1

Semiautomatic Differentiation for Efficient Gradient Computations

David M. Gay

Optimization and Uncertainty Estimation Department
Sandia National Laboratories [**]
dmgay@sandia.gov

Summary. Many large-scale computations involve a mesh and first (or sometimes higher) partial derivatives of functions of mesh elements. In principle, automatic differentiation (AD) can provide the requisite partials more efficiently and accurately than conventional finite-difference approximations. AD requires source-code modifications, which may be little more than changes to declarations. Such simple changes can easily give improved results, e.g., when Jacobian-vector products are used to solve nonlinear equations iteratively. When gradients are required (say, for optimization) and the problem involves many variables, "backward AD" in theory is very efficient, but when carried out automatically and straightforwardly, may use a prohibitive amount of memory. In this case, applying AD separately to each element function and manually assembling the gradient pieces — semiautomatic differentiation — can deliver gradients efficiently and accurately. This paper concerns on-going work; it compares several implementations of backward AD, describes a simple operator-overloading implementation specialized for gradient computations, and compares the implementations on some mesh-optimization examples. Ideas from the specialized implementation could be used in fully general source-to-source translators for C and C++.

Key words: Semiautomatic differentiation, backward AD, RAD package, operator overloading, mesh elements, manual assembly, Jacobian-vector products, C/C++ source-to-source, TFad.

1.1 Introduction

Many large-scale computations concern partial differential equations (PDEs) based on physical systems and thus involve discretizations that approximate

[**] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000. This document is released as SAND Number 2004-4688P and is to appear in [8].

physical objects on meshes. Such discretizations generally yield systems of nonlinear equations whose residuals involve the elements of a mesh. As a PDE model matures, interest often grows in optimizing some aspects of the model, i.e., of solving optimization problems with PDE constraints. Like the constraint residuals, the objectives are generally sums of contributions from functions of the mesh elements.

For solving both discretized PDEs themselves and optimization problems involving them, partial derivatives (or approximations to them) are required. Conventionally, these partials are often approximated by finite differences, but finite differences have several drawbacks. Finding suitable step sizes that balance truncation and round-off error can be tricky, and the overall error in a finite-difference approximation can contribute to computational difficulties. Moreover, when partials with respect to many variables are required, finite differences can be slow. Automatic differentiation (AD) provides a more accurate and often faster alternative to finite differences.

As Griewank [18] showed in a survey that appeared in 1989, AD has been reinvented many times. His more recent book [19] tells much more about AD than we will discuss here. Of primary interest here are first derivatives, which may be computed either by *forward AD*, in which one recurs the desired partials while carrying out each operation in the expression of interest, or by *backward AD*, in which one first evaluates an expression, then visits its operations again in reverse order to recur partials (so-called *adjoints*) of the final expression result with respect to the result of each operation.

Forward AD works well when only a few independent variables are involved, but its complexity can grow with the number of independent variables. In particular, when there is only one independent variable, forward AD can efficiently and conveniently compute derivatives of high order. Nonlinear equations can be solved by matrix-free Newton-Krylov methods, which simply use Jacobian-vector products. Such computations effectively involve just one independent variable, and are well handled by forward AD. (For example, TFad [3] works well in some applications at Sandia National Labs [5].)

When there are many independent variables, as is often the case in optimization problems, backward AD is attractive for computing gradients. It delivers function and gradient in time proportional to that required for a function evaluation alone. Unfortunately, when used straightforwardly, backward AD may require memory proportional to the number of operations in the function evaluation, which may appear prohibitive in large-scale computations.

The rest of this paper is organized as follows. The next section gives more discussion of computations on a mesh. Section 1.3 reviews some currently available ways to implement AD. Section 1.4 describes a new, simple, specialized implementation of backward AD by operator overloading in C++. Some timing results on mesh-optimization objectives appear in Sect. 1.5. Section 1.6 discusses implications for source-to-source transformation of C and C++, and Sect. 1.7 offers concluding remarks.

1.2 Action on a Mesh

As mentioned in Sect. 1.1, many large-scale computations involve meshes. For optimization problems whose objectives and constraints involve sums of functions of mesh elements, one can use backward AD to compute (separately) the contributions of each mesh element to the objective and constraint gradients and manually assemble these pieces into the overall gradients, an approach sketched in [1]. Preliminary investigations suggest that this approach should work well in some problems of interest at Sandia, such as mesh optimization (moving interior mesh points to improve the quality of a given mesh) and PDE-constrained optimization. Only a few kinds of mesh elements appear in such problems, making it feasible to treat each separately, either by operator overloading or by source transformation and optimization of routines that compute the element functions. This results in what might be called semiautomatic differentiation: combining use of an AD tool with manual assembly. An advantage of this approach is that it greatly reduces the memory requirements of backward AD.

1.3 Some AD Alternatives

Straightforward use of AD is facilitated by various tools, such as those listed on the `autodiff` web site [4]. These tools work with computations expressed in a suitable programming language, such as C++ or Fortran 95, or a special-purpose language, such as MATLAB [25] or AMPL [10, 11], and use several implementation techniques, as sketched below.

1.3.1 Operator Overloading

Perhaps the most straightforward implementation technique is operator overloading in languages that support it, such as C++ and Fortran 90. An excellent, general, and often used example for C++ is ADOL-C [20, 21]. Use of ADOL-C requires some simple source modifications, which are typical of the sort of modifications needed by AD tools that work with conventional programming languages. Variables with respect to which derivatives are required, and all variables computed from them, must be given a special type. When such "active" variables appear in an "active section," delimited in ADOL-C by `trace_on` and `trace_off` statements, ADOL-C records arithmetic operations on the variables in a "tape," a data structure that summarizes the computation. Subsequently the tape can be "played" to carry out various AD computations. (Forward AD using the operator-overloading approach does not require use of a tape, but ADOL-C gains flexibility and generality from its use of tapes.) With ADOL-C, a special syntax involving `<<=` indicates assignment of input values to the input variables, and another syntax involving `>>=` indicates assignment of output values, partial derivatives of which can be computed subsequently by AD. The process of recording a tape is somewhat slow (as indicated by the timings in Sect. 1.5), but once a tape has been recorded, it can be reused with different values of the input variables, so long as all logical expressions involving active variables come out the same as during the taping. Reusing a previously recorded tape is faster than recording a new one, as illustrated in Sect. 1.5.

1.3.2 Source Transformation

Source transformation is an implementation technique that can give faster execution than straightforward operator overloading. The idea is for a tool to rewrite a computation expressed in given imperative language, such as C or Fortran, giving a more elaborate computation in the same language that carries out the original computation along with automatic differentiation thereof. An early general-purpose instance of this approach is Kedem's use [23] of the AUGMENT preprocessor [9] to carry out forward AD or computation of Taylor coefficients for computations expressed in Fortran 66. AUGMENT effectively implemented operator overloading via source transformation and did not attempt to optimize the computations. ADIFOR [6, 7] is a more recent effort that addresses Fortran 77 and does backward AD within statements while carrying out forward AD overall, thus often achieving greater efficiency than a simple forward AD computation would give. ADIFOR does not use a tape, which also helps make its forward-mode computations faster than those of ADOL-C. A still more recent effort is TAF [29], a commercial successor to TAMC [17] that addresses Fortran 95 and with which several speakers reported impressive numerical results at the Fourth International Conference on Automatic Differentiation [2].

1.3.3 Special Compiler

A variant of source transformation is to have the compiler itself recognize special types and statements that cause AD computations. The NAGWare Fortran 95 compiler [26, 27] provides an example of this approach.

1.3.4 Implicit Domain Knowledge

Special-purpose languages can exploit automatic differentiation without the small syntactic burden imposed on users of general-purpose programming languages. For example, users of the AMPL language for mathematical programming [10, 11] merely express objectives and constraints in a mathematical notation without indicating anything about the partial derivatives that a solver might need. The system deduces “active” variables behind the scenes and arranges AD computations where needed.

1.3.5 Interpreted Evaluations

Interpreted evaluations are an implementation technique that offers considerable flexibility at some cost of speed. Rather than compiling problem-specific machine code, one uses expression representations that are constructed and evaluated easily "on the fly." There are many ways to handle the details (and the distinction between compiled and interpreted evaluations can become murky). One can define a virtual machine in the style of Pascal or Java. Logically equivalent to a virtual machine is the list of 4-tuples of integers that GlobSol [22] uses, the first indicating an operation, the latter three operands.

Another logically equivalent form of interpreted evaluation is to use function pointers in an expression graph. Because of its convenience, this is the approach taken by AMPL and its solver-interface library [14]. Timings involving this approach appear below, so sketching some more of its details seems appropriate. Each operation is represented by a structure with pointers to operands and to a function that carries out the operation and stores partial derivatives for use in reverse AD. For example, a binary operation in a setting where function and gradient are desired may have the form (in C notation)

```
struct expr {
             real (*op)(expr*);
             expr *L, *R; /* left and right operands */
             real dL, dR; /* left and right partials */
             };
```

and the `op` in an `expr` for a multiplication operation might be

```
real OPMULT(expr *e) {
           e->dR = (*e->L->op)(e->L);
           e->dL = (*e->R->op)(e->R);
           return e->dR * e->dL;
           }
```

In reality, there may be other fields and auxiliary variables and a different layout that considers alignment, but this illustrates the gist of the approach.

With this latter approach, when setting up the data structures, one can arrange for the backwards computation of adjoint values to be carried out by a very simple loop, as illustrated by Fig. 1.1, in which a `derp` describes a *der*ivative *p*ropagation operation. The initial assignment of `1.` reflects that the partial of the final result f with respect to itself is 1, i.e., $\frac{\partial f}{\partial f} = 1$. Each iteration of the loop in Fig. 1.1 updates the adjoint corresponding to an operand of one of the operations in the computation. Both `d->a` and `d->c` point to adjoints, and `d->b` points to a partial derivative; unary `*` dereferences pointers, so `*d->a` is the adjoint to which `d->a` points. Thus "`*d->a += *d->b * *d->c`" adds the product of `*d->b` and `*d->c` to the adjoint `*d->a`.

```
void derprop(derp *d) {
             *d->b = 1.;
             do *d->a += *d->b * *d->c;
                     while(d = d->next);
             }
```

Fig. 1.1. Backward propagation of adjoints in the AMPL/solver interface library.

1.3.6 Optimized Compiled Evaluations with *nlc*

For solving nonlinear programming problems, the above style of interpreted evaluations often suffices when the times taken by other parts of the computation dominate the times taken to carry out function and gradient evaluations. In some settings such interpreted evaluations may be too slow, so it is interesting to ask about the extent to which the evaluations can be made faster by generating and compiling problem-specific source code. For example, doing a multiplication directly rather than invoking `OPMULT` will save call-overhead time, and the computations carried out by `derprop` often involve adding zero to a number or multiplying a number by one and thus present opportunities for optimization when we generate problem-specific source code. The *nlc* program [15] carries out such code optimizations in the process of writing C or Fortran to compute function and gradient values for the objectives and constraints expressed in a "`.nl`" file, which AMPL writes to convey problem information to solvers. The test results in Sect. 1.5 below include times from C produced by *nlc*.

One drawback of *nlc* is that AMPL only expresses primitive-recursive functions, i.e., those that can be turned into in straight-line code (with no loops — only forward branches). Imported functions provide an escape hatch that permits anything to be computed, but AMPL's imported functions must provide partial derivatives with respect to their arguments for use in AD computations.

1.4 The *RAD* Package for Reverse AD

It seems interesting to ask how efficiently we can carry out function and gradient evaluations with an implementation of operator overloading in C++ that is specialized for such computations. To this end, I have written a simple backwards AD package, *RAD* (for Reverse AD), that consists of a header file, `rad.h`, and a source file of auxiliary functions; see [28]. When a function is evaluated, *RAD* sets up data structures that permit the backwards AD sweep to take a form similar to that in Fig. 1.1. This form is shown in Fig. 1.2, in which each `aval` is an *a*djoint *val*ue and `*d->a` is a partial derivative.

```
for(; d; d = d->next)
        d->c->aval += *d->a * d->b->aval;
```

Fig. 1.2. Inner loop of *RAD*'s `ADcontext::Gradcomp();`

One target use for *RAD* is computing a sum of functions defined on mesh elements, with a separate evaluation of function and gradient on one mesh element before moving on to the next one, and with manual summing of the element gradients into the overall gradient. Because of this goal, memory is allocated in large chunks that are not freed, but are retained for reuse on subsequent mesh elements, thus reducing the overhead of allocating small objects and eliminating the overhead of freeing them.

With *RAD*, "active" variables that appear in function evaluations have type `ADvar`. Independent `ADvar` variables — inputs with respect to which partial derivatives are desired — are simply assigned numeric values. Dependent `ADvar` variables are computed from expressions involving independent ones, previously computed dependent ones, and any other numeric values with respect to which partial derivatives are not needed. Dependent `ADvar` variables may be updated as desired, and all `ADvar` variables may participate in loops and function calls without restriction. Once the dependent `ADvar` variable representing the function result has been assigned its (final) value, one invokes

```
ADcontext::Gradcomp();
```

to cause the backwards AD sweep and reclamation of memory used for the computation just completed. Because the memory is not freed, the last value assigned to an `ADvar v` and the corresponding adjoint value (computed by `ADcontext::Gradcomp()`) remain available as `v.val()` and `v.adj()`, respectively, until the next assignment to an `ADvar`, which will start reusing the allocated memory.

What enables `ADvar` values to be updated is that an `ADvar` is implemented as a pointer to a structure that contains fields for the `val()` and `adj()` values of the `ADvar`'s current value and for partial derivatives associated with the operation that gave the `val` field its value. In Fig. 1.2, `d->c->aval` and `d->b->aval` are `adj()` fields and `d->a` points to a partial derivative. When an `ADvar` is updated, it is adjusted to point to a new structure.

As an example on which we report timings in Sect. 1.5, Fig. 1.3 shows source for a function, `phi1(x,g)`, that returns a quality measure, $\phi_1(A)$, for an element of a three-dimensional mesh [12, 13] and stores its gradient in the second argument. The function $\phi_1(A)$ is given by

$$\phi_1(A) = \frac{3\det(AW^{-1})^{2/3}}{\| AW^{-1} \|_F^2}, \tag{1.1}$$

in which the 3×3 matrix A has the form

$$A = [v_1 - v_0, v_2 - v_0, v_3 - v_0],$$

where v_0, v_1, v_2, and v_3 are four vertices of a mesh element. The 3×3 matrix W is constant for each kind of mesh element and represents an ideal shape; the source in Fig. 1.3 deals with one kind of mesh element, and the multiplication AW^{-1} is computed in the assignments to the `aw` array. The coordinates of the v_i appear in successive components of the incoming `xx` array. Note how the gradient components are read out after the invocation of `ADcontext::Gradcomp()` and how `f.val()` is returned as the function value.

1.5 Test Results

Here we report comparative timings of some alternative ways of carrying out function and gradient evaluations by backwards AD. The timings were done on two Linux machines, *Desktop* with a 3 GHz Intel Xeon processor having 512 MB of cache, and *Laptop* with a 1.6 GHz Intel Pentium M processor having no cache. Compilation was with `g++ -O` or `gcc -O`, and the same binaries ran on both machines. The reason for showing results from these two machines is to illustrate that architectural details (such as cache) can affect relative timings.

Table 1.1 shows timings for the function $f = \phi_1$ given by (1.1). The timings are relative to the time for computing f alone by C++ code similar to that in Fig. 1.3, with "`ADvar`" replaced by "`double`" and without references to `g` or `ADcontext::Gradcomp()`. The time per function or function and gradient evaluation behind each table entry was computed in a separate timing loop that ran for several seconds. (On *Desktop*, the computations should all have been running in the cache. This seems fair, as we would try to organize the evaluation of a mesh objective so much of the inner loop would involve data and instructions from the cache.)

Table 1.1. Relative times for $f = \phi_1$

	Desktop	*Laptop*
Compiled f	1.	1.
$f + \nabla f$ by *RAD* (§1.4)	11.0	10.1
$f + \nabla f$ by *nlc* (§1.3.6)	1.35	1.53
ADOL-C taped f (§1.3.1)	4.83	5.54
" taped $f + \nabla f$	14.5	14.9

The last two lines of Table 1.1 are for ADOL-C evaluating a previously recorded tape. The computation of f from the tape looks quite efficient. That *RAD* outperforms ADOL-C when computing f and ∇f confirms that specialized operator overloading for AD can be worthwhile. The *nlc* evaluations

```
double phi1(double *xx, double *g) {

     ADvar aw[3][3], det, f, x[4], y[4], z[4];
     int i, j;
     static double one_over_root3 = sqrt(1./3.),
           two_over_root3 = sqrt(4./3.),
           one_over_root6 = sqrt(1./6.),
           root_3_halves  = sqrt(3./2.);

     for(i = j = 0; i < 12; i += 3, j++) {
           x[j] = xx[i];
           y[j] = xx[i+1];
           z[j] = xx[i+2]; }
     for(i = 1; i <= 3; i++) {
           x[i] -= x[0];
           y[i] -= y[0];
           z[i] -= z[0]; }
     aw[0][0] = x[1];    aw[1][0] = y[1];   aw[2][0] = z[1];

     aw[0][1] = two_over_root3*x[2] - one_over_root6*x[1];
     aw[1][1] = two_over_root3*y[2] - one_over_root6*y[1];
     aw[2][1] = two_over_root3*z[2] - one_over_root6*z[1];
     aw[0][2] = root_3_halves*x[3] - one_over_root6*(x[1] + x[2]);
     aw[1][2] = root_3_halves*y[3] - one_over_root6*(y[1] + y[2]);
     aw[2][2] = root_3_halves*z[3] - one_over_root6*(z[1] + z[2]);

     for(f = 0., i = 0; i < 3; i++)
           for(j = 0; j < 3; j++)
                 f += aw[i][j]*aw[i][j];

     det =   aw[0][0]*aw[1][1]*aw[2][2]
           + aw[1][0]*aw[2][1]*aw[0][2]
           + aw[2][0]*aw[0][1]*aw[1][2]
           - aw[2][0]*aw[1][1]*aw[0][2]
           - aw[1][0]*aw[0][1]*aw[2][2]
           - aw[0][0]*aw[2][1]*aw[1][2];

     f = 3*pow(det, 2./3.) / f;

     ADcontext::Gradcomp();

     for(i = j = 0; i < 12; i += 3, j++) {
           g[i]   = x[j].adj();
           g[i+1] = y[j].adj();
           g[i+2] = z[j].adj(); }
     return f.val();
     }
```

Fig. 1.3. Source for `phi1(x,g)` corresponding to (1.1).

look remarkably efficient, delivering on the promise of AD to compute f and ∇f in a small multiple of the time for computing f alone.

Some of the overhead in evaluating ϕ_1 and $\nabla\phi_1$ is masked by the time taken by the `pow` invocation in Fig. 1.3, i.e., by raising $\det(AW^{-1})$ to the power 2/3. We can eliminate this overhead by dealing with $\phi_2 = (\phi_1/3)^3$, i.e.,

$$\phi_2(A) = \frac{\det(AW^{-1})^2}{\| AW^{-1} \|_F^6} \ . \tag{1.2}$$

Using ϕ_2 for mesh optimization (the problem giving rise to ϕ_1) is not necessarily desirable because ϕ_2 penalizes "large" elements much more than ϕ_1 does, but it is interesting to see how the values in Table 1.1 change when the overhead of exponentiation goes away. Table 1.2 gives relative timings for (1.2); the overheads for all the variants of computing ∇f go up but are qualitatively similar to those in Table 1.1, and the *nlc* evaluations still give f and ∇f in less than thrice the time of computing f alone.

Table 1.2. Relative times for $f = \phi_2 = (\phi_1/3)^3$

	Desktop	*Laptop*
Compiled f	1.	1.
$f + \nabla f$ by *RAD*	37.8	27.2
$f + \nabla f$ by *nlc*	2.54	2.13
ADOL-C taped f	16.6	13.7
" taped $f + \nabla f$	55.6	40.0

We conclude this section by showing timings on a more elaborate mesh-quality function [24], $\mu_1(A)$, defined by (1.3)–(1.5):

$$\tau = \det(AW^{-1}), \tag{1.3}$$

$$h = \frac{1}{2}(\tau + \sqrt{\tau^2 + 4\delta^2}), \tag{1.4}$$

$$\mu_1(A) = h^{-2/3} \ \| AW^{-1} - I \|_F^2 \ . \tag{1.5}$$

The 3×3 matrices A and W in (1.3) and (1.5) are as in (1.1), and δ in (1.4) is a constant. Note that evaluating $f = \mu_1$ involves extra overhead from both exponentiation and a square-root computation.

Relative timings for $f = \mu_1$ appear in Table 1.3. All times are for evaluations of f and ∇f. The "Compiled f" times are for hand-coded function and gradient evaluations. They factor A, compute $\det(A)$ from the factorization, and use the identity

$$\frac{\partial \log \det A}{\partial t} = \operatorname{trace}\left(A^{-1}\frac{\partial A}{\partial t}\right)$$

in computing ∇f, in part because this machinery is useful in computing $\nabla^2 f$, a matter discussed briefly in Sect. 1.7 below. Even with the factorization, etc., done with inline, loop-free code, the calculation is slightly slower than the corresponding one derived by applying *nlc* to the AMPL model shown in Fig. 1.4, so the times in Table 1.3 are relative to these *nlc* times.

```
var xyz{i in 0..2, j in 0..2};
var winv{0..2, 0..2};   # really a constant param
var delta := .1;        # really a constant param

var aw{i in 0..2, j in 0..2} = sum{k in 0..2} xyz[i,k]*winv[k,j];

var det = aw[0,0]*aw[1,1]*aw[2,2]
        + aw[1,0]*aw[2,1]*aw[0,2]
        + aw[2,0]*aw[0,1]*aw[1,2]
        - aw[2,0]*aw[1,1]*aw[0,2]
        - aw[1,0]*aw[0,1]*aw[2,2]
        - aw[0,0]*aw[2,1]*aw[1,2];

var h = 0.5 * (det + sqrt(det^2 + 4*delta^2));

var mu1a = 0.5 * sum{i in 0..2, j in 0..2}
                        (aw[i,j] - if i == j then 1)^2;
minimize mu1: mu1a / h^(2/3);
```

Fig. 1.4. AMPL model for μ_1.

Table 1.3. Relative times for $f = \mu_1$ and ∇f

	Desktop	*Laptop*
Hand-coded	1.07	1.12
ASL	11.3	11.6
ASL for $\nabla^2 f$	13.0	13.4
RAD	9.14	7.06
nlc	1.	1.
ADOL-C new tape	55.0	37.7
ADOL-C old tape	15.4	14.1

The ASL times are for interpreted evaluations of Fig. 1.4 with the AMPL/solver interface library, as in Sect. 1.3.5. When set up to do Hessian computations, these evaluations incur the extra overhead during function evaluations of storing some second partial derivatives. This overhead is reflected in the "ASL for $\nabla^2 f$" line of Table 1.3.

The "ADOL-C new tape" times in Table 1.3 show the cost with ADOL-C of recording a tape. These times are to be contrasted with those in the "ADOL-C old tape" line for reusing a previously recorded tape, and with those in the *RAD* line for overloading specialized to f and ∇f.

1.6 Implications for Source Transformation

The optimizations done by the *nlc* program could also be done (at least on straight-line code) by a source-to-source translator or special compiler that focused on automating gradient computations. The gap between the times for *RAD* and *nlc* in Table 1.3 reflects the opportunities mentioned in Sect. 1.3.6 for optimization in such transformations. Of course, like *RAD*, such transformations should handle completely general source, with only the usual limitations on AD computations. (For example, AD applied to "`(x == 3 ? 5 : x + 2)`" would compute 0 rather than 1 for the derivative at $\mathtt{x} = 3$.) The approach taken in *RAD* could work well in such transformations, at least as long as sufficient memory is available. This approach would present various opportunities to further reduce overheads by computing some things at compile (or transformation) time and thus to speed up the computations.

1.7 Concluding Remarks

One motivation for this work was to research AD approaches that might work well on an objective function defined on elements of a mesh, particularly when the objective is the sum of functions computed on individual mesh elements. Although the memory required for straightforward backward AD could be prohibitive on large meshes, little memory may be needed to compute a function and its gradient on an individual mesh element, and assembling the individual mesh-element gradients into an overall objective gradient "by hand" may be straightforward. Thus we obtain a reliable and efficient way to carry out function and gradient evaluations for some problems (albeit not necessarily for problems with objectives or constraints that involve integration over time — unless time is treated analogously to the spacial dimensions).

An AD approach introduced in this paper is the *RAD* package for function and gradient computations via operator overloading in C++. Since it is fully general and easy to use, *RAD* may find uses in various applications. The implementation techniques described in Sect. 1.4 could prove useful in special source-to-source translators or compilers meant to facilitate AD computations.

A growing number of nonlinear programming solvers use Hessians (matrices of second partials), so it is of interest to see how easily we can arrange for their efficient computation. The interpreted Hessian evaluations offered by the AMPL/solver interface library [15, 16] are convenient but not very fast. For example, for the function $f = \mu_1$ given by (1.5), hand-coded evaluations of f, ∇f, and $\nabla^2 f$ run about 100 times faster than computations with the AMPL/solver interface library. It would be interesting to see how much these computations could be sped up by an extension of *nlc* that addressed Hessian computations along with functions and gradients.

Acknowledgment. I thank Scott Mitchell and the referees and editors for their helpful comments on the manuscript.

References

1. Jason Abate, Steve Benson, Lisa Grignon, Paul D. Hovland, Lois C. McInnes, and Boyana Norris. Integrating AD with object-oriented toolkits for high-performance scientific computing. In George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors, *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, chapter 20, pages 173–178. Springer, New York, NY, 2002.
2. The 4th International Conference on Automatic Differentiation, 2004. `http://www.autodiff.org/ad04/`.
3. Pierre Aubert, Nicolas Di Césaré, and Olivier Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*, 3:197–208, 2001.
4. Autodiff web site. `http://www.autodiff.org/Tools/index.php`.
5. Roscoe A. Bartlett (`rabartl@sandia.gov`). Private communication, 2004.
6. Christian H. Bischof, Alan Carle, George F. Corliss, Andreas Griewank, and Paul D. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1:11–29, 1992.
7. Christian H. Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.
8. H. Martin Bücker, George F. Corliss, Paul Hovland, Uwe Naumann, and Boyana Norris, editors. *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering.* Springer, New York, NY, 2005.
9. Fred D. Crary. A versatile precompiler for nonstandard arithmetics. *ACM Trans. Math. Software*, 5(2):204–217, 1979.
10. Robert Fourer, David M. Gay, and Brian Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990.
11. Robert Fourer, David M. Gay, and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2003. Second edition; ISBN 0-534-38809-4.
12. Lori Freitag, Patrick Knupp, Todd Munson, and Suzanne Shontz. A comparison of optimization software for mesh shape-quality improvement problems. In *Proceedings of the 11th International Meshing Roundtable*, Ithaca, NY, 2002. `http://www.imr.sandia.gov/papers/imrll/freitag.pdf`.
13. Lori Freitag Diachin, Patrick Knupp, Todd Munson, and Suzanne Shontz. A comparison of inexact newton and coordinate descent mesh optimization techniques. In *Proceedings of the 13th International Meshing Roundtable*, Williamsburg, VA, 2004. `http://www.imr.sandia.gov/papers/imrll/freitag.pdf`.
14. David M. Gay. Automatic differentiation of nonlinear AMPL models. In A. Griewank and G. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 61–73. SIAM, 1991.
15. David M. Gay. Hooking your solver to AMPL. Numerical Analysis Manuscript No. 93-10, AT&T Bell Laboratories, Murray Hill, NJ, 1993, revised 1997. `http://www.ampl.com/REFS/hooking2.ps.gz`.
16. David M. Gay. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation : Techniques, Applications, and Tools*, pages 173–184. SIAM, 1996.

17. Ralf Giering and Thomas Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Software*, 24(4):437–474, 1998.
18. A. Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming*, pages 83–107. Kluwer Academic Publishers, 1989.
19. A. Griewank. *Evaluating Derivatives, Principles and Techniques of Algorithmic Differentiation*, volume 19 of *Frontiers in Applied Mathematics*. SIAM, 2000.
20. A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math Software*, 22(2):131–167, 1996.
21. Andreas Griewank, David Juedes, Hristo Mitev, Jean Utke, Olaf Vogel, and Andrea Walther. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. Technical report, Institute of Scientific Computing, Technical University Dresden, 1998. Version 1.8, `ftp://ftp.math.tu-dresden.de/pub/ADOLC/ADOLC_1.8/adolc_1.8.ps.gz`.
22. Baker Kearfott. *Rigorous Global Search: Continuous Problems*, volume 23 of *Nonconvex Optimization and its Applications*. Kluwer, 1996.
23. Gershon Kedem. Automatic differentiation of computer programs. *ACM Trans. Math. Software*, 6(2):150–165, 1980.
24. Patrick Knupp (`pknupp@sandia.gov`). Private communication, 2004.
25. MATLAB web site. `http://www.mathworks.com`.
26. NAG. Automatic differentiation: Differentiation enabled Fortran compiler technology. `http://www.nag.co.uk/nagware/research/ad_overview.asp`.
27. U. Naumann and J. Riehme. Computing adjoints with the NAGWare Fortran 95 compiler. In Bücker et al. [8], pages xxx–yyy.
28. URL for downloading the *RAD* package for reverse AD. `http://endo.sandia.gov/~dmgay/rad.tar.gz`.
29. TAF web site. `http://www.fastopt.de/`.