JOURNAL OF OPTIMIZATION THEORY AND APPLICATIONS: Vol. 60, No. 2, FEBRUARY 1989

Truncated Newton Method for Sparse Unconstrained Optimization Using Automatic Differentiation[1]

L. C. W. DIXON[2] AND R. C. PRICE[3]

Abstract. When solving large complex optimization problems, the user is faced with three major problems. These are (i) the cost in human time in obtaining accurate expressions for the derivatives involved; (ii) the need to store second derivative information; and (iii), of lessening importance, the time taken to solve the problem on the computer. For many problems, a significant part of the latter can be attributed to solving Newton-like equations. In the algorithm described, the equations are solved using a conjugate direction method that only needs the Hessian at the current point when it is multiplied by a trial vector. In this paper, we present a method that finds this product using automatic differentiation while only requiring vector storage. The method takes advantage of any sparsity in the Hessian matrix and computes exact derivatives. It avoids the complexity of symbolic differentiation, the inaccuracy of numerical differentiation, the labor of finding analytic derivatives, and the need for matrix store. When far from a minimum, an accurate solution to the Newton equations is not justified, so an approximate solution is obtained by using a version of Dembo and Steihaug's truncated Newton algorithm (Ref. 1).

Key Words. Optimization, truncated Newton method, automatic differentiation.

1. Introduction

The problem to be considered is

$$\min F(x), \qquad x \in R^n,$$

[1] This paper was presented at the SIAM National Meeting, Boston, Massachusetts, 1986.

[2] Associate Dean (Research), School of Information Sciences, Hatfield Polytechnic, Hatfield, Hertfordshire, England.

[3] Research Student, Numerical Optimisation Centre, Hatfield Polytechnic, Hatfield, Hertfordshire, England.

where F is assumed to be a nonlinear, continuous, and twice differentiable function. We wish to find a point x^* such that, if $\delta > 0$, then

$$F(x^*) < F(x), \text{ for all } x\text{: } \|x - x^*\| < \delta.$$

If we denote $\nabla F(x)$ by $g(x)$, the gradient of F at the point x, and $\nabla^2 F(x)$ by $H(x)$, the Hessian at the point x, then a first-order necessary condition for a local minimum of F is that $g(x) = 0$, while the second-order necessary condition for a minimum is that the matrix $H(x)$ is positive semidefinite.

One of the best-known methods for unconstrained optimization is Newton's method, which approximates the objective function F at each iteration by a quadratic model ϕ and effectively minimizes ϕ. As

$$\phi(x+p) = F(x) + g(x)^T p + (1/2) p^T H(x) p,$$

then the first-order condition implies that ϕ' must equal 0, where

$$\phi'(x+p) = F'(x) + H(x)p = 0,$$

and hence the following system of equations must be solved at each iteration:

$$H(x)p = -\nabla F.$$

An unsafeguarded Newton algorithm therefore consists of the following steps.

Step 1. Calculate $F(x_k)$, $g(x_k)$, $H(x_k)$.
Step 2. Check if $\|g(x_k)\|_2 < \epsilon$; if so, stop.
Step 3. Solve $H(x_k)p_k = -g(x_k)$.
Step 4. Set $x_{k+1} = x_k + p_k$.
Step 5. Set $k = k+1$, and go to Step 1.

Newton's method can be shown to have second-order convergence. Basically, this is because it utilizes a quadratic model, which is minimized at each iteration. However, it is not globally convergent to the solution as the iteration can diverge from an arbitrary starting point. As the Newton equations have to be solved at each iteration, the method breaks down when $H(x_k)$ is singular.

For large problems, a major part of the cost can be attributed to the solution of the Newton equations. For these problems, the conjugate direction method is widely used, due to its low storage requirements.

The conjugate direction algorithm for solving $Au = b$ is as follows. First, choose $u_0 \in R^n$, and put $p_0 = r_0 = Au - b$.

Step 1. If $p_k = 0$, stop; x_k is the solution. Otherwise, go to Step 2.

Step 2. Compute

$$\alpha_k = r_k^T r_k / p_k^T A p_k,$$

$$u_{k+1} = u_k + \alpha_k p_k,$$

$$r_{k+1} = r_k + \alpha_k A p_k,$$

$$\beta_k = r_{k+1}^T r_{k+1} / r_k^T r_k,$$

$$p_{k+1} = -r_{k+1} + \beta_k p_k,$$

and go to Step 1.

The conjugate direction method is truncated in order to obtain an approximate solution to the Newton equations in Dembo and Steihaug's truncated Newton algorithm, which is described in Section 4.

One of the major problems, when solving large complex optimization problems, is the need to provide the first and second derivatives of the objective function. In this paper, we propose an adaptation of Rall's approach to automatic differentiation (Ref. 2), to calculate these quantities. We embed the automatic differentiation within the conjugate direction subroutine that solves the Newton equations. Within the conjugate direction subroutine in unconstrained optimization, the coefficient matrix in the set of linear equations being solved is the Hessian of the objective function at the current point. This matrix occurs only in the algorithm when it is multiplied by a search direction p_j, at the jth minor iteration. While it would be possible to calculate the Hessian H, store it, and then form Hp at each iteration, for large problems we have found it possible to reduce the storage required by introducing automatic differentiation at each iteration, to calculate the product Hp automatically for any known p. With this modification, no matrix store is required. In Ref. 3, results were reported on the efficiency of our implementation of Dembo and Steihaug's truncated Newton method. When far from a miinimum, an accurate solution to the Newton equations may not be justified. Dembo and Steihaug's method solves the equations by the conjugate direction method, but truncates the iteration when a required degree of accuracy is obtained.

2. Automatic Differentiation

Rall's automatic differentiation method (Ref. 2) can be used to calculate the value of a function, its gradient, and the Hessian at a known point $x \in R^n$. He represents each variable x_i as a triplet $X = (x_i, x_i', x_i'')$, where $x_i \in R$, $x_i' \in R^n$, and x_i'' is a symmetric, real $n \times n$ matrix. If we let T^n be the set of these triplets, then all the standard arithmetic operations can be defined in T^n. If

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

then we have

$$U+V=(u+v, u'+v', u''+v''), \tag{1}$$

$$U-V=(u-v, u'-v', u''-v''), \tag{2}$$

$$U * V=(uv, uv'+vu', uv''+u'v'^T+v'u'^T+vu''), \tag{3}$$

$$U / V=(u/v, (vu'-uv')/v^2, (v^2u''-v(v'u'^T+u'v'^T)+2uv'v'^T-uvv'')/v^3). \tag{4}$$

To represent an independent variable in T^n space, Rall lets

$$X_i \Rightarrow (x_i, e_i, 0),$$

where e_i is the ith column of the identity matrix, and 0 the zero matrix. Similarly, constants are represented by

$$C \Rightarrow (c, 0, 0).$$

The calculation of the value, gradient vector, and Hessian matrix of a rational function can be done simply in FORTRAN by defining subroutines that implement (1)-(4) and calling these in the function subroutine. Instead of using the separate triplet for constants, he defines rules for all the arithmetic operations when mixing constants and elements of T^n, namely:

$$C+U=U+C=(c+u, u', u''), \tag{5}$$

$$C-U=(c-u, -u', -u''), \tag{6}$$

$$U-C=(u-c, u', u''), \tag{7}$$

$$C * U=U \cdot C=(c \cdot u, c \cdot u', cu''), \tag{8}$$

$$C / U=(c/u, -cu'/u^2, (2cu'u'^T-cuu'')/u^3)), \tag{9}$$

$$U / C=(u/c, u'/c, u''/c). \tag{10}$$

Example 2.1. If we take the two-dimensional Rosenbrock function

$$F(x)=100(x_2-x_1^2)^2+(1-x_1)^2, \tag{11}$$

then the standard starting point $(-1.2, 1.0)$ becomes, in triplet form,

$$X_1=(-1.2, (1, 0)^T, 0),$$

$$X_2=(1.0, (0, 1)^T, 0).$$

Then, considering the function evaluation in T^n, we carry out the following steps (here, ST and ST1 are workspace triplets).

Step 1.	$\mathrm{ST}=X_1 * X_1$,	using	(3).
Step 2.	$\mathrm{ST}=(X_2-\mathrm{ST})$,	using	(2).
Step 3.	$\mathrm{ST}=\mathrm{ST} * \mathrm{ST}$,	using	(3).
Step 4.	$\mathrm{ST}=100 * \mathrm{ST}$,	using	(8).
Step 5.	$\mathrm{ST1}=1-X_1$,	using	(6).
Step 6.	$\mathrm{ST1}=\mathrm{ST1} * \mathrm{ST1}$,	using	(3).
Step 7.	$\mathrm{ST}=\mathrm{ST}+\mathrm{ST1}$,	using	(1).

ST now contains the triplet

$$\left(24.2,\ \begin{bmatrix}-215.6\\ -88.0\end{bmatrix},\begin{bmatrix}1330.0 & 480.0\\ 480.0 & 200.0\end{bmatrix}\right)=(F(\mathbf{x}), \nabla F(\mathbf{x}), \nabla^2 F(\mathbf{x})),$$

which is what would be obtained by differentiating the function analytically and evaluating $(F(\mathbf{x}), \nabla F(\mathbf{x}), \nabla^2 F(\mathbf{x}))$ at the point $(-1.2, 1.0)$.

In general, if $g: R \Rightarrow R$ is a twice differentiable function, then it can be extended to the mapping $g: T^n \Rightarrow T^n$ by the use of the chain rule

$$\begin{aligned} g(U) &= g(u, u', u'') \\ &= (g(u), g'(u) \cdot u', g'(u) \cdot u'' + g''(u) u' u'^T). \end{aligned}$$

So, for example,

$$\begin{aligned} \sin(U) &= \sin(u, u', u'') \\ &= (\sin(u), \cos(u) \cdot u', \cos(u) \cdot u'' - \sin(u) u' u'^T). \end{aligned}$$

An interesting mapping to consider is

$$g = \exp(f(x)),$$

since

$$g_i' = (\partial f(x)/\partial(x_i)) \exp(f(x)),$$

and so

$$g' = \nabla f(x) \exp(f(x)), \tag{12}$$

and since

$$\begin{aligned} g_i'' = {} & (\partial f(x)/\partial(x_i)) \cdot (\partial f(x)/\partial(x_i)) \cdot \exp(f(x)) \\ & + \exp(f(x))(\partial^2 f(x)/\partial(x_i)^2), \end{aligned}$$

and so

$$g'' = (\nabla f(x)^2 \exp(f(x)) + (\nabla^2 f(x)) \exp(f(x))). \tag{13}$$

To differentiate automatically this function, we must first differentiate automatically $f(x)$ to get $\nabla f(x)$ and $\nabla^2 f(x)$, and then substitute these values into (11) and (12).

Rall states in Ref. 2 that automatic differentiation is suited to languages such as ADA and PASCAL-SC, which permit the introduction of data types and additional definitions of the standard operator symbols to manipulate such types. However, when implemented in FORTRAN, as in this paper, the objective function must be rewritten as a sequence of calls to subroutines.

3. Adaptation of Automatic Differentiation to Avoid Storing Matrices

When using the conjugate direction (or any alternative indirect method) to solve $Ax = b$, the coefficient matrix A occurs only when multiplied by a vector p at any point in the algorithm. In an optimization algorithm, the coefficient matrix in the Newton equations is the Hessian of the objective function at the current point. Therefore, the Hessian multiplied by a vector p is required. This can be obtained by automatic differentiation without storing the coefficient matrix. In our modification, we change the third element of the triplet to a vector quantity by multiplying by a vector p. So each x_i is held as

$$(x_i, e_i, 0p) = (x_i, e_i, 0),$$

and all the rules for the arithmetic operations are redefined appropriately. The new formulation is

$$\begin{aligned}(U+V) &= (u, u', u''p) + (v, v', v''p)\\ &= (u+v, u'+v', u''p+v''p), \end{aligned}\tag{14}$$

$$\begin{aligned}(U-V) &= (u, u', u''p) - (v, v', v''p)\\ &= (u-v, u'-v', u''p-v''p), \end{aligned}\tag{15}$$

$$\begin{aligned}(U*V) &= (u, u', u''p) * (v, v', v''p)\\ &= (u\cdot v, u\cdot v' + v\cdot u', uv''p\\ &\quad + u'v'^Tp + v'u'^Tp + vu''p), \end{aligned}\tag{16}$$

$$\begin{aligned}(U/V) &= (u, u', u''p)/(v, v', v''p)\\ &= (u/v, (vu'-uv')/v^2,\\ &\quad (v^2u''p - v(v'u'^Tp + u'v'^Tp) + 2uv'v'^Tp - uvv''p)/v^3). \end{aligned}\tag{17}$$

It can be seen that many of the quantities in the third part of the triplets are scalars arising from the inner products, multiplied by vectors.

4. Sparse Automatic Differentiation

When the automatic differentiation described in the previous section was applied, it was found to be inefficient for large problems when compared with Dembo and Steihaug's differencing approach (Ref. 1), devised to approximate the product *Hp*. However, a further adaptation to the automatic differentiation approach can be implemented easily, which takes advantage of any sparsity and structure in the Hessian matrix, and thus avoids many unnecessary calculations.

In the previous approach, all vectors were stored as length n; however, if the operation under consideration is x_1 multiplied by x_2 within a large problem, then only the first two elements of the vectors are needed, since the other elements do not make any contribution to the gradient or Hessian of $x_1 * x_2$. A similar consideration applies to most operations that occur in the calculation of a function value. Therefore, by passing the correct suffices (1 and 2 in this example) through to the multiplication subroutine, the work can be reduced substantially. This approach implies that each objective function must be subdivided into a sequence of subfunction calculations. After the function, gradient, and *Hp* of each subfunction have been evaluated, they have to be added to the workspace triplet storing the function value, gradient, and *Hp* of the overall objective function. This addition will probably have to be performed in n-space; however, this work is minimal, since the automatic differentiation for addition involves simply $2n+1$ additions. The triplet corresponding to each subfunction can be calculated with vectors of reduced length, providing the subfunction involves only a limited number of the optimization variables. Results using this second adaptation, reported in the next section, show it to be far more efficient than working always in full n-space. It is suitable particularly for highly structured problems that occur in finite-element optimization (Ref. 4) and parameter model fitting (Ref. 5).

5. Truncated Newton Method

In large-scale optimization, the solution of the Newton equations may be expensive, and so Dembo and Steihaug (Ref. 1) introduced the idea of calculating an approximate solution. When far from a minimum, it may not be justified to solve the Newton equations accurately. Dembo suggested solving the equations by the conjugate direction method and truncating the iteration before an accurate solution is obtained. In large-scale optimization, where the Hessian matrix is likely to be sparse, an indirect method that does not require an $n \times n$ matrix store has advantages when solving the Newton equations. The conjugate direction method is one such indirect algorithm and is chosen, since it reduces steadily the quadratic model ϕ. The obvious problem is to decide how accurate a solution is required at each major iteration. One measure used to decide this is the size of the relative residual $r_j^T r_j / g_k^T g_k$ at the jth minor iteration and kth major iteration. When this quantity is less than $\min(0.1/k^2, g_k^T g_k)$, Dembo truncates the conjugate direction iteration and starts the next major iteration. Therefore, the relative accuracy is tightened as the iteration number increases and as the norm of the gradient vector is reduced, i.e., as a solution to the optimization problem is approached.

In Ref. 3, we report details of our implementation of a truncated Newton method that introduces a trust region and a cone of acceptable directions to guarantee convergence. In that paper, results are given that demonstrate that the method is more efficient than an ordinary safeguarded Newton code, a variable metric code, and a conjugate gradient code all using analytically programmed derivatives.

6. Numerical Results

Since the truncated Newton method with sparse automatic differentiation requires only vector storage, it is in direct competition with first-order conjugate gradient methods. To indicate the efficiency of the new method, we report now the numerical results for solving various test problems using E04KDF, the NAG modified Newton code; OPCG, the Hatfield Polytechnic OPTIMA library conjugate gradient subroutine (Ref. 8); OPVM, the OPTIMA variable metric code (Ref. 8); and TNEWAD, the truncated Newton code with automatic differentiation.

The test problems are given in the Appendix. The following results for the test problems were obtained on a DEC-1901 system at the Hatfield Polytechnic. All codes used a stopping criterion $\|g\|_2 \leq 10^{-5}$. Tables 1-7 contain the number of iterations $N(\text{IT})$, the number of function calls $N(\text{F})$, the number of gradient calls $N(\text{G})$, and the number of effective function evaluations $N(\text{EFE})$, calculated from the formula

$$N(\text{EFE}) = N(\text{F}) + nN(\text{G}).$$

The CPU time is given in seconds.

Table 1. Results for TNEWAD.

Problem	n	N(IT1)	N(IT2)	N(F)	N(G)	N(EFE)	CPU time
1	2	11	9	17	20	57	0.53
2	3	23	11	21	34	123	0.96
3	4	59	20	39	79	355	2.56
4	2	35	21	41	56	153	0.24
5	2	28	21	41	49	139	0.29
6	4	149	43	85	192	853	1.19
7	4	51	18	35	69	311	0.46
8	10	35	21	41	56	601	0.47
8	20	35	21	41	56	1161	0.71
9	30	68	10	19	78	2359	1.78

n, number of variables; N(IT1), number of minor iterations; N(IT2), number of major iterations; N(F), number of function calls; N(G), number of gradient calls; and N(EFE), number of effective function evaluations.

All of the codes, apart from TNEWAD, required the user to provide the derivatives. If second-order derivative information was required for a particular algorithm, it would be calculated using a difference formula. However, TNEWAD used the automatic differentiation to calculate both first-order and second-order derivative information.

On the small problems reported above, the algorithms all performed reasonably well and the differences in CPU time must be considered marginal. On these problems, TNEWAD generally took more CPU time but did surprisingly well, since on these problems there is less scope for truncation and little chance to gain significantly from the structure of the function. To see the effect of increased dimensionality and structure, Table 6 shows

Table 2. Results for OPCG (analytic derivatives).

Problem	n	F	N(IT)	N(F)	N(G)	N(EFE)	CPU time
1	2	2.0E−13	16	43	16	75	0.43
2	3	6.8E−12	22	52	22	118	0.52
3	4	7.5E−10	50	114	50	314	0.99
4	2	3.0E−14	35	108	35	178	0.37
5	2	7.3E−10	15	60	15	90	0.25
6	4	1.8E−13	59	139	59	375	0.34
7	4	4.5E−9	65	150	65	410	0.45
8	10	1.6E−6	33	93	33	429	0.54
8	20	9.1E−13	40	139	40	939	1.00
9	30	2.3E−8	118	316	118	3856	1.48

For abbreviation definitions, see Table 1.

Table 3. Results for OPCGAD (automatic derivatives).

Problem	n	F	N(IT)	N(F)	N(G)	N(EFE)	CPU time
1	2	2.0E−13	16	43	16	75	0.77
2	3	6.8E−12	22	52	22	118	1.39
3	4	7.5E−10	50	114	50	314	2.75
4	2	3.0E−14	35	108	35	178	0.60
5	2	7.3E−10	15	60	15	90	0.39
6	4	1.8E−13	59	139	59	375	0.48
7	4	4.5E−9	65	150	65	410	0.78
8	10	1.6E−6	33	99	33	429	0.82
8	20	9.1E−13	40	139	40	939	1.91
9	30	2.3E−8	118	316	118	3856	3.71

For abbreviation definitions, see Table 1.

Table 4. Results for OPVM.

Problem	n	F	N(IT)	N(F)	N(G)	N(EFE)	CPU time
1	2	0.2E−10	26	43	26	95	0.44
2	3	0.3E−12	14	21	14	63	0.36
3	4	0.5E−11	25	37	25	137	0.52
4	2	0.3E−11	31	49	31	111	0.29
5	2	0.5E−7	23	43	23	89	0.20
6	4	0.3E−13	74	136	74	432	0.35
7	4	0.4E−8	28	41	28	161	0.22
8	10	0.9E−6	81	143	81	953	0.73
8	20	0.3E−7	120	212	120	2612	3.22
9	30	0.4E−8	53	86	53	1676	2.72

For abbreviation definitions, see Table 1.

Table 5. Results for EO4KDF.

Problem	n	F	N(IT)	N(F)	N(G)	N(EFE)	CPU time
1	2	0.1E−10	9	11	29	98	0.37
2	3	0.3E−10	9	22	52	178	0.60
3	4	0.3E−12	13	26	82	354	0.87
4	2	0.6E−9	21	28	70	168	0.26
5	2	0.9E−8	21	22	66	154	0.28
6	4	0.3E−11	37	54	202	862	0.42
7	4	0.2E−9	17	18	90	378	0.31
8	10	0.3E−10	14	41	181	1851	0.53
8	20	0.2E−8	13	52	312	6292	1.21
9	30	F	F	F	F	F	F

For abbreviation definitions, see Table 1.

Table 6. Results for Problem 7 in higher dimensions.

Problem	n	Code	F	N(F)	N(G)	N(EFE)	CPU time
7	60	OPVM	0.56E−8	282	160	9882	28.12
7	80	OPVM	0.19E−7	290	158	12930	46.30
7	60	EO4KDF	0.23E−13	42	1302	78162	15.31
7	80	EO4KDF	0.88E−13	43	1643	1.3E+5	32.84
7	60	OPCG	1.75E−8	115	49	3055	1.53
7	80	OPCG	8.58E−8	137	60	4937	2.93
7	60	OPCGAD	1.75E−8	115	49	3055	3.47
7	80	OPCGAD	8.58E−8	137	60	4937	5.45
7	60	TNEWAD	0.59E−7	39	83	5019	6.71
7	80	TNEWAD	0.83E−7	39	86	6919	9.24

For abbreviation definitions, see Table 1.

the results for the five codes on the versions of Problem 7 with dimensions 60 and 80.

When the times for OPCG and TNEWAD are compared, it must be noted that TNEWAD calculates the derivatives automatically at each minor iteration and therefore incurs extra cost, but does not require analytic derivatives. Although TNEWAD took about three times as long to solve each problem, it does not require the human time needed to calculate the analytic derivatives. Results for OPCG using automatic derivatives are shown in Table 3.

In Section 4, it was stated that automatic differentiation working in n-space was found to be inefficient. Table 7 illustrates the need for the sparse version of automatic differentiation.

Table 7. Sparse form (SP) versus nonsparse form (NS).

Problem	SP/NS	n	F	N(F)	N(G)	N(EFE)	CPU time
7	SP	60	0.59E−7	39	83	5019	6.71
7	SP	80	0.83E−7	39	86	6919	9.24
8	SP	20	0.00	41	56	1161	0.71
9	SP	30	0.69E−12	19	78	2359	1.78
7	NS	60	0.59E−7	39	83	5019	31.86
7	NS	80	0.83E−7	39	86	6919	53.83
8	NS	20	0.00	41	56	1161	4.46
9	NS	30	0.69E−12	19	78	2359	7.41

For abbreviation definitions, see Table 1.

As expected, the advantages of the sparse version of automatic differentiation are more apparent on the larger problems. For the $m = 60$ version of Problem 8, the sparse form took only 6.71 seconds; however, the nonsparse form took 31.86 sec. When n was increased to 80 for the same problem, the sparse form took 9.24 sec, compared with 53.83 sec for the nonsparse form.

Since OPVM and EO4KDF proved to be uncompetitive with OPCG and TNEWAD for larger problems, OPCG and TNEWAD were run on Problems 8 and 9 with n increased to 2000. For Problem 8, OPCG took 393.47 sec compared with TNEWAD's time of 250.13 sec; for Problem 9, OPCG failed to find an acceptable minimum while TNEWAD solved the problem in 1026.15 sec.

To investigate the overheads associated with automatic differentiation compared with analytic derivatives, OPCG was run on the same test problems with automatic differentiation (OPCGAD). If the results in Tables 2 and 3 for OPCG and OPCGAD are compared, it is not too surprising that the two sets of results have identical numbers of iterations, function calls, and gradient calls, since they are both computing the same derivatives. However, the times for OPCGAD are only ~ 1.5 to 2 times more than those for analytic derivatives. When the user time spent differentiating analytically the objective function is taken into account, the times for OPCGAD must be viewed as very competitive. If the times for OPCGAD and TNEWAD are compared, TNEWAD takes less time to solve each problem, apart from Problem 7.

7. Conclusions

In this paper, we have demonstrated that:

(i) Rall's method of automatic differentiation can be used to save the human time required to obtain analytic expressions for derivatives in unconstrained optimization;

(ii) if we combine automatic differentiation with Dembo's truncated Newton method, then we require only the product Hp, which can be obtained by a modification of Rall's method that requires only a vector store;

(iii) when the objective function is structured and the Hessian matrix is sparse, then a more efficient automatic differentiation method is implemented readily using the local variable vector for each subfunction.

Computer results are reported that indicate that, even when implemented inefficiently in FORTRAN, the penalty for automatic differentiation is only a factor of 3 on computer time. This factor must of course be weighed against the considerable saving of human time in formulating analytic derivative formulas of complicated industrial problems.

8. Appendix: Test Problems

The test problems used were as follows.

Problem 1

$$F(x)=\sum_{i=1}^{10}[\exp(-x_1 z_i)-5\exp(-x_2 z_i)-\exp(-z_i)+5\exp(-10 z_i)]^2, \qquad z_i=i/10,$$

$$x(0)=(1,2)^T.$$

Problem 2

$$F(x)=\sum_{i=1}^{10}[\exp(-x_1 z_i)-x_3\exp(-x_2 z_i)-\exp(-z_i)+5\exp(-10 z_i)]^2, \qquad z_i=i/10,$$

$$x(0)=(1,2,1)^T.$$

Problem 3

$$F(x)=\sum_{i=1}^{10}[x_3\exp(-x_1 z_i)-x_4\exp(-x_2 z_i)-\exp(z_i)+5\exp(-10 z_i)]^2, \qquad z_i=i/10,$$

$$x(0)=(1,2,1,1)^T.$$

Problem 4. Rosenbrock's function:

$$F(x)=100(x_1^2-x_2)^2+(1-x_1)^2,$$

$$x(0)=(-1.2,1.0)^T.$$

Problem 5

$$F(x)=100(x_1^2-x_2)^8+(1-x_1)^8,$$

$$x(0)=(-1.2,1.0)^T.$$

Problem 6. Wood's function:

$$F(x) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2 + 90(x_4 - x_3^2)^2 + (1 - x_3)^2$$
$$+ 10.1[(x_2 - 1)^2 + (x_4 - 1)^2] + 19.8(x_2 - 1)(x_4 - 1),$$
$$x(0) = (-3, -1, -3, -1)^T.$$

Problem 7. Powell's function:

$$F(x) = \sum_{j=1}^{n/4} (x_{4j-3} + 10x_{4j-2})^2 + 5(x_{4j-1} - x_{4j})^2$$
$$+ (x_{4j-2} - 2x_{4j-1})^4 + 10(x_{4j-3} - x_{4j})^4,$$
$$x(0) = (3, -1, 0, 1, 3, -1, 0, 1, \ldots)^T.$$

Problem 8. Extended Rosenbrock's function:

$$F(x) = \sum_{j=1}^{n/2} 100(x_{2j} - x_{2j-1}^2)^2 + (1 - x_{2j-1})^2,$$
$$x(0) = (-1.2, 1.0, -1.2, 1.0, \ldots)^T.$$

Problem 9

$$F(x) = \sum_{i=1}^{n} i(2x_i^2 - x_{i-1})^2 + (x_1 - 1)^2,$$
$$x(0) = (1, 1, 1, 1, 1, 1, 1, \ldots, 1)^T.$$

References

1. DEMBO, R., and STEIHAUG, T., *Truncated Newton Methods for Large Scale Optimization,* Mathematical Programming, Vol. 26, pp. 190–212, 1982.
2. RALL, L. B., *Automatic Differentiation: Techniques and Applications,* Springer-Verlag, Berlin, Germany, 1981.
3. DIXON, L. C. W., and PRICE, R. C., *Numerical Experience with the Truncated Newton Method,* Numerical Optimisation Centre, Hatfield Polytechnic, Technical Report No. 169, 1986.
4. DIXON, L. C. W., DOLAN, P. D., and PRICE, R. C., *Finite Element Optimization,* Proceedings of the Meeting on Simulation and Optimization of Large Systems, Reading, England, 1986, edited by A. J. Osiadacz, Oxford Science Publications, Oxford, England, 1988.
5. POPE, J., and SHEPHERD, J., *On the Integrated Analysis of Catch-at-Age and Groundfish Survey or CPUE Data,* Ministry of Fisheries and Food, Lowestoft, England, 1984.

6. DEMBO, R., EISENSTAT, S. C., and STEIHAUG, T., *Inexact Newton Methods*, SIAM Journal on Numerical Analysis, Vol. 19, pp. 400–408, 1982.
7. STEIHAUG, T., *The Conjugate Gradient Method and Trust Regions in Large Scale Optimization*, SIAM Journal on Numerical Analysis, Vol. 20, pp. 626–637, 1983.
8. ANONYMOUS, *Optima Manual*, Numerical Optimisation Centre, Hatfield Polytechnic, Hatfield, Hertfordshire, England, 1984.