Interval Computations
No 3, 1993

Automatic Differentiation Applied to Unconstrained Nonlinear Optimization with Result Verification

Ronald Van Iwaarden

This paper explores using both forward and reverse modes of automatic differentiation to solve the standard unconstrained optimization problem and verify the solution that is found. The two types of automatic differentiation are compared when the dimension of the problem is increased. This research shows that the reverse mode is superior when time is the largest constraint and that the forward mode is superior when memory requirements are of greatest concern.

Применение автоматического дифференцирования в задаче нелинейной оптимизации без ограничений с верификацией результата

Р. Ван Иварден

Рассматривается применение прямого и обратного способа автоматического дифференцирования для решения стандартной задачи оптимизации без ограничений и верификации получаемого таким образом результата. Эти два типа автоматического дифференцирования сравниваются между собой при увеличении размерности задачи. Настоящее исследование показывает, что обратный способ оказывается лучше, когда наиболее существенным ограничением является время, и что прямой способ имеет преимущества, когда важнее всего требования, предъявляемые к объему памяти.

© R. Van Iwaarden, 1994

1 Introduction: Newton's method

The unconstrained optimization problem

$$\min_{x \in \Re^N} f(x)$$

where f is a real valued twice continuously differentiable function, and its level sets are bounded. Our goal is to compute some x^* such that

$$f(x^*) \leq f(x) \qquad \forall x \in \Re^N.$$

Computing this local minimizer may not always be feasible so that the generally easier problem of finding a local minimum is substituted for the original one. That is, an x^* is computed for which there exists $\delta > 0$ such that

$$f(x^*) \leq f(x) \qquad \forall x \quad \text{such that } \|x - x^*\| \leq \delta.$$

If x^* is a local minimizer, then the first-order necessary condition is

$$\nabla f(x^*) = 0. \tag{1}$$

If the Hessian matrix at x^*, denoted $H_f(x^*)$, is positive definite, then the inequalities above hold with strict inequality when $x \neq x^*$. This is referred to as a strong local minimum.

A classical method for solving problem (1) is Newton's method. To develop Newton's method, approximate $f(x)$ by Taylor's theorem and assume that f is given exactly by the first three terms of its Taylor series expansion

$$f(x + p) = f(x) + \nabla^T f(x)p + \frac{1}{2}p^T H_f(x)p.$$

To minimize this function over all p, the first order necessary conditions give

$$\nabla f(x) + H_f(x)p = 0.$$

Newton's method uses the solution

$$p = -H_f^{-1}(x)\nabla f(x)$$

and we must solve an $N \times N$ system of equations to find the search direction p. This process yields the following algorithm:

Step 1. Solve $H_f(x)p = -\nabla f(x)$.

Step 2. Set $x \leftarrow x + p$.

Step 3. If $\|\nabla f(x)\| < \epsilon$, for some $\epsilon > 0$ stop; else go to Step 1.

As is well known, the main advantage and attraction of Newton's method are that, if the function is sufficiently smooth and x is close enough to a strong local minimizer x^*, then Newton's method is quadratically convergent.

This advantage, however, is not generally enough to counteract the disadvantages of Newton's method, some of which are the following.

1. The method is not globally convergent. That is, it may not converge to a stationary point for an arbitrary x_0, and it may even diverge.
2. If the Hessian $H_f(x)$ is singular, the method is not defined.
3. If the function is not convex, the set of directions given may not define a set of descent directions.
4. An $N \times N$ system of equations must be solved at each iteration.

Of these disadvantages, (1) and (3) can be solved by a slight modification of Newton's method. Define $\phi(\alpha) = f(x + \alpha p)$ for $0 < \alpha \leq 1$ and the modified Newton's method becomes

Step 1. Solve $H_f(x)p = -\nabla f(x)$.

Step 2. Find α so that the following are satisfied:

a) $\phi(\alpha) < \phi(0) + \epsilon\phi'(0)\alpha$,
b) $\phi(\alpha) > \phi(0) + (1 - \epsilon)\phi'(0)\alpha$.

Step 3. Set $x \leftarrow x + \alpha p$.

Step 4. If $\|\nabla f(x)\| < \epsilon$, stop; else go to Step 1.

The conditions in Step 2 are known as the Goldstein test [13] and they help to control the global behavior of the method. Early in the calculation, they play a significant role, forcing the method to make a step that

minimizes the function at each step, and they also prevent the modified Newton's method from making too radical a change. Later in the calculation, assuming the function tends asymptotically toward a quadratic, the stepsize approaches 1, and the Goldstein test has less of a role.

1.1 Truncated Newton's methods

It may not be wise to solve the Newton's equations, Step 1, when the present value for x is far from the solution. Dembo and Steinhaug [2] proposed using the method of conjugate gradients for solving Newton's equations. If we let the present solution to Newton's equations be given by x_k, then the residual is given by $r_k = H_f x_k - \nabla f(x)$. The iterations of the conjugate gradient method are then truncated when the error, given by $||r_k||/||\nabla f(x)||$, is "small enough" rather than continuing until the error is zero. (Hence the name truncated Newton's method). Dembo and Steinhaug showed that this method is globally convergent to a local minimum with any rate of convergence between linear and quadratic being possible depending on the size of $||r_k||/||\nabla f(x)||$.

The conjugate gradient method has a very important property that makes it especially suited for using automatic differentiation to solve Newton's equations. That property is that the conjugate gradient method never requires the Hessian matrix to be known explicitly, only a Hessian-vector product $H_f(x)p$ is required. It is assumed that the function f is in C^2, which implies that the Hessian is symmetric, and so the product $p^T H_f(x)$ is also acceptable.

This research implemented the standard conjugate gradient method to demonstrate the tradeoffs between forward AD and reverse AD with respect to storage and computational time. The reader should note that much has been written about conjugate gradient methods (see, e.g., [8, 9]) and that even a simple diagonal preconditioner can speed up the convergence of this method. This as well as other basic ideas of both the standard Newton's method and the truncated Newton's method [3, 5, 16, 13] are areas for further research.

2 Forward automatic differentiation

There are two types of automatic differentiation (AD), forward and reverse. The forward mode [21] is used by Dixon and Price [4] in their implementation of the truncated Newton's method. To implement forward AD, an algebra on the space T^n of ordered $(n+1)$-tuples $(x, x', \ldots, x^{(n)})$ is defined where $x^{(j)}$ is an j-dimensional tensor. For simplicity and since only second derivatives for the optimization method, $n = 2$, and any element $x \in T^2$ can be represented by (x, x', x''), where x is a scalar, x' is an N-vector, and x'' is an $N \times N$ matrix. Let $U, V \in T^2$, and let $\phi(x) \in C^2$, the operations that are needed are defined in the space T^2 as follows,

$$U = (u, u', u'') \qquad \text{and} \qquad V = (v, v', v'').$$

This gives

$$\begin{aligned}
U + V &= (u + v, u' + v', u'' + v'') \\
U - V &= (u - v, u' - v', u'' - v'') \\
U * V &= (u * v, uv' + vu', uv'' + u'v'^T + v'u'^T + vu'') \\
U \,/\, V &= \big(u/v, (vu' - uv')/v^2, \\
&\qquad (v^2u'' - v(v'u'^T + u'v'^T) + 2uv'v'^T - uvv'')/v^3\big) \\
\phi(U) &= \big(\phi(u), \phi'(u)u', \phi'(u)u'' + \phi''(u)u'u'^T\big) \\
&\qquad \text{for the elementary function } \phi.
\end{aligned}$$

These are the standard operations given by the algebra of Taylor's series, which is just the implementation of the chain rule. With this notation, a variable x_i becomes $(x_i, e_i, 0)$, where e_i is the ith unit vector and 0 is the $N \times N$ matrix of zeros. The rules for Taylor arithmetic come directly from any elementary calculus text and can be modified for faster implementation on a computer. For example:

$$\begin{aligned}
w &= u/v \\
U/V &= \big(w, (u' - wv')/v, (vu'' - (v'u'^T + u'v'^T) + 2wv'v'^T - wvv'')/v^2\big).
\end{aligned}$$

All of these operations require a matrix store and are not what is needed for the conjugate gradient method where a matrix-vector product is used. Dixon and Price [4] adapted forward AD to retain only a vector-Hessian product as follows. Define the space S^2, where any element $u \in S$ has

the form $u = (u, u', u''p)$, where u is a scalar, u' and p are N-dimensional vectors, and u'' is an $N \times N$ matrix. Then, given $U, V \in S$ and $\phi(x) \in C^2$, the operations on this space are given by

$$U = (u, u', u''p) \qquad \text{and} \qquad V = (v, v', v''p)$$

$$\begin{aligned}
U + V &= (u + v, u' + v', u''p + v''p) \\
U - V &= (u - v, u' - v', u''p - v''p) \\
U * V &= (u * v, uv' + vu', uv''p + u'v'^T p + v'u'^T p + vu''p) \\
U \,/\, V &= \big(u/v, (vu' - uv')/v^2, \\
&\qquad (v^2 u''p - v(v'u'^T p + u'v'^T p) + 2uv'v'^T p - uvv''p)/v^3\big) \\
\phi(U) &= \big(\phi(u), \phi'(u)u', \phi'(u)u''p + \phi''(u)u'u'^T p\big).
\end{aligned}$$

With this new notation, an independent variable x_i is represented by $(x_i, e_i, 0)$, where x_i is a scalar, e_i is again the ith unit vector, and 0 is the N-dimensional zero vector. It should also be noted that in all of the above operations, at most dot products are required. The Hessian is never explicitly required. Since the entire Hessian need not be computed, this adaptation results in both vector storage and faster operation for each Hessian-vector product. As with the matrix version, some of these expressions can be simplified for faster implementation on a computer. Iri and Kubota [10] have shown that computing the gradient and Hessian by the forward AD method requires $w(f)O(N^2)$ operations, where $w(f)$ is the work required to compute the function $f(x)$. To compute just the gradient and Hessian-vector product requires at most $w(f)6N$ operations or $w(f)O(N)$ operations.

3 Reverse automatic differentiation

The present form of reverse AD appeared in 1980 [20]. However, the earliest known paper that explicitly recognizes the potential savings that arise in reverse AD is by G. M. Ostrowski et al. [18]. Similar to forward AD, the reverse mode is based on the chain rule, but the chain rule is used in a slightly different form from what we are used to.

Consider the simple example

$$F(x, y) = \sin(x^3 + y).$$

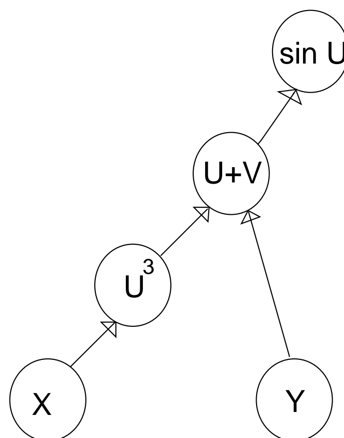

Figure 1: Computational graph for $F(x, y) = \sin(x^3 + y)$.

One possible computational graph is shown in Figure 1, where $v_1 = x^3$, $v_2 = x^3 + y = v_1 + y$, and $v_3 = \sin v_2 = \sin(x^2 + y)$. This computational graph is essentially a graphical representation of the computational process that is used in forward AD. A computational graph is by no means unique. Even in this small example, node v_1 could have been replaced by a $u * v$ node with two input arcs that both originate from the node x and a second $u * v$ node with input arcs from both x and this new v_1.

Once the computational graph has been created, the technique of reverse AD can be applied to the computational graph to compute the needed derivatives. For this process, compute the partial derivatives of any arc $a = (v_i, v_j)$, where v_i is the node where the arc originates, and v_j is the terminal node. The computational graph is then extended to make a new computational graph that computes the partial derivatives. The new graph is created by placing a mirror image of the original graph next to the original. For this mirror image, the direction of each arc is reversed. In the middle of each arc, a multiplication node is placed. All of the original nodes in the mirror image become addition nodes with the exception of the node which corresponded to the function value F. This node receives the value 1. Between the two graphs, one intermediate node is then placed for every arc in the original graph. Finally, the value of 0 is assigned to every + node in the mirror graph.

For each node v_i in the original graph that inputs to node v_j, where node v_j also has inputs from node v_k, three arcs are made that connect these three nodes with the intermediate node $w_{i,j}$. This node contains the function that corresponds to $\frac{\partial v_j}{\partial v_i}$. This node then has an arc that connects to node $u_{i,j}$ in the new computational graph. The node $u_{i,j}$ is the multiplication node between nodes u_i and u_j.

To illustrate this, we once again consider the previous example:

$$\begin{aligned}
\frac{\partial u^3}{\partial u} &= \frac{\partial x^3}{\partial x} = \frac{3u^3}{u} = 3x^2 \\
\frac{\partial u + v}{\partial v} &= \frac{\partial x^3 + y}{\partial y} = 1 \\
\frac{\partial u + v}{\partial u} &= \frac{\partial x^3 + y}{\partial x^3} = 1 \\
\frac{\partial \sin u}{\partial v_2} &= \frac{\partial \sin(x^3 + y)}{\partial x^3 + y} = \cos(x^2 + y).
\end{aligned}$$

The intermediate node corresponding to the arc from $(u+v)$ to $\sin u$ contains the function $\cos u = \cos(x^3 + y)$. This function depends only on inputs from the node $x^3 + y$. The arc from $\sin(x^3 + y)$ does not need to be included in the computational graph. Continuing in this fashion, the intermediate node for the arc from u^3 to $(u + v)$ has the value 1 and has no inputs. Similarly, the intermediate node for the arc from y to $(u + v)$ has the value 1 with no inputs. The intermediate node for the arc from x to u^3 has the value $3x^3/x$, where it has its first input from the node with the value u^3 and its second input from node x. The resultant graph for $\nabla F(x, y)$ appears in Figure 2.

This new computational graph yields the gradient of the function $F(x, y) = \sin(x^3 + y)$. Once this computational graph has been traversed, $\frac{\partial F}{\partial x}$ is contained in the mirrored x node, and $\frac{\partial F}{\partial y}$ is contained in the mirrored y node. This graph has a similar geometry to the original computational graph, and it is rarely computed is practice.

If the Hessian is needed, the new computational graph can be differentiated N times, since the Hessian is just the derivative of the gradient. Again, this computational graph is not significantly different from the original graph, and so it does not need to be explicitly computed. Also, the Hessian-vector product is easily computed using only the original computational graph since this third graph again has a similar geometry to the

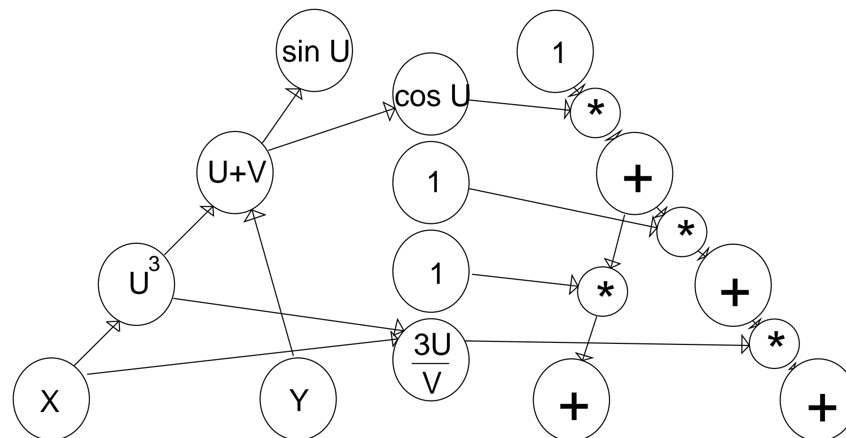

Figure 2: Computational graph for the gradient of $F(x, y) = \sin(x^3 + y)$.

original computational graph. For a thorough discussion of the algorithm and the computational complexity, the reader is referred to Iri [10].

4 Comparison of forward and reverse AD

It is next shown that the reverse and forward methods of AD can be viewed simply as different interpretations of the chain rule. When the forward method is implemented, the total derivatives of node v_i with respect to x_i are known at the point the computation of f reaches node i. Consider the example from the beginning of Section 3, this can be represented as follows:

1. $\frac{\partial v_1}{\partial x} = 3x^2$
   $\frac{\partial v_1}{\partial y} = 0.$

2. $\frac{\partial v_2}{\partial x} = 3x^2$
   $\frac{\partial v_2}{\partial y} = 1.$

3. $\frac{\partial v_3}{\partial x} = 3x^2 \cos(x^3 + y)$
   $\frac{\partial v_3}{\partial y} = \cos(x^3 + y).$

Since $v_3 = f$, at 3, the gradient has been computed. This is how the forward (bottom-up) method of AD is implemented. Assuming x^y is evaluated by

$e^{y \ln x}$ is five multiplications. The total number of additional computations that must be done in order to get the derivative are five multiplications, two additions, and three transcendental function evaluations.

In contrast to this, the reverse method computes the entire function first and saves the intermediate values along with the elementary partial derivatives of each node with respect to its inputs. Then, the graph is traversed in the reverse direction (top-down). The partial derivative of f with respect to node v_i is known when the computation reaches that point. When this is finished, the partial derivative of f with respect to each of the original inputs $x_1, \ldots, x_n$ is known for each x_i. For the example that was used, this can be represented as follows.

1. $\frac{\partial v_3}{\partial v_2} = \cos v_2$
2. $\frac{\partial v_3}{\partial y} = \cos v_2$
3. $\frac{\partial v_3}{\partial v_1} = \cos v_2$
4. $\frac{\partial v_3}{\partial x} = (\cos v_2) 3v_1/x.$

Again, since $v_3 = f$ and $3v_1/x = 3x^2$, the gradient of the function has been computed. Instead of computing $\frac{\partial v_3}{\partial x} = 3x^2$, which requires transcendental function evaluations, we compute $\frac{\partial v_3}{\partial x} = (\cos V_2)3v_1/x$, saving work. For the reverse method, the total amount of additional computational effort expended is five multiplications, one division, one transcendental function evaluation, and four additions. If a division costs approximately two multiplications, a transcendental function costs approximately three multiplications, and addition and subtraction cost half a multiplication, then to evaluate the function alone requires approximately seven multiplications. For the gradient by means of forward AD, an additional fifteen multiplications are needed. For the gradient by means of reverse AD, an additional ten multiplications are needed.

The advantages given by reverse AD in terms of computational time are not great with this problem. If a larger problem is solved, however, the difference is often more striking. The most obvious disadvantage for reverse AD is the need to store the computational graph and all of the intermediate variables. For forward AD, only a vector store is required to compute the gradient. For reverse AD, we are not aware of any bounds on the growth of

Derivative	Finite differences	Reverse AD	Forward AD
∇f	$2Nw(f)$	$5w(f)$	$5Nw(f)$
$\nabla^2 f$	$3N^2w(f)$	$(6+9N)w(f)$	$O(N^2)w(f)$
$y^t\nabla^2 f$	$4N^2w(f)$	$15w(f)$	$O(N)w(f)$

Table 1: Orders of magnitude for different differentiation methods.

the computational graph. However, recent work by Griewank [7] has shown that it is possible to achieve logarithmic growth in the spatial complexity of the computational graph with respect to the run time with a trade-off of logarithmic growth in the temporal complexity.

The primary advantage of reverse AD has been the speed at which derivatives are computed. Iri and Kubota [10] have shown that the work required to compute the function and its gradient requires at most $5w(f)$ evaluations. They also show that the work required to compute the function, its first derivative, and the Hessian-vector product does not exceed $15w(f)$. In other words, the time requirement for evaluating the product of the Hessian and a vector requires at most a constant times the amount of work required to evaluate the function itself. To compute the full Hessian requires fewer than (6+9N)$w(f)$ operations. It is very surprising to note that the reverse method is of an order of magnitude faster than the forward method. This remarkable fact will become even more evident when both methods are used for nonlinear optimization.

The reason the reverse mode of AD is so much faster is that the amount of work required to compute the gradient is dependent on the size of the computational graph. Since the computational graph for the gradient is about three times the size of the graph for the function alone, the amount of work to compute the gradient would appear to be on the order of three times the work to evaluate the function. The constant, however, can be greater than three since the time required to access the values from memory has not been considered. These results are summarized in Table 1.

Table 1 represents only functions of N variables onto the real numbers. When a function from N variables onto $\Re^M$ where $M > 1$ is considered, Table 1 can be completely false. Table 1 would lead one to believe that reverse AD is superior in all cases, but it is very possible that forward AD will be a superior method for certain cases. For example, consider

a function from one variable into 8 variables with a computational graph shaped like a binary tree. With this graph, both forward AD and reverse AD with go through the same amount of effort in computing derivatives. The reader is also referred to the work of Bischof et al. [1], where hybrid schemes are explored to capitalize on the strengths of both the forward and reverse mode. These hybrid schemes have been implemented in the package ADIFOR, which is a Fortran preprocessor to translate a Fortran subroutine into a routine that computes the derivative of the subroutine.

5 Result verification

Once a candidate point $x \in \Re^N$ for the approximate local minimizer (1) has been identified, it is next validated. It is known from Moore [15] that if f is twice continuously differentiable on an interval vector X,

$$K(X) \subseteq X, \tag{2}$$

$$K(X) = y - Y\nabla f(y) + \big[I - YH_f(X)\big](X - y) \tag{3}$$

where y is any vector such that $y \in X$ and Y a nonsingular real matrix, then the solution to $\nabla f(x) = 0$ is contained in X.

Let $Y = \big(H_f(y)\big)^{-1}$ and $y = \text{mid}(X)$. Then $K(X)$ can be seen as a Newton's step

$$y - Y\nabla f(y) \equiv y - \big(H_f(y)\big)^{-1}\nabla f(y)$$

plus a contraction of the present interval

$$\big[I - YH_f(X)\big](X - y) \equiv \big[I - \big(H_f(y)\big)^{-1}H_f(X)\big](X - y).$$

In general, $I - (H_f(y))^{-1}H_f(X) \neq 0$, since $H_f(X)$ is an interval matrix. This yields the following algorithm:

Step 1. Find an approximate solution $\bar{x}$ to the equation $\nabla f(x) = 0$.

Step 2. Set $X = \bar{x} + [-\epsilon, \epsilon]\bar{x}$.

Step 3. If Equation 2 is satisfied, then stop; else go to 1.

At Step 3, $y = \text{mid}(X)$, $Y = \big(\text{mid}(H_f(X))\big)^{-1}$, and $\epsilon = 10^{-10}$ are used in Equation 3. In the examples that are given in the next section, the problems have sparse, narrow-banded Hessians that allow inversion in $O(N^2)$ time. With non-sparse systems, it would be advisable to maintain an approximation to the inverse so that the $O(N^3)$ work involved in inversion can be avoided. The major computational challenge for this research was the memory required for storing the Hessian and the matrix $I - YH_f(X)$. This required N^2 data elements. For the implementation described here, a 256×256 variable matrix required 1 megabyte of memory. This quickly overwhelmed the test machine even though it had 16 megabytes of memory and so larger problems were not implemented.

The implementation for the numerical experiments verifies only that a value for X such that $\exists x \in X \ni \nabla f(x) = 0$ has been found. This does not guarantee a local minimum, but only that an interval X has been found for which $\exists x \in X$ which satisfies the Kuhn-Tucker conditions (see [13]). A new method by Ratz [19] verifies that the interval matrix $H_f(X)$ is positive definite which is a necessary condition for a local minimum, whereas what is implemented here is only a sufficient condition for a local minimum.

6 Implementation and results

Several implementations of AD have been written for C++ that take advantage of the operator overloading in the language, among them are [6, 14]. The approach of one of these implementations, Adol-C, is to build the computational graph each time that the function is evaluated, and then traverse it in order to get the desired derivatives. The advantage of this method is that conditional branches can be easily differentiated. Its disadvantage is that the repeated building of the computational graph can be time-consuming.

A second approach is to write a program that will differentiate functions rather than general computer programs, that is, a program will read in a procedure that describes the function $f(x)$, and then writes another procedure that will compute $f(x)$ and a specified number of derivatives. The advantage of this method is its speed, and it can then be linked into an optimization algorithm. Another advantage is that if a large amount of memory is required, paging can be handled much more efficiently. A disadvantage is that conditional branches are not as easily handled in this case.

A third approach, the one used here, is to compute the computational graph once and then retained in main memory. This has the advantage of speed since the computational graph does not need to be recomputed. A disadvantage is that memory cannot be handled very efficiently, and that conditional evaluations in the function cannot be handled very easily. However, since the test problems used no conditional branches, the drawbacks of this approach were not significant.

The test problems are as follows.

Problem 1. Extended Rosenbrock's function:

$$\begin{aligned} F(x) &= \sum_{i=1}^{n/2} 100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2 \\ X_0 &= (-1.2, 1, -1.2, 1, \ldots, -1.2, 1)^T \\ X^* &= (1, 1, 1, \ldots, 1)^T. \end{aligned}$$

Problem 2.

$$\begin{aligned} F(x) &= \sum_{i=2}^{n} i(2x_i^2 - x_{i-1})^2 + (x_1 - 1)^2 \\ X_0 &= (1, 1, \ldots, 1) \\ X^* &= \left(1, \frac{1}{2^{1/2}}, \frac{1}{2^{3/4}}, \frac{1}{2^{7/8}}, \ldots, \frac{1}{2^{(2^{n-1}-1)/2^{n-1}}}\right). \end{aligned}$$

The programs were run on an IBM-compatible system running an 80386 at 33 MHz with a math coprocessor using the GNU C++ version 2.3.3 running under OS/2 version 2.0. The computational graph was designed as a linked list that could be traversed in both the forward and reverse directions. The graph was created by overloading the standard operators (+, −, ∗, /) and the elementary functions (sin, cos, log, exp). The time to compute the computational graph was not included in the tables to follow. However, the time to compute the computational graph was not more than about 8 seconds, even when the number of variables exceeded 1024.

The forward AD method was implemented by overloading the operators as for the reverse method. The second form of the forward method was used, where only a vector-Hessian product was stored for the third element of the Taylor's series.

n	Forward AD			Reverse AD		
	Point	Hessian	Total	Point	Hessian	Total
4	17.40	0.66	18.34	9.88	0.43	10.60
8	57.90	6.09	63.78	18.43	1.59	20.75
16	303.50	40.87	384.31	35.97	6.18	43.57
32	886.28	326.40	1,222.72	80.15	23.53	121.44
64	3,854.35	2,569.37	6,491.06	140.35	99.28	299.13
128	14,464.19	20,672.75	35,478.31	269.19	403.84	1,019.28

Table 2: Time results with Rosenbrock's function.

All methods used the same truncated Newton optimization code for comparison. The linesearch as done by means of first checking to see whether $\alpha = 1$ was an acceptable stepsize according to the Wolf Test [13]. If it was not acceptable, the new point was found by fitting a cubic interpolant to $\phi(x)$, $\phi'(x)$, $\phi(1)$, and $\phi'(1)$, where $\phi(\alpha) = f(x + \alpha P)$ and P is the search direction. If this again does not find an acceptable stepsize, the stepsize was cut in half until an acceptable stepsize is found.

The point method was terminated when $\|\delta x\| < 10^{-15}$. In Tables 2 and 3, the columns "Point" and "Hessian" refer to the time required to find the potential optimum and the time to compute the Hessian, respectively. "Total" refers to the total time for finding a potential optimum, computing the Hessian, and verifying the solution. The time was obtained by asking for the system time just before the start of the optimization algorithm and then again just after the end of the algorithm. These two values were then subtracted appropriately to obtain the figures that are given. Unfortunately, these are real (wall clock) time elapsed rather than the actual CPU time. In order to try to get a good estimate, most tests were run three times, and the average figure is presented in the table.

The results in the tables suggest that when the size of the problem is doubled, the time to find a potential solution point when using the reverse method is approximately doubled. This is due to the fact that the reverse method is able to compute the gradient and the Hessian vector product for a constant times the work to compute the function alone, which is overall $O(N)$. For the verification portion of the algorithms, the entire Hessian must be computed. This requires $(5+16N)w(f)$ operations for an overall $O(N^2)$, which is reflected in the fact that as the size of the problem is doubled, the

n	Forward AD			Reverse AD		
	Point	Hessian	Total	Point	Hessian	Total
4	25.13	0.81	26.10	16.69	0.53	17.50
8	170.12	6.97	177.91	52.90	2.25	54.78
16	924.32	56.47	983.38	115.60	8.97	127.16
32	4,722.56	472.62	5,206.40	358.37	35.79	405.37
64	24,149.31	3,749.09	27,957.94	913.03	148.78	1,121.53

Table 3: Time results with Problem 2.

time to compute the Hessian goes up by a factor of four as can be seen above.

The time to find a potential solution approximately quadruples when the size of problem is doubled for the forward AD method. This is indicative of the fact that gradients and Hessian vector products are computed linearly in time giving an overall time of $O(N^2)$. For the verification portion of the algorithms, again, the entire Hessian must be computed. This requires $O(N^2)w(f)$ operations for an overall $O(N^3)$, which is reflected in the fact that as the size of the problem is doubled, the time to verify goes up by a factor of eight. The larger versions of the different examples were not run because of the amount of time that would be required.

Problem 2 turned out to be difficult for both methods to solve. It differs from the Rosenbrock's function in that as the size of the problem is increased, the Hessian matrix becomes more ill conditioned. With Rosenbrock's function, the condition number of the Hessian matrix is constant for any number of variables. This is what caused the methods to take considerably longer to Problem 2 and have erratic computation times as the number of variables increase. However, we still see the quadratic growth in time for computing the Hessian with the reverse AD versus the eightfold increase for computing the Hessian with forward AD. Also, reverse AD is again much superior to forward AD in the time to find a potential optimum.

Dixon and Price [4] compare the truncated Newton's method with forward AD to E04KDF, the NAG-modified Newton code, OPCG, the Hatfield Polytechnic OPTIMA library conjugate gradient subroutine [17], and OPVM, the OPTIMA variable metric code [17]. They found that in all problems considered, the truncated Newton's method with forward AD took approximately three times as long to solve as OPCG, was comparable to

OPVM, and was slower than E04KDF when the dimension of the problem was small. When the size of the problem was increased, OPVM and E04KDF were both slower than the truncated Newton's method.

Two problems related to the computation of the Hessian for the purposes of result verification remain, computing $K(X)$ core storage of the Hessian. It is obvious that the reverse method is superior when one wishes to compute 2 (and then 3) in the least amount of time for a large number of variables. For the second problem, if the Hessian is sparse, it cannot be assumed that the inverse is also sparse. This will greatly exceed the additional memory required for the reverse AD and so cannot be considered a disadvantage.

Finally, it should also be noted that the truncated Newton's method without verification and with verification falls apart in the case where the function has a singular Hessian at the solution. For the truncated Newton's method, one technique may be to use the higher order method when far from the solution and then revert to a lower order method such as a modified gradient descent when close to the solution or when convergence begins to stall. For result verification, it may be necessary to use a large number of decimal places of accuracy to invert the Hessian accurately when close to the solution. This may not be feasible and is an area for further research.

When using the truncated Newton's method or any other optimization method, AD should be considered whenever gradients or Hessian are required. The major drawback to reverse AD is the memory requirement but this is not significant if a full Hessian is required. For interval analysis, AD is a perfect match since the goals of the two ideas are similar. AD intends to give accurate derivatives that are devoid of truncation error, and interval analysis desires to give a tight bound of the roundoff error that remains. With the advent of languages such as C++, Pascal–XSC [11], and C-XSC [12], writing code that integrates these ideas is now easy.

7 Acknowledgments

The author is deeply indebted to several people in the final stages of this paper. The author would like to recognize the efforts of Dr. Weldon Lodwick of the University of Colorado at Denver, Dr. George Corliss of Marquette University, and Ms. Gail Pieper of Argonne National Laboratory.

References

[1] Bischof, C., Carle, A., Corliss, G., Griewank, A., and Hovland, P. *ADIFOR: Generating derivative codes from Fortran programs.* Scientific Programming **1** (1) (1992), pp. 1–29.

[2] Dembo, R. S. and Steinhaug, T. *Truncated-Newton algorithms for large-scale optimization.* Mathematical Programming **26** (1982).

[3] Dennis, J. and Schnabel, R. *Numerical methods for unconstrained optimization and nonlinear equations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[4] Dixon, L. C. W. and Price, R. C. *The truncated Newton method for sparse unconstrained optimization using automatic differentiation.* J. Opt. Theory and Appl. **60** (2) (1989), pp. 261 ff.

[5] Fletcher, R. *Practical methods of optimization.* John Wiley & Sons, Inc., Somerset, New Jersey, 1988 (2nd ed.).

[6] Griewank, A., Judes, D., Srinivasan, J., and Charles, T. *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++.* To appear in ACM Trans. Math. Software. Also appeared as Preprint MCS–P180-1190, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., November 1990.

[7] Griewank, A. *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation.* Optimization Methods & Software **1** (1) (1991), pp. 35–54.

[8] Hager, W. *Applied numerical linear algebra.* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[9] Golub, G. H. and Van Loan, C. F. *Matrix computations.* The Johns Hopkins University Press, Baltimore, 1983.

[10] Iri, M. and Kubota, K. *Methods of fast automatic differentiation and applications.* Research Memorandum RMI 87–02, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 1987.

[11] Klatte, R., Kulisch, U., Neaga, M., Ratz, D., and Ullrich, Ch. *PASCAL–XSC – language reference with examples.* Springer-Verlag, Heidelberg, 1992.

[12] Klatte, R., Kulisch, U., Lawo, C., Rauch, M., and Wiethoff, A. *C–XSC — a C++ class library for extended scientific computing.* Springer-Verlag, Heidelberg, 1993.

[13] Luenberger, D. G. *Linear and nonlinear programming.* Addison-Wesley Publishing Company, Reading, Mass., 1989 (2nd ed.).

[14] Micholetti, L. *MXYZPTLK: A practical, user-friendly C++ implementation of differential algebra: User's guide.* Technical Memorandum FN-535, Fermi National Accelerator Laboratory, Batavia, Ill., January 1990.

[15] Moore, R. E. *Methods and applications of interval analysis.* SIAM, Philadelphia, 1979.

[16] Moré, J. J. and Sorensen, D. C. *Newton's method.* In: Golub, G. H. (ed.) "Studies in numerical analysis", The Mathematical Association of America, 1984, pp. 29–82.

[17] *Optima manual.* Numerical Optimization Centre, Hatfield Polytechnic, Hatfield, Hertfordshire, England, 1984.

[18] Ostrovski, G. M., Wolin, Ju. M., and Borisov, W. W. *Über die Berechnung von Ableitungen.* Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg **13** (4) (1971), pp. 382–384.

[19] Ratz, D. *Automatische Ergebnisverifikation bei globalen Optimierungsproblemen.* Dissertation, Universität Karlsruhe, 1992.

[20] Speelpenning, B. *Compiling fast partial derivatives of functions given by algorithms.* Ph. D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, January 1980.

[21] Wengert, R. E. *A simple automatic derivative evaluation program.* Comm. ACM **7** (8) (1964), pp. 463–464.

2208 Williams St.
Denver, CO 80210
USA