of the chain rule of calculus (see [6] for a discussion). Unfortunately, the development of robust source transformation tools is a substantial effort. ADIFOR and ADIC, source transformation tools for Fortran and C, both implement relatively simple algorithms for propagating derivatives. Most of the development time so far has concentrated on producing tools that handle the full range of the language, rather than developing more efficient algorithms to propagate derivatives.

To make it easier to experiment with algorithmic tech-

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, and Cooperative Agreement No. NCC 1 21, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

†This work was partially performed while the author was a research associate at Argonne National Laboratory.

niques, we have developed AIF, the Automatic Differentiation Intermediate Form. AIF acts as the glue layer between a language-specific front-end and a largely language-independent transformation module that implements AD transformations at a high level of abstraction.

We have implemented an AIF-based module for computing second-order derivatives. The Hessian module, as we call it, implements several different algorithms and employs them in a fashion that is determined by the code presented to it. However, this context-sensitive logic, which is based on machine-specific performance models, is transparent to the AD front-end. The Hessian module currently interfaces with ADIFOR and ADIC. Experimental results show that the resulting codes outperform the recursive application of first-order tools by a factor of two when computing full, dense Hessians and are able to compute full, sparse Hessians and partial Hessians at significantly reduced expense.

Section 2 outlines the two derivative propagation strategies that we have explored for Hessians, including cost estimates for computing various types of Hessians. Section 3 shows the performance of the various approaches for a sample code, and Section 4 describes the infrastructure that was used to develop the Hessian augmentation tool. Lastly, we summarize our results.

2 Strategies for Computing Second Derivatives

2.1 Forward Mode Strategies

The standard forward mode of automatic differentiation can easily be expanded to second order to compute Hessians. For $z = f(x, y)$, we can compute ∇z and $\nabla^2 z$, the gradient and Hessian of z, as

$$\nabla z = \frac{\partial z}{\partial x}\nabla x + \frac{\partial z}{\partial y}\nabla y \tag{1}$$

$$\begin{aligned}\nabla^2 z = {} & \frac{\partial z}{\partial x}\nabla^2 x + \frac{\partial z}{\partial y}\nabla^2 y \\ & + \frac{\partial^2 z}{\partial x^2}(\nabla x \cdot \nabla x^T) + \frac{\partial^2 z}{\partial y^2}(\nabla y \cdot \nabla y^T) \\ & + \frac{\partial^2 z}{\partial x \partial y}(\nabla x \cdot \nabla y^T + \nabla y \cdot \nabla x^T)\end{aligned} \tag{2}$$

This approach is conceptually simple and produces efficient results for small numbers of independent variables. For n independent variables, gradients are stored in arrays of length n and Hessians, due to their symmetric nature, are stored using the LAPACK [1] packed symmetric scheme, which reduces the storage requirements from n^2 to $\frac{1}{2}n(n+1)$. The cost of computing a full Hessian using the forward mode is $\mathcal{O}(n^2)$.

Many algorithms do not need full knowledge of the Hessian, but require only a Hessian-vector product, $H \cdot v$, or a projected Hessian, $v^T \cdot H \cdot w$, where v and w are matrices with n_v and n_w columns, respectively. Rather than computing the full Hessian at a cost of $\mathcal{O}(n^2)$ followed by one or two matrix multiplications, we can multiply Equation (2) on the left and/or right by v^T and w to produce new propagation rules. By modifying the derivative objects that get propagated, the required computations can be performed at a much lower cost. These costs are summarized in Table 1. In the case of large Hessians and relatively small values of n_v or n_w, the saving can be significant. Additionally, the coloring techniques which have been applied to structured Jacobians [2] can be applied to Hessians for a significant savings.

Hessian Type	Cost
$\nabla^2 f$	$\mathcal{O}(n^2)$
$\nabla^2 f \cdot v$	$\mathcal{O}(nn_v)$
$v^T \cdot \nabla^2 f \cdot v$	$\mathcal{O}(n_v^2)$
$v^T \cdot \nabla^2 f \cdot w$	$\mathcal{O}(n_v n_w)$

Table 1: Summary of Hessian costs. n is the number of independent variables, n_v and n_w are the number of columns of v and w, respectively.

2.2 Taylor Series Strategies

As an alternative to the forward mode propagation of gradients and Hessians, we can propagate two-term univariate Taylor series expansions about each of the non-zero directions in the Hessian [4]. To compute derivatives at a point x_o in the direction u, we consider f as a scalar function $f(x_o + tu)$ of t. Its Taylor series, up to second order, is

$$\begin{aligned} f(x_o + tu) &\approx f(x_o) + \left.\frac{\partial f}{\partial t}\right|_{t=0} \cdot t + \frac{1}{2}\left.\frac{\partial^2 f}{\partial t^2}\right|_{t=0} \cdot t^2 \\ &= f + f_t t + f_{tt} t^2 \end{aligned} \tag{3}$$

where f_t and f_{tt} are the first and second Taylor coefficients. The uniqueness of the Taylor series implies that for $u = \widehat{e}_i$ we obtain

$$f_t = \left.\frac{\partial f}{\partial x_i}\right|_{x = x_o} \tag{4}$$

$$f_{tt} = \frac{1}{2}\left.\frac{\partial^2 f}{\partial x_i^2}\right|_{x = x_o}. \tag{5}$$

That is, we computed the ith diagonal element in the Hessian. Similarly, to compute the (i, j) off-diagonal entry in the Hessian, we set $u = \widehat{e}_i + \widehat{e}_j$. The uniqueness of Taylor expansion implies

$$f_t = \left.\frac{\partial f}{\partial x_i}\right|_{x = x_o} + \left.\frac{\partial f}{\partial x_j}\right|_{x = x_o} \tag{6}$$

$$f_{tt} = \frac{1}{2} \cdot \frac{\partial^2 f}{\partial x_i^2} + \frac{\partial^2 f}{\partial x_i x_j} + \frac{1}{2} \cdot \frac{\partial^2 f}{\partial x_i^2} \tag{7}$$

If Taylor expansions are also computed for the i and j diagonal elements, the off-diagonal Hessian entries can be recovered by interpolation. As with the forward mode, simple rules specify the propagation of the expansions for all arithmetic and intrinsic operators [10, 13].

The Taylor series approach can compute any set of Hessian entries without computing the entire Hessian. This technique is ideal for sparse Hessians when the sparsity pattern is known in advance and for situations where only certain elements (such as the diagonal entries) are desired. Additionally, each Taylor series expansion is independent. This allows very large Hessians, which can easily overwhelm the available memory, to be computed in a stripmined fashion by by partitioning the expansion directions and computing them independently with multiple sweeps through the code in a fashion that is similar to the stripmining technique described in [5].

2.3 Preaccumulation

The associativity of the chain rule allows derivative propagation to be performed at arbitrary levels of abstraction. At the simplest, the forward mode works at the scope of a single binary operation. By expanding the scope to a higher level, such as an assignment statement, a loop body or a subroutine, it is possible to decrease the amount of work necessary to propagate derivatives, as shown in [7, 9].

The preaccumulation technique computes the gradient and Hessian of the variable on the left side of the assignment statement in two steps. Assume that for the statement $z = f(x_1, x_2, \ldots x_N)$, we have ∇x_i and $\nabla^2 x_i, i = 1, \cdots N$, the global gradient and Hessian of x_i and that we wish to compute the global gradient and Hessian of z, ∇z and $\nabla^2 z$.

Step 1: Preaccumulation of local derivatives
The variables on the right side of the statement are considered to be independent and we compute "local" derivative objects, $\frac{\partial z}{\partial x_i}$ and $\frac{\partial^2 z}{\partial x_i \partial x_j}, i, j = 1, \ldots N$, with respect to the right-hand side variables. This can be done using either the forward or Taylor series mode.

Step 2: Accumulation of global derivatives
We accumulate the global gradient and Hessian of z. When using the forward mode for global propagation of derivatives, this is done as follows:

$$\nabla z = \sum_{i=1}^{N} \frac{\partial z}{\partial x_i} \nabla x_i$$

$$\begin{aligned} \nabla^2 z &= \sum_{i=1}^{N} \frac{\partial z}{\partial x_i} \nabla^2 x_i + \sum_{i=1}^{N} \frac{\partial^2 z}{\partial x_i^2} (\nabla x_i \cdot \nabla x_i^T) \\ &+ \sum_{i=1}^{N} \sum_{j=i+1}^{N} \frac{\partial^2 z}{\partial x_i \partial x_j} (\nabla x_i \cdot \nabla x_j^T + \nabla x_j \cdot \nabla x_i^T) \end{aligned}$$

The rules for Taylor series expansions can be generalized in a similar fashion.

Gradient codes produced by ADIFOR and ADIC currently employ statement-level preaccumulation for all assignment statements more complicated than a single binary operation. Experiments with similar "global" preaccumulation strategies for computing Hessians have produced inconsistent results across various codes and machines. No global strategy outperformed all other strategies on all test codes and all machines.

Thus, we have developed an adaptive strategy where the costs of using and not using statement level preaccumulation are computed and compared when the derivative code is generated. These costs are estimated based on machine-specific performance models of the actual propagation code. Thus the Hessian module decides which strategy to use based on the structure of a particular computation. We believe that such context-sensitive strategies to be crucial for future improvement of AD tools.

3 Hessian Performance on a CFD Code

Hessian code was generated for a steady shock tracking code provided by Greg Shubin, Boeing Computer Services, Seattle, Washington [14]. Due to memory constraints, a 20×20 section of the full, 190×190 Hessian was computed for each of the 190 dependent variables. The section of the Hessian being studied exhibits some sparsity, with 72 nonzero entries on or above the diagonal.

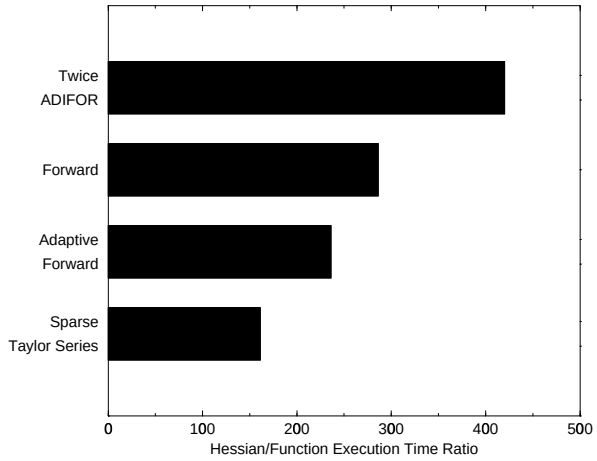

Figure 1: Hessian performance of the Shubin Hessian test code with 20 independent variables.

Hessian codes were generated using four different strategies. Figure 1 shows the ratio of the Hessian computation time to the function computation time while Figure 2 shows the memory requirements of the augmented Hessian codes on a Sun UltraSparc 1. The original code required 8.0×10^{-4} seconds of execution time and used 360 KB. The first strategy, labeled "Twice ADIFOR", was generated by first producing a gradient code with ADIFOR 2.0, and then running the gradient code through ADIFOR again. The "Forward" case implements the forward mode on a binary operation level. The "Adaptive Forward" code uses the forward mode, with preaccumulation at a statement level where appropriate. The "Sparse Taylor Series" mode uses the Taylor series mode to compute just the entries which are known to be zero.

Clearly, the "Twice ADIFOR" scheme can be easily beaten by exploiting the symmetry of the Hessian, both in terms of execution speed and memory usage, which is done in both the "Forward" and "Adaptive Forward" codes. This also shows that the use of an adaptive preaccumulation strategy can outperform the operation-level forward mode. Further improvements in the strategy used to decide when to use preaccumulation should further increase the efficiency of the adaptive scheme. Finally, the "Sparse Taylor Series" code shows that, if the sparsity structure of a problem is known, it can be exploited for additional savings.

4 Language and Tool Independence with AIF

The algorithms of automatic differentiation are, for the most part, independent of the language to which they are applied. For example, the Fortran assignment statement

```
z = 2.0 + x * y
```

and the more complicated C assignment statement

```
foo->struct.z = 2.0 + bar->x * q[c]
```

both can be abstracted to

$$variable_1 = constant + variable_2 * variable_3$$

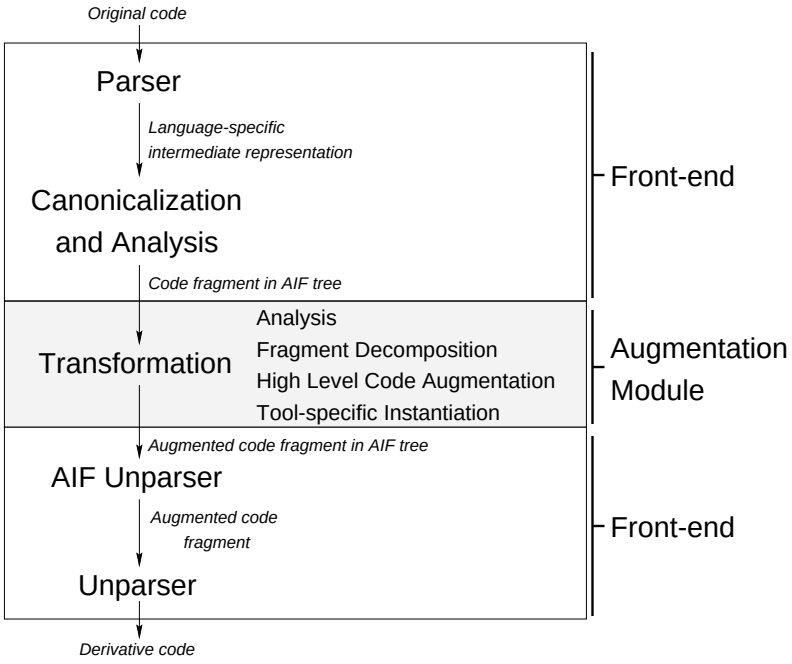

Figure 3: AIF process

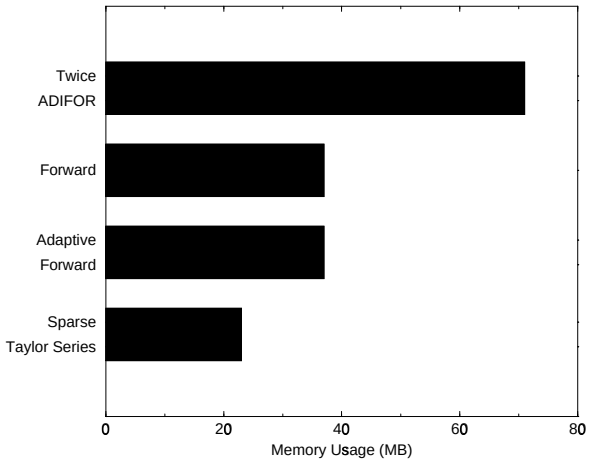

Figure 2: Hessian memory usage of the Shubin Hessian test code with 20 independent variables. The original code used 360 KB.

when thinking about the propagation of derivatives.

This simplicity should be reflected in the AD augmentation modules and in an attempt to simplify the development of new AD algorithms, we have developed the Automatic Differentiation Intermediate Form (AIF). AIF tries to capitalize on the work that has been done in producing robust language front-ends for automatic differentiation and to simplify AD development by insulating developers from the specifics of the underlying language. Thus, AIF aims at providing a framework for experimenting with more advanced AD augmentation algorithms and to speed the development of robust tools which implement these advanced algorithms.

A source-transformation approach to AD is illustrated in Figure 3. This is an idealized representation; not all stages are included in all tools. The first few stages are handled by the language-specific front-end. First, the original source code is parsed. During the canonicalization and analysis phase, the code transformed to a semantically equivalent form which is more appropriate for AD and high level information is gathered, such as a determination of which variables need associated derivative objects (the so-called active variables). For details on the canonicalization and analysis phase in ADIC and ADIFOR, see [6, 3]. The front-end then collects code fragments, which may range in size from single assignment statements to entire subroutines, and passes them to the augmentation module. Figure 4 shows the AIF tree corresponding to the sample Fortran statement above. The first line in each node is the node type. The second line, if present, contains an attribute, which could be a variable name, constant value or subroutine name.

In addition to the AIF trees, the front-end also passes a set of bindings to specify global information. This includes information about the desired augmentation strategy and the maximum number of independent variables.

The augmentation module then modifies the tree to propagate derivative values. The SORCERER tree parser generator [11, 12] is used to analyze and modify these trees, along with a set of utility routines provided in the AIF library which assist the augmentation process. For the Hessian module, the augmentation process includes:

Analysis:
Each assignment statement is analyzed to gather information such as the sparsity of the local Hessian and the number of variables on the right side of the statement. This information is then used to estimate the cost of alternative approaches to computing Hessians, and to select the least expensive strategy.

Fragment Decomposition:
Each statement is broken down into a sequence of unary and binary operations. Temporary variables are

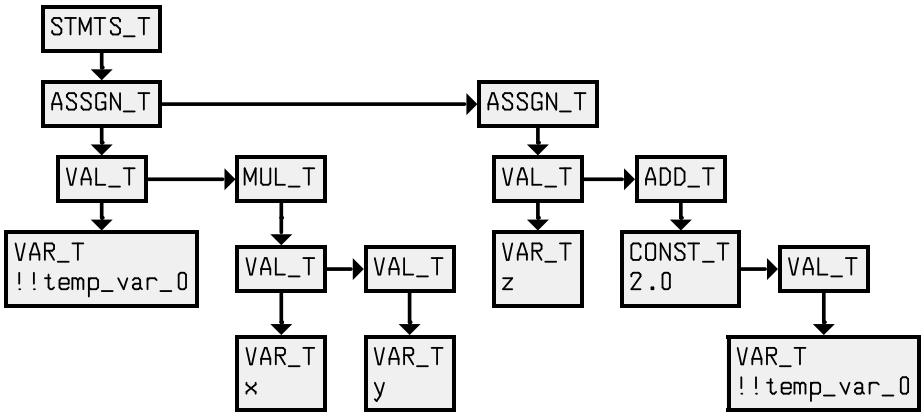

Figure 5: Sample tree after breakup into binary operations. The `!!temp_var_0` node is a request for a temporary variable.

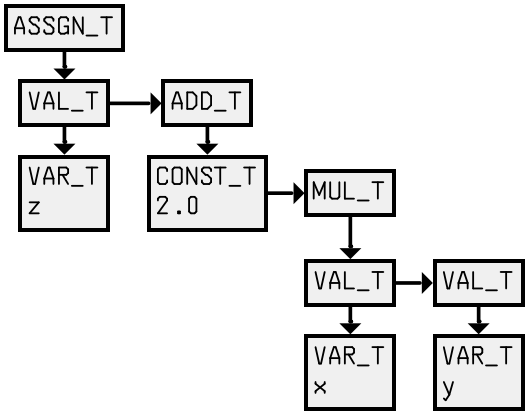

Figure 4: AIF tree produced by the ADIFOR front-end.

requested as necessary. Figure 5 shows the tree after breakup into binary operations. The `!!temp_var_0` node is a request for a temporary variable. The front-end will later instantiate this request by generating and declaring a suitably typed temporary variable.

High level Code Augmentation:
The tree is augmented with templates which specify the high level algorithmic operations to be performed, such as "initialize a local gradient" or "handle a multiply with two active variables." At this level, no assumptions are made of what the output code will look like. Figure 6 shows the tree after the high level augmentation. The `HESS_FOR_MUL_A_A_T` node is a template for the forward mode multiplication of two active variables. Similarly, `HESS_FOR_ADD_I_A_T` is the addition of an inactive value (the `CONST_T` node) and an active variable. In this example, the high level version is not much different than in the previous step. For more complicated augmentation algorithms, which involve initialization, interpolation and accumulation of local derivative objects, however, even this high-level representation can become quite involved.

Tool-specific Instantiation:
The templates added in the previous step are expanded into actual AIF trees. Multiple varieties of templates can be written to produce, for example, calls to a subroutine library or inlined code, or to account for back-end peculiarities. Figure 7 shows part of the instantiated tree derived from the one in Figure 6. The `CALL_T` nodes represent subroutine calls to the routines listed in the `NAME_T` node. All of the nodes attached to the `LIST_T` node are arguments to the subroutine call.

The augmentation module then returns the augmented code fragments to the front-end in AIF trees. It also passes a set of return bindings which specify, for example, the shape and size of derivative objects to be associated with active variables and the type of temporary variables.

The front-end receives the augmented tree and bindings from the augmentation module and converts them from AIF to its native representation and glues them back in their appropriate place. It also declares all of the requested temporary variables and derivative objects and handles the association of active variables and their associated derivative objects. Figure 8 shows the Fortran produced by ADIFOR for the sample statement. Not shown are the pieces of code which declare the temporary variables and derivative objects (`r_var0`, `adg_r_var0` and `adh_r_var0`) and the derivative objects associated with `x`, `y` and `z`, such as `adg_x` and `adh_x`.

5 Future Work

The current Hessian tool is our first attempt at producing an AD source transformation module in the AIF environment. We plan to continue this work in three areas.

1. In the area of algorithms, we plan to implement the preaccumulation of univariate Taylor series vectors in hopes of achieving similar speedup as with the forward mode. We also plan extensions of the Taylor series mode to arbitrary higher order derivatives.

2. We will refine the timing models used to characterize the performance of the Hessian codes on a particular machine. This will help to better determine under what conditions it is beneficial to use preaccumulation, which should further improve the performance of the adaptive mode strategy. This will also be useful with preaccumulation over larger sections of code which will be supported in the future.

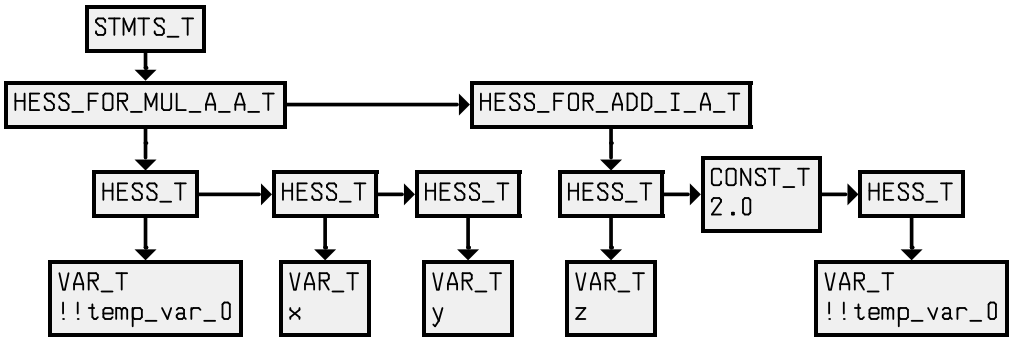

Figure 6: Sample tree after augmentation with high level templates. The **HESS_FOR_MUL_A_A_T** and **HESS_FOR_ADD_I_A_T** are templates representing forward mode multiplication and addition operations.

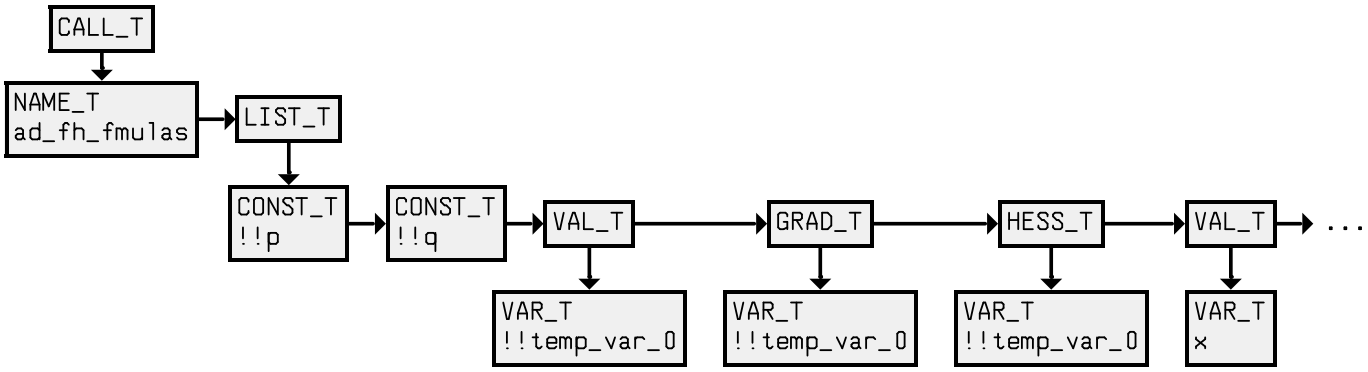

Figure 7: The sample tree after template instantiation with calls to a subroutine library to propagate derivatives. Note that this is only a small part of the total tree produced. The `!!p` and `!!q` nodes will be instantiated with references to the gradient and Hessian lengths.

```
 call ad_fh_fmulas(ad_p_, ad_q_, r_var0, adg_r_var0, ad_pmax_,
+     adh_r_var0, ad_qmax_, x, adg_x, adg_ld1_x, adh_x, adh_ld2_x,
+     y, adg_y, adg_ld1_y, adh_y, adh_ld2_y)
 call ad_fh_faddcs(ad_p_, ad_q_, z, adg_z, adg_ld1_z, adh_z,
+     adh_ld2_z, 2.0, r_var0, adg_r_var0, ad_pmax_,adh_r_var0,
+     ad_qmax_)
```

Figure 8: Fortran code generated by ADIFOR.

3. Finally, we plan to integrate the SparsLinC library to support sparse Hessians where the sparsity pattern is now known in advance and to produce inlined code through an additional template expansion stage. While experiments with inline code generation have suggested that the code expansion is unacceptably large, inline versions of the sections of a code which most impact the performance should produce a suitable compromise between code expansion and execution speed.

In conclusion, even though AIF is in its infancy, the AIF approach has proven itself valuable for experimenting with AD algorithms. All of the language-specific issues are removed from the augmentation module, allowing full concentration on the algorithms, and greatly accelerating implementation of algorithmic improvements like the ones discussed above.

References

[1] Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., and Sorensen, D. *LAPACK User's Guide Release 2.0.* SIAM, Philadelphia, 1994.

[2] Averick, B., Moré, J., Bischof, C., Carle, A., and Griewank, A. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing 15*, 2 (1994), 285–294.

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

[4] Bischof, C., Corliss, G., and Griewank, A. Structured second- and higher-order derivatives through univariate Taylor series. *Optimization Methods and Software 2* (1993), 211–232.

[5] Bischof, C., Green, L., Haigler, K., and Knauff, T. Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4261 (1994), American Institute of Aeronautics and Astronautics, pp. 73–84.

[6] Bischof, C., Roh, L., and Mauer, A. ADIC — An extensible automatic differentiation tool for ANSI-C. Preprint ANL/MCS-P626-1196, 1996.

[7] Bischof, C. H., and Haghighat, M. R. On hierarchical differentiation. Preprint ANL/MCS-P571-0396, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[8] Griewank, A. Some bounds on the complexity of gradients, Jacobians, and Hessians. Preprint MCS-P355-0393, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.

[9] Hovland, P., Bischof, C., Spiegelman, D., and Casella, M. Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. Preprint MCS-P491-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995. To appear in SIAM J. Scientific Computing 18-4 (July 97).

[10] Moore, R. E. *Interval Analysis.* Prentice-Hall, 1966.

[11] Parr, T. SORCERER — a source-to-source translator generator. Preprint AHPCRC 93-094, Army High Performance Computing Research Center, University of Minnesota, 1993.

[12] Parr, T. J. *Language Translation using PCCTS and C++: A Reference Guide.* Automata Publishing Co., 1997.

[13] Rall, L. B. *Automatic Differentiation: Techniques and Applications*, vol. 120 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1981.

[14] Shubin, G. R., Stephens, A. B., Glaz, H. M., Wardlaw, A. B., and Hackerman, L. B. Steady shock tracking, Newton's method, and the supersonic blunt body problem. *SIAM J. on Sci. and Stat. Computing 3*, 2 (June 1982), 127–144.