ADIFOR Working Note #2:

Using ADIFOR to Compute Dense and Sparse Jacobians*

Christian H. Bischof
bischof@mcs.anl.gov

Paul Hovland
hovland@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439–4801

Technical Memorandum MCS–TM–158

October 1991

Abstract. ADIFOR is a source translator that, given a collection of Fortran subroutines for the computation of a "function," produces Fortran code for the computation of the derivatives of this function. More specifically, ADIFOR produces code to compute the matrix-matrix product JS, where J is the Jacobian of the "function" with respect to the user-defined independent variables, and S is the composition of the derivative objects corresponding to the independent variables. This interface is flexible; by setting $S = x$, one can compute the matrix-vector product Jx, or by setting $S = I$, one can compute the whole Jacobian J. Other initializations of S allow one to exploit a known sparsity structure of J. This paper illustrates the proper initialization of ADIFOR-generated derivative codes and the exploitation of a known sparsity structure of J.

1 Introduction

ADIFOR (**A**utomatic **D**ifferentiation **i**n **For**tran) is a tool for the automatic generation of derivative codes from user-supplied Fortran subroutines. That is, given a collection of Fortran subroutines for the computation of a "function," **ADIFOR** produces Fortran code for the computation of the derivatives of this function. **ADIFOR** differs from previous approaches to automatic differentiation (see [13] for a recent survey) in several ways:

Generality: The "function" can be composed of many subroutines, and these subroutines may communicate via parameter lists and/or common blocks. In general, almost all of Fortran-77 is supported.

Portability: **ADIFOR** produces vanilla Fortran-77 code.

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38 and through NSF Cooperative Agreement No. CCR-8809615.

Efficiency: While primarily based on the so-called forward mode of automatic differentiation, **ADIFOR** uses the so-called reverse mode to process assignment statements with composite right-hand sides [9,10]. In addition to saving storage, this approach significantly enhances performance.

Parallelism and Vectorization: The code produced by **ADIFOR** respects the data-flow structure of the original program. That is, if the code vectorizes and parallelizes well, so does the **ADIFOR**-generated derivative code. If anything, the derivative code offers more scope for vectorization and parallelization because of the addition of another 'data parallel' dimension in derivative objects.

Extensibility: The fact that **ADIFOR** employs a consistent subroutine-naming scheme allows the user to supply his or her own derivative routines. In this fashion, the user can exploit domain-specific knowledge, utilize vendor-supplied libraries, and speed up computational bottlenecks.

Ease of Use: **ADIFOR** requires the user to supply the Fortran source code for the subroutine representing the function to be differentiated and for all lower-level subroutines. The user then selects the variables (either in parameter lists or in common blocks) that correspond to the independent and dependent variables. By using the powerful interprocedural analysis tools of the ParaScope programming environment [3], **ADIFOR** then automatically determines which other variables throughout the program must have derivative information associated with them.

Interactive Interface: An X-windows interface for **ADIFOR** (called **xadifor**) is also provided. **Xadifor** makes it easy for the user to set up the problem and to rerun **ADIFOR** if changes in the code for the target function require a new translation.

ADIFOR is applied to the code of the subroutine that corresponds to the subroutine we wish to differentiate (`foo`, say), and to all subroutines called directly or indirectly from `foo`. Let us assume that `foo` describes a function $f : [x, w] \mapsto [y, z]$ and that we are interested in the derivatives $\frac{\partial y}{\partial x}$; that is, the input variable w is treated as constant, and the output variable z is irrelevant. If this is the case, we call x the **independent variable** and y the **dependent variable**. We are aware of the fact that the terms "dependent," "independent," "variable," and "parameters" are used in many different contexts, yet we found that this terminology corresponds best to our mathematical idea of derivatives, since we will compute derivatives of the "dependent" variables with respect to the "independent" ones.

We require the user to tell **ADIFOR** the names of the independent variables and the names of the dependent variables. In many codes, dependent and independent variables may share storage. For example, on entry to `foo`, array `A` may be initialized to what we consider mathematically to be the value of the independent variable x, and during the course of executing `foo`, y will be written into `A`. This poses no problem for **ADIFOR**. It produces a subroutine named `g$foo$<n>` (where `<n>` is some number encoding which variables were dependent and independent), which computes the first derivatives of the function computed by `foo`, as well as `foo` itself.

To propagate derivative information in the forward mode, we have to associate derivative objects with the independent variables, the dependent variables, and all those program variables whose value depends (directly or indirectly) on an independent variable and that influence the value of a dependent variable. That is, if `x` is independent, `y` is dependent, and `z` depends on `x` and `y` depends on `z`, then `z` also needs a derivative object. A variable with which we associate a derivative object is called an **active variable**, any other variable is a **passive variable**. Dependent and independent variables are always active, and `integer` variables are always passive.

The user need not specify as passive or active variables local to `foo` or parameters or local variables in routines called by `foo`. Using the powerful interprocedural analysis tools available in the ParaScope environment [3], we can determine all active variables from a definition of the independent and dependent ones. This allows for a simple user interface that corresponds as much as possible to the mathematical intuition underlying `foo`.

The derivative codes produced by **ADIFOR** have a gradient object associated with every active variable. The convention is to associate a gradient `g$<var>` of leading dimension `ldg$<var>` with variable `<var>`. The calling sequence of `g$foo$<n>` is derived from that of `foo` by inserting an argument `g$p` denoting the length of the gradient vectors as the first argument, and then copying the calling sequence of `foo`, inserting `g$<var>` and `ldg$<var>` after every active variable `<var>`. Passive variables are left untouched.

In its simplest form, the functionality of **ADIFOR** can be summarized as follows:

> In general, if `x(1:n)` are the independent variables, and `y(1:m)` the dependent ones, then `g$x` is a `g$p` $\times n$ matrix (`ldg$x` $\geq$ `g$p`), and `g$y` is a `g$p` $\times m$ matrix (`ldg$y` $\geq$ `m`). The functionality of `g$foo` is: Given input values `x` and `g$x`, this subroutine computes $y = foo(x)$, and $g\$y = (foo'(x)g\$x^T)^T$.

In this paper, we shall not concern ourselves with the way code is generated or with the input provided to **ADIFOR**. For these details, the reader is referred to [2]. Even though the **ADIFOR** interface conceptually never changes, the actual initialization of **ADIFOR** code may vary depending on context. We focus instead on the proper and efficient use of **ADIFOR**-generated codes through detailed examination of the following cases:

- Dense Jacobian, one independent, one dependent variable
- Dense Jacobian, multiple independent, multiple dependent variables
- Sparse Jacobian, one independent, one dependent variable
- Sparse Jacobian, two independent variables, one dependent variable
- Partially separable functions

In most of these cases, a "variable" denotes an array; thus, we shall be dealing with vector-valued functions.

2 Case 1: Dense Jacobian, one independent, one dependent variable

Our first example is adapted from Problem C2 in the STDTST set of test problems for stiff ODE solvers [7] and was brought to our attention by George Corliss. The routine **FCN2** computes the right-hand side of a system of ordinary differential equations $y' = yp = f(x, y)$ by calling a subordinate routine **FCN**:

```
C  File:  FCN2.f
      SUBROUTINE FCN2(M,X,Y,YP)
      INTEGER  N
      DOUBLE PRECISION X, Y(M), YP(M)
      INTEGER        ID, IWT
      DOUBLE PRECISION W(20)
      COMMON         /STCOM5/W, IWT, N, ID

      CALL FCN(X,Y,YP)
      RETURN
      END


C  File:  FCN.f

      SUBROUTINE FCN(X,Y,YP)

C     ROUTINE TO EVALUATE THE DERIVATIVE F(X,Y) CORRESPONDING TO THE
C     DIFFERENTIAL EQUATION:
C                    DY/DX = F(X,Y) .
C     THE ROUTINE STORES THE VECTOR OF DERIVATIVES IN YP(*). THE
C     DIFFERENTIAL EQUATION IS SCALED BY THE WEIGHT VECTOR W(*)
C     IF THIS OPTION HAS BEEN SELECTED (IF SO IT IS SIGNALLED
C     BY THE FLAG IWT).

      DOUBLE PRECISION X, Y(20), YP(20)
      INTEGER        ID, IWT, N
      DOUBLE PRECISION W(20)
      COMMON         /STCOM5/W, IWT, N, ID
      DOUBLE PRECISION SUM, CPARM(4), YTEMP(20)
      INTEGER        I, IID
      DATA           CPARM/1.D-1, 1.D0, 1.D1, 2.D1/

      IF (IWT.LT.0) GO TO 40
      DO 20 I = 1, N
        YTEMP(I) = Y(I)
        Y(I) = Y(I)*W(I)
   20 CONTINUE
   40 IID = MOD(ID,10)
```

```
C     ADAPTED FROM PROBLEM C2
      YP(1) = -Y(1) + 2.D0
      SUM = Y(1)*Y(1)
      DO 50 I = 2, N
         YP(I) = -10.0D0*I*Y(I) + CPARM(IID-1)*(2**I)*SUM
         SUM = SUM + Y(I)*Y(I)
   50 CONTINUE

      IF (IWT.LT.0) GO TO 680
      DO 660 I = 1, N
         YP(I) = YP(I)/W(I)
         Y(I) = YTEMP(I)
  660 CONTINUE
  680 CONTINUE
      RETURN
      END
```

Most software for the numerical solution of stiff systems of ODEs requires the user to supply a subroutine for the Jacobian of f with respect to y. Such a subroutine can easily be generated by **ADIFOR**. For the purposes of automatic differentiation, the vector `Y` is the independent variable, and the vector `YP` is the dependent variable. Then **ADIFOR** produces

```
      subroutine g$fcn2$6(g$p$, m, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
C
C       ADIFOR: runtime gradient index
        integer g$p$
C       ADIFOR: translation time gradient index
        integer g$pmax$
        parameter (g$pmax$ = 20)
C       ADIFOR: gradient iteration index
        integer g$i$
C
        integer ldg$y
        integer ldg$yp
        integer n
        double precision x, y(m), yp(m)
        integer id, iwt
        double precision w(20)
        common /stcom5/ w, iwt, n, id
C
C       ADIFOR: gradient declarations
        double precision g$y(ldg$y, m), g$yp(ldg$yp, m)
        if (g$p$ .gt. g$pmax$) then
          print *, "Parameter g$p is greater than g$pmax."
          stop
        endif
        call g$fcn$6(g$p$, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
```

```
      return
      end


      subroutine g$fcn$6(g$p$, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
C
C       ADIFOR: runtime gradient index
        integer g$p$
C       ADIFOR: translation time gradient index
        integer g$pmax$
        parameter (g$pmax$ = 20)
C       ADIFOR: gradient iteration index
        integer g$i$
C
        integer ldg$y
        integer ldg$yp
C       ROUTINE TO EVALUATE THE DERIVATIVE F(X,Y) CORRESPONDING TO THE
C       DIFFERENTIAL EQUATION:
C       DY/DX = F(X,Y) .
C       THE ROUTINE STORES THE VECTOR OF DERIVATIVES IN YP(*). THE
C       DIFFERENTIAL EQUATION IS SCALED BY THE WEIGHT VECTOR W(*)
C       IF THIS OPTION HAS BEEN SELECTED (IF SO IT IS SIGNALLED
C       BY THE FLAG IWT).
        double precision x, y(20), yp(20)
        integer id, iwt, n
        double precision w(20)
        common /stcom5/ w, iwt, n, id
        double precision sum, cparm(4), ytemp(20)
        integer i, iid
        data cparm /1.d-1, 1.d0, 1.d1, 2.d1/
C
C       ADIFOR: gradient declarations
        double precision g$y(ldg$y, 20), g$yp(ldg$yp, 20)
        double precision g$sum(g$pmax$), g$ytemp(g$pmax$, 20)
        if (g$p$ .gt. g$pmax$) then
          print *, "Parameter g$p is greater than g$pmax."
          stop
        endif
        if (iwt .lt. 0) then
          goto 40
        endif
        do 99999, i = 1, n
C            ytemp(i) = y(i)
          do g$i$ = 1, g$p$
            g$ytemp(g$i$, i) = g$y(g$i$, i)
          enddo
          ytemp(i) = y(i)
```

```
C           y(i) = y(i) * w(i)
          do g$i$ = 1, g$p$
            g$y(g$i$, i) = w(i) * g$y(g$i$, i)
          enddo
          y(i) = y(i) * w(i)
20        continue
99999   continue
40      iid = mod(id, 10)
C       ADAPTED FROM PROBLEM C2
C       yp(1) = -y(1) + 2.d0
        do g$i$ = 1, g$p$
          g$yp(g$i$, 1) = -g$y(g$i$, 1)
        enddo
        yp(1) = -y(1) + 2.d0
C       sum = y(1) * y(1)
        do g$i$ = 1, g$p$
          g$sum(g$i$) = y(1) * g$y(g$i$, 1) + y(1) * g$y(g$i$, 1)
        enddo
        sum = y(1) * y(1)
        do 99998, i = 2, n
C           yp(i) = -10.0d0 * i * y(i) + cparm(iid - 1) * (2 ** i) * sum
          do g$i$ = 1, g$p$
            g$yp(g$i$, i) = cparm(iid - 1) * (2 ** i) * g$sum(g$i$) + -1
     *0.0d0 * i * g$y(g$i$, i)
          enddo
          yp(i) = -10.0d0 * i * y(i) + cparm(iid - 1) * (2 ** i) * sum
C           sum = sum + y(i) * y(i)
          do g$i$ = 1, g$p$
            g$sum(g$i$) = g$sum(g$i$) + y(i) * g$y(g$i$, i) + y(i) * g$y
     *(g$i$, i)
          enddo
          sum = sum + y(i) * y(i)
50        continue
99998   continue
        if (iwt .lt. 0) then
          goto 680
        endif
        do 99997, i = 1, n
C           yp(i) = yp(i) / w(i)
          do g$i$ = 1, g$p$
            g$yp(g$i$, i) = (1 / w(i)) * g$yp(g$i$, i)
          enddo
          yp(i) = yp(i) / w(i)
C           y(i) = ytemp(i)
          do g$i$ = 1, g$p$
            g$y(g$i$, i) = g$ytemp(g$i$, i)
          enddo
```

```
        y(i) = ytemp(i)
660       continue
99997   continue
680     continue
      return
    end
```

In accordance with the general policy outlined in § 1, the derivative objects `g$y` and `g$yp` are declared as matrices with 20 columns (since both `y` and `yp` were declared as vectors of length 20) and leading dimension `ldg$y` and `ldg$yp`, respectively. The parameter `g$p` denotes the actual length of the gradient objects in a call to `g$fcn2$6`. Since Fortran 77 does not allow dynamic memory allocation, derivative objects for local variables are statically allocated with leading dimension `pmax`, whose value was selected by the user during the invocation of **ADIFOR**. A variable and its associated derivative object are treated in the same fashion; that is, if `x` is a function parameter, so is `g$x`. Derivative objects corresponding to locally declared variables or variables in common blocks are declared locally or in common blocks as well.

Subroutine `g$fcn2$6` relates to the Jacobian

$$J_{yp} = \begin{pmatrix} \frac{\partial yp_1}{\partial y_1} & \dots & \frac{\partial yp_1}{\partial y_m} \\ \vdots & & \vdots \\ \frac{\partial yp_m}{\partial y_1} & \dots & \frac{\partial yp_m}{\partial y_m} \end{pmatrix}$$

as follows: Given input values for `g$p$, m, x, y, g$y, ldg$y,` and `ldg$yp`, the routine `g$fcn2$6` computes both `yp` and `g$yp`, where

$$\texttt{g\$yp(1:g\$p,1:m)} = (J_{yp}(\texttt{g\$y(1:g\$p,1:m)}^T))^T.$$

The superscript T denotes matrix transposition. The user must allocate `g$yp` and `g$y` with leading dimensions `ldg$yp` and `ldg$y` that are at least `g$p`. While the implicit transposition may seem awkward at first, this is the only way to handle assumed-size arrays (like `real a(*)`) in subroutine calls.

Assume that `m` and `g$p` are 20 and that `ldg$yp` and `ldg$y` are at least 20. Then we can compute the derivative matrix J_{yp} simply by initializing `g$y` to the identity:

```
**************
* Approach 1 *
**************
      DO 10 I = 1, M
         DO 5 J = 1, M
            G$Y(I,J) = 0.0D
  5      CONTINUE
         G$Y(I,I) = 1.0D0
 10   CONTINUE
      call g$fcn2$6(20, m, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
```

On exit from `g$fcn2$6`, the variable `g$yp` contains the transpose of the Jacobian J_{yp}.

Alternatively, we could have computed the Jacobian one column at a time:

```
**************
* Approach 2 *
**************
      DO 10 I = 1, M
*
*        initialize first row of G$Y to i-th unit vector
*
         DO 5 J = 1, M
            G$Y(1,J) = 0.0D
    5    CONTINUE
         G$Y(1,I) = 1.0D0
*
*        call ADIFOR-generated derivative code
*
         call g$fcn2$6(1, m, x, y, g$y, ldg$y, yp, g$yp, ldg$yp)
*
*        store ith column of the Jacobian in ith row of Jactrans array
*
         DO 15 J = 1,M
            JACTRANS(I,J) = G$YP(1,J)
   15    CONTINUE
   10 CONTINUE
```

Even though **g$yp(i,j)** as computed in Approach 1 equals **jactrans(i,j)** computed in Approach 2, the second method is significantly less efficient. This inefficiency arises from the fact that the value of **yp** itself is computed once in the first approach, but **m** times in the second approach. Thus, it is usually best to compute as large a slice of the Jacobian as memory restrictions will allow.

3 Case 2: Dense Jacobian, multiple independent and multiple dependent variables

The second example involves a code that models adiabatic flow [16], a commonly used module in chemical engineering. This code models the separation of a pressurized mixture of hydrocarbons into liquid and vapor components in a distillation column, where pressure (and, as a result, temperature) decrease. This example was communicated to us by Larry Biegler.

In its original version, the top-level subroutine

```
      subroutine aifl(kf)
      integer kf
```

has only one argument. All other information is passed in common blocks. For demonstration purposes, we changed the interface slightly to

```
      subroutine aifl(kf,feed,pressure,liquid,vapor)
      integer kf
      real feed(*), pressure(*), liquid(*), vapor(*)
```

copying the values passed in those arguments into the proper common blocks in `aifl`. As our first example, assume that we are interested in $\frac{\partial\, liquid}{\partial\, feed}$ and $\frac{\partial\, vapor}{\partial\, feed}$ †. In this case, **ADIFOR** generates

```
 subroutine g$aifl$26(g$p, kf, feed, g$feed, ldg$feed, pressure,
$                     liquid, g$liquid, ldg$liquid,
$                     vapor, g$vapor, ldg$vapor)
 integer g$p, kf, ldg$feed, ldg$liquid, ldg$vapor
 real feed(*), g$feed(ldg$feed,*), pressure(*),
$     liquid(*), g$liquid(ldg$liquid,*),
$     vapor(*), g$vapor(ldg$vapor,*)
```

In our example, the feed was a mixture of the hydrocarbons N-butane, N-pentane, 1-butene, cis-2-butene, trans-2-butene, and propylene, so the length of `feed`, `liquid`, and `vapor` was six, with `feed(1)` corresponding to the N-butane feed, and so on. So if we set `g$p=6` and initialize `g$feed` to a 6×6 identity matrix, then on exit `g$liquid(i,j)` contains

$$\frac{\partial\,(\text{component } j \text{ in liquid})}{\partial\,(\text{component } i \text{ in feed})},$$

which predicts by what amount the liquid portion of substance j will change, if the feed of component i changes.

Suppose that we also wish to treat the pressure at the various inlets as an independent variable, but (because of the conservation law) we decide not to declare "vapor" as a dependent variable, **ADIFOR** generates

```
 subroutine g$aifl$14(g$p, kf, feed, g$feed, ldg$feed,
$                     pressure, g$pressure, ldg$pressure,
$                     liquid, g$liquid, ldg$liquid, vapor)
```

The initialization is a little more complicated this time. Assuming that we have 3 feeds (so `pressure` has three elements), the total number of independent variables is $6 + 3 = 9$. `g$liquid` measures the sensitivity of the 6 substances with respect to changes in the 9 independent variables. Thus,

$$J_{liquid} = \left(\frac{\partial\, liquid}{\partial\, pressure}, \frac{\partial\, liquid}{\partial\, feed} \right)$$

is a 6×9 matrix. **ADIFOR** computes

$$\texttt{g\$liquid} = \left(J_{liquid} \begin{pmatrix} \texttt{g\$feed}^T \\ \texttt{g\$pressure}^T \end{pmatrix} \right)^T .$$

If we wish to compute the whole Jacobian J, then

$$\begin{pmatrix} \texttt{g\$feed}^T \\ \texttt{g\$pressure}^T \end{pmatrix}$$

† Actually, it is sufficient to compute one or the other, since, because of conservation laws, $\frac{\partial\, liquid}{\partial\, feed} + \frac{\partial\, vapor}{\partial\, feed}$ equals the identity matrix.

must be initialized to a 9×9 identity matrix. Thus, $\texttt{g\$feed}^T$ must contain the first six rows of a 9×9 identity matrix (since there are six variables in the feed), and $\texttt{g\$pressure}^T$ must contain the last three rows of a 9×9 identity matrix. This configuration is achieved by initializing

$$\texttt{g\$feed} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \texttt{g\$pressure} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

4 Case 3: Sparse Jacobian, one independent, one dependent variable

From the previous discussion, **ADIFOR** may seem to be well suited for computing dense Jacobian matrices, but rather expensive for sparse Jacobians. A primary reason is that the forward mode of automatic differentiation upon which **ADIFOR** is mainly based (see [2]) requires roughly `g$p` operations for every assignment statement in the original function. Thus, if we compute a Jacobian J with n columns by setting $\texttt{g\$p} = n$, its computation will require roughly n times as many operations as the original function evaluation, independent of whether J is dense or sparse. However, it is well known [5,8] that the number of function evaluations that are required to compute an approximation to the Jacobian by finite differences can be much less than n if J is sparse. Fortunately, the same idea can be applied to greatly reduce the running time of **ADIFOR**-generated derivative code as well.

The idea is best understood with an example. Assume that we have a function

$$F = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{pmatrix} : x \in \mathbf{R}^4 \mapsto y \in \mathbf{R}^5$$

whose Jacobian J has the following structure (symbols denote nonzeros, and zeros are not shown):

$$J = \begin{pmatrix} \bigcirc & & & \\ \bigcirc & & & \diamond \\ & \triangle & & \diamond \\ & \triangle & \square & \\ & \triangle & \square & \end{pmatrix}.$$

That is, the function f_1 depends only on x_1, f_2 depends only on x_1 and x_4, and so on. The key idea in sparse finite difference approximations is to identify so-called *structurally orthogonal* columns j_i

of J– that is, columns whose inner product is zero, independent of the value of x. In our example, columns 1 and 2 are structurally orthogonal, and so are columns 3 and 4. This means that the set of functions that depend nontrivially on x_1, and the set of functions that depend nontrivially on x_2 are disjoint.

To exploit this structure, recall that **ADIFOR** (ignoring transposes) computes $J \cdot S$, where S is a matrix with **g$p** columns. For our example, setting $S = I_{4\times 4}$ will give us J at roughly four times the cost of evaluating F, but if we exploit the structural orthogonality and set

$$S = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix},$$

the running time for the **ADIFOR** code is roughly halved. *Note that the ADIFOR-generated code remains unchanged.*

As a more realistic example, we consider the swirling flow problem, part of the MINPACK-2 test problem collection [1]. Here we solve a nonlinear system of equations $F(x) = 0$ for $F : \mathbf{R}^n \rightarrow \mathbf{R}^n$. The swirling flow code has the form

```
subroutine dswirl3(nxmax,x,fvec,fjac,ldfjac,job,eps,nint)
integer nxmax, ldfjac, job, nint
double precision x(*), fvec(*), fjac(ldfjac,*), eps
```

Like all codes in the MINPACK-2 test collection, it is set up to compute the function values (in `fvec`) and, if desired, the analytic first-order derivatives (in `fjac`) as well. The vectors `x` and `fvec` are of size `nxmax` = `14*nint`. For example, for `nint` = 4, the Jacobian of F is of size `nxmax` = 56 and has the structure shown in Figure 1.

The derivative subroutine produced by **ADIFOR** is

```
      subroutine g$dswrl3$3 (g$p, nxmax, x, g$x, ldg$x,
     +                fvec, g$fvec, ldg$x,
     +                fjac, ldfjac, 1, eps, nint)
```

If we initialize `g$x` to a 56×56 identity matrix, and let `g$p`=56, and if `ldg$x` is at least 56, then on exit from `g$dswrl3$3`, `g$fvec` will contain the transpose of $\frac{\partial F}{\partial x}$, stored as a dense matrix. As it turns out, less than 7 % of the total operations performed with gradient objects in the **ADIFOR** code involve nonzeros. On the other hand, by using a graph-coloring algorithm designed to identify structurally orthogonal columns (we used the one described in [4]), we can determine that this Jacobian can be grouped into 14 sets of structurally orthogonal columns, independent of the size of the problem. In our example, columns 1, 16, 31, and 51 were in the first group; columns 2, 17, 37, and 43 were in the second group; and so on. We can take advantage of this fact by initializing the first column of $\texttt{g\$x}^T$ such that it has 1.0 in rows 1, 16, 31, and 51; by initializing the second column of $\texttt{g\$x}^T$ such that it has 1.0 in rows 2, 17, 37, and 43; and so on. The structure of $\texttt{g\$x}^T$ thus initialized is shown in Figure 2 together with the resulting compressed Jacobian $\texttt{g\$fvec}^T$. Note that instead of `g$p`= 56 we now can get by with `g$p`= 14, a sizeable reduction in cost.

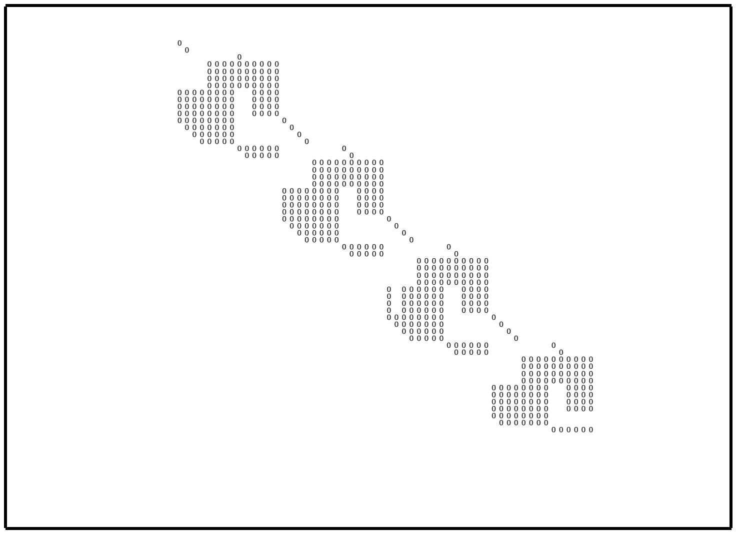

Figure 1: Structure of the swirling flow Jacobian, n = 56

Assuming that `color(i)` is the "color" of column i of the Jacobian and that `nocolors` is the number of colors (in our example we had 14 colors), the following code fragment properly initializes `g$x`, calls `g$dswrl3$3` to compute the compressed Jacobian, and then extracts the Jacobian.

```
      n = 14*nint
      do i = 1, n
        do j = 1, nocolors
          g$x(j,i) = 0
        enddo
        g$x(color(i),i) = 1
      enddo

      call g$dswrl3$3 (nocolors, nxmax, x, g$x, pmax,
     +                 fvec, g$fvec, pmax,
     +                 fjac, ldfjac, 1, eps, nint)
c     job = 1 indicates that only the function value is to be computed in
c             dswrl3.


c     nonzero(j,i) is TRUE if the (j,i) entry in the Jacobian is nonzero,
c     and FALSE otherwise.

      do i = 1, n
        do j = 1, n
          if nonzero(j,i) then
            jac(j,i) = g$fvec(color(i),j)
```

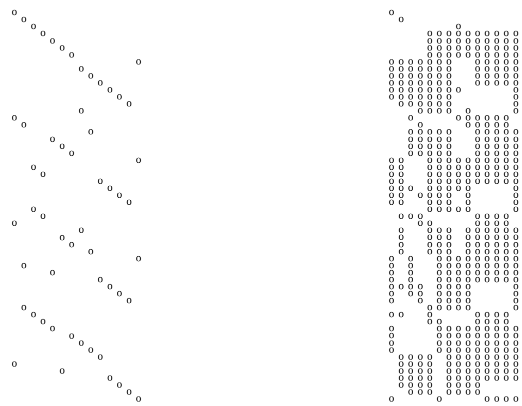

Figure 2: Left: Structure of $(\texttt{g\$x})^T$ Right: Structure of $(\texttt{g\$fvec})^T$

```
      else
        jac(j,i) = 0.0
      endif
    enddo
  enddo
```

Computing the Jacobian with **ADIFOR** in this way performed at least as well as the analytic MINPACK-2 Jacobian on both a SPARC-compatible Solbourne 5E/900 and a one-processor Cray Y/MP.

5 Case 4: Sparse Jacobian, two independent variables, one dependent variable

The coating thickness problem, conveyed to us by Janet Rogers of the National Institute of Standards and Technology, presents many alternatives for using **ADIFOR**-generated subroutines. The code for this problem is (in abbreviated form) shown below:

```
      SUBROUTINE fun(n,m,np,nq,
     +               beta,xplusd,ldxpd,
     +               f,ldf)

c  Subroutine Arguments
c      ==¿ n        number of observations
c      ==¿ m        number of columns in independent variable
c      ==¿ np       number of parameters
c      ==¿ nq       number of responses per observation
```

```
c      ==¿ beta     current values of parameters
c      ==¿ xplusd   current value of independent variable, i.e., x + delta
c      ==¿ ldxpd    leading dimension of xplusd
c     ¡== f         predicted function values
c      ==¿ ldf      leading dimension of f

c  Variable Declarations
      INTEGER          i,j,k,ldf,ldxpd,m,n,np,nq,numpars
      INTEGER          ia, ib
      DOUBLE PRECISION beta(np),f(ldf,nq),xplusd(ldxpd,m)

      double precision par(20),fn(2)

      do 10 k=1,np
         par(k) = beta(k)
   10 continue

      do 100 i=1,n
         do 20 j=1,m
            par(np+j) = xplusd(i,j)
   20    continue

c  compute function values (fn) given parameters (par)
         call fnc(par,fn)

         f(i,1) = fn(1)
         f(i,2) = fn(2)

  100 continue
      return
      end


      subroutine fnc(x,fn)
      integer m,np,nq
      parameter (np=8,m=2,nq=2)
      integer i
      double precision x(np+m),fn(nq)
      double precision beta(np),xplusd(m)

      do 10 i=1,np
         beta(i) = x(i)
   10 continue
      do 20 i=1,m
         xplusd(i) = x(np+i)
   20 continue
```

```
c  compute first of multi-response observations

   fn(1) =   beta(1)
    +              + beta(2)*xplusd(1)
    +              + beta(3)*xplusd(2)
    +              + beta(4)*xplusd(1)*xplusd(2)

c  compute second of multi-response observations

   fn(2) =   beta(5)
    +              + beta(6)*xplusd(1)
    +              + beta(7)*xplusd(2)
    +              + beta(8)*xplusd(1)*xplusd(2)

      return
      end
```

The special format of this code is due to its embedding in the ODRPACK software for orthogonal distance regression. We are interested in the derivatives of `f` with respect to the variables `beta` and `xplusd`. We shall explore various ways to do this in some detail.

5.1 Approach 1 – Generate derivatives only for `fnc`

The easiest approach is to generate the derivative code only for `fnc`, since it is clear from the code that `f(i,1:2)` depends only on `beta(1:np)` and `xplusd(i,1:m)`. **ADIFOR** then produces

```
subroutine g$fnc$3(x,g$x,ldg$x,fn,g$fn,ldg$fn)
integer m, np, nq
parameter( np = 8, m = 2, nq = 2)
double precision x(np+m), fn(nq), g$x(ldg$x,np+m), g$fn(ldg$fn,nq)
```

If inside `fun` we replace the call to `fnc` with a call to `g$fnc$3`, always initializing `g$x` to a 10×10 identity matrix before the call, then

$$\texttt{g\$fn(k,j)} = \frac{\partial\, \texttt{f(i,j)}}{\partial\, \texttt{beta(k)}}, k = 1, \ldots 8, j = 1, 2.$$

and

$$\texttt{g\$fn(k,j)} = \frac{\partial\, \texttt{f(i,j)}}{\partial\, \texttt{xplusd(i,k}-\texttt{np)}}, k = 9, 10\,.$$

Closer inspection reveals that the 10×2 array `g$fn` always has the following structure (numbers are used to identify nonzero elements):

$$\begin{pmatrix} 1 & 0 \\ 2 & 0 \\ 3 & 0 \\ 3 & 0 \\ 0 & 5 \\ 0 & 6 \\ 0 & 7 \\ 0 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}.$$

In other words, `fn(i,1)` depends only on `beta(1:4)`, and `fn(i,2)` depends only on `beta(5:8)`. Hence, we can compute a compressed version of `g$fn` at reduced cost by merging rows 1 and 5, 2 and 6, 3 and 7, and 5 and 8 of `g$fn`. Keeping in mind that `g$fn` is the *transpose* of the Jacobian, this is an especially simple case of the compression strategy outlined in the previous section. This is achieved by initializing

$$\mathtt{g\$x} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

which results in

$$\mathtt{g\$fn} = \begin{pmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}.$$

All the nonzero values of the Jacobian are now computed at roughly 60% of the cost of the previous approach.

On a SPARC-compatible Solbourne 5E/900 with a clock resolution of 0.01 seconds, executing `fun` took 0.01 seconds, computing derivative values using `g$fnc` without compression took 0.06 seconds, and exploiting the structure of `g$fn` through the initialization of `g$x` shown above reduced that time to 0.03 seconds.

5.2 Approach 2 – Generate derivatives for `fun`

An alternative method of applying **ADIFOR** is to process subroutine `fun`. **ADIFOR** detects the interprocedural data dependence between `fun` and `fnc` and therefore generates `g$fun$176` as well as `g$fnc$3`, with `g$fnc$3` called properly within `g$fun$176`. We obtain

```
      subroutine g$fun$176(g$p,n,m,np,nq,beta,g$beta,ldg$beta,
```

```
     $                 xplusd,g$xplusd,ldg$xplusd,ldxpd,f,g$f,ldg$f,ldf)
      integer g$p, n, m, np, nq, ldg$beta,ldg$xplusd,ldxpd,ldg$f,ldf
      double precision beta(np), g$beta(ldg$beta,np),
     $                 xplusd(ldxpd,m), g$xplusd(ldg$xplusd,ldxpd,m),
     $                 f(ldf,nq), g$f(ldg$f,ldf,nq)
```

Now we have three-dimensional derivative objects, which somewhat complicates the initialization of **g$xplusd** and the interpretation of the results in **g$f**. However, this is not too difficult if we keep in mind that we wish to initialize

$$\begin{pmatrix} \texttt{g\$beta}^T \\ \texttt{g\$xplusd}^T \end{pmatrix}$$

to an identity matrix. The number of elements in **xplusd** is **n*m**, and the number of elements in **beta** is **np**. For the coating thickness problem, **n=63, m=2,** and **np=8**. Hence, the identity matrix should be 134×134. This is also the value we shall use for **g$p**. Initialization of **g$beta** follows the scheme outlined in § 3; that is, the first 8 rows should be an 8×8 identity matrix, and the remaining 126 rows should be initialized to zero. How to initialize **g$xplusd** is less readily apparent, for it is not immediately obvious how to form a 126×126 identity matrix from a three-dimensional structure. However, if one looks at the way Fortran stores two-dimensional structures in memory, a simple scheme for storing the Jacobian develops. In Fortran, element (j, i) in an $n \times m$ array is stored as if it were element $n * (i - 1) + j$ of a one-dimensional array. Thus, we can apply this technique to map the 126 columns of the Jacobian that should be initialized to the identity onto **g$xplusd**. Specifically, element $(np + k, j, i)$ is initialized to 1 if and only if $k = 63 * (i - 1) + j$. The following code segment accomplishes this initialization.

```
c n=63, m=2, np=8

      g$p$ = np + m*n
      do 44 i = 1, np
        do 144 j = 1, g$p$
          g$beta(j,i) = 0.0
 144    continue
        g$beta(i,i) = 1.0
  44  continue
      do 45 i = 1, m
        do 145 j = 1, n
          do 245 k = 1, g$p$
            g$xplusd(k,j,i) = 0.0
 245      continue
          g$xplusd(np+((i-1)*n)+j,j,i) = 1.0
 145    continue
  45  continue
```

When initialized in this manner, **ADIFOR** computes

$$\mathtt{g\$f} = \left(J_f = \left(\frac{\partial\, f}{\partial\, beta}, \frac{\partial\, f}{\partial\, xplusd} \right) \right)^T .$$

However, the performance of this approach is poor, since we totally ignore the sparsity structure of the Jacobian. As a result, the computation of J_f takes 0.77 seconds on a Solbourne 5E/900. A better way to find the Jacobian of `f` using `g$fun$176` is to take note of the structures used by `fun`. From this, it becomes obvious that $\frac{\partial f[i,j]}{\partial xplusd[k,l]}$ is nonzero only when $i = k$. As a consequence, we may change the

```
g$p = np + m*n
. . .
g$xplusd(np+((i-1)*n)+j,j,i) = 1.0
```

to the much simpler

```
g$p = np + m
. . .
g$xplusd(np+i,j,i) = 1.0
```

with the understanding that `g$f(np+i,j,k)` ($i = 1..m$) represents $\frac{\partial f[j,k]}{\partial xplusd[j,i]}$. This is equivalent to initializing

$$\mathtt{g\$beta} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \mathtt{g\$xplusd[n]} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} .$$

This implementation is much more efficient than that described in the preceding paragraph and more closely mimics the behavior of the original subroutine `fun`. As a consequence, the time required to execute `g$fun$176` using this initialization is 0.07 seconds.

As discussed in § 5.1, only half of the derivatives of `f` with respect to `beta` are nonzero. Specifically, $\frac{\partial f[i,1]}{\partial beta[j]}$ is nonzero for $j = 1..4$ and zero for $j = 5..8$, while $\frac{\partial f[i,2]}{\partial beta[j]}$ is zero for $j = 1..4$ and nonzero for $j = 5..8$. This information can be used to further compress the Jacobian. The initialization

$$\texttt{g\$beta} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \texttt{g\$xplusd}[\texttt{n}] = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

compresses the Jacobian into only 6 columns. Columns 1 through 4 represent the nonzero derivatives of `f` with respect to `beta`, while columns 5 and 6 correspond to the derivatives of `f[i,j]` with respect to `xplusd[i,1..2]`, as above. This initialization may be accomplished with the following code fragment.

```
c n=63, m=2, np=8
      halfnp = 4
      g$p$ = 4 + m
      do 44 i = 1, halfnp
        do 144 j = 1, g$p$
          g$beta(j,i) = 0.0
          g$beta(j,i+halfnp) = 0.0
  144   continue
        g$beta(i,i) = 1.0
        g$beta(i,i+halfnp) = 1.0
   44 continue
      do 45 i = 1, m
        do 145 j = 1, n
          do 245 k = 1, g$p$
            g$xplusd(k,j,i) = 0.0
  245     continue
          g$xplusd(halfnp+i,j,i) = 1.0
  145   continue
   45 continue
```

This approach is efficient, capable of computing all derivatives in 0.03 seconds. However, it has the disadvantage that the initialization routine might have to be changed if `fnc` or `np` is altered.

6 Computing Gradients of Partially Separable Functions

A particular class of functions that arises often in optimization contexts is that of the so-called *partially separable* functions [6,11,12,14,15]. That is, we have a function $f : \mathbf{R}^n \to \mathbf{R}$ which can be expressed as

$$f(x) = \sum_{i=1}^{nf} f_i(x).$$

Usually each f_i depends on only a few (say, n_i) of the x's, and one can take advantage of this fact in computing the (sparse) Hessian of f.

As was pointed out to us by Andreas Griewank, this structure can also be used advantageously in computing the (usually dense) gradient ∇f of f.

Assume that the code for computation of f looks as follows:

```
subroutine f(n,x,fval)
integer n
real x(n), fval, temp

fval = 0

call f1(n,x,temp)
fval = fval + temp

......

call fnb(n,x,temp)
fval = fval + temp

return
end
```

If we submit `f` to ADIFOR, it generates

```
subroutine g$fn(n,x,g$x,ldg$x,fval,g$fval,ldg$fval).
```

To compute ∇f, the first (and only) row of the Jacobian of f, we set `g$p=` n and initialize `g$x` to a $n \times n$ identity matrix. Hence, the cost of computing ∇f is of the order of n times the function evaluation.

As an alternative, we realize that with $f : \mathbf{R}^n \to \mathbf{R}^{nb}$ defined as

$$g = \begin{pmatrix} f_1 \\ \vdots \\ f_{nb} \end{pmatrix},$$

we have the identities

$$f(x) = e^T g(x), \text{ and hence } \nabla f(x) = e^T J_g,$$

where e is the vector of all ones, and J_g is the Jacobian of g. We can get the gradient of f by computing J_g and adding up its rows. The corresponding code fragment for computing f is

```
subroutine f(n,x,fval)
integer n
real x(n)

integer nf, i
parameter (nf = <whatever>)
real gval(nf)
```

```
call g(n,x,gval)

fval = 0
do i = 1,nb
   fval = fval + gval(i)
enddo

return
end
```

It may not appear that we have gained anything, since J_g is $nf \times n$: if we initialize **g$x** in

```
subroutine g$g(g$p,n,x,g$x,ldg$x,gval,g$gval,ldg$gval)
```

to an $n \times n$ identity matrix, then the computation of J_g still takes about n times as long as the computation of g (or f).

The key observation is that the Jacobian J_g is likely to be sparse, since

$$J_g = \begin{pmatrix} (\nabla f_1)^T \\ \vdots \\ (\nabla f_{nb})^T \end{pmatrix},$$

and each of the f_i's depends only on n_i of the x's. By using the graph coloring techniques described in Section 4, we can compute J_g at a cost that is proportional to the number of columns in the compressed J_g, and then add up its (sparse) rows. As a result, we can compute ∇f at a cost that is potentially much less than n times the evaluation of f.

7 Conclusions

This report demonstrated how to properly use **ADIFOR**-generated derivative codes. One of the strengths of **ADIFOR** is that it does not assume a particular calling sequence of the function to be differentiated. We gave examples that showed how to properly use **ADIFOR**-generated codes for various styles of codes. We also showed how to exploit a known sparsity structure of the derivative matrix in the initialization of **ADIFOR** code. By properly initializing the derivative objects corresponding to independent variables, we can merge structurally orthogonal columns and hence compute derivatives at greatly reduced cost. We also mentioned partially separable functions, where this technique can also be applied advantageously to the computation of dense gradient objects.

Acknowledgments

We would like to thank Alan Carle, George Corliss and Andreas Griewank for the many suggestions that found their way into this report. We would also like to thank Larry Biegler and Janet Rogers for supplying us with test problems.

References

[1] Brett Averick, Richard G. Carter, and Jorge J. Moré. The MINPACK-2 test problem collection (preliminary version). Technical Report ANL/MCS–TM–150, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[2] Christian Bischof, Alan Carle, George Corliss, and Andreas Griewank. ADIFOR–generating derivative codes from Fortran programs. ADIFOR Working Note #1, MCS–P263–0991, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[3] D. Callahan, K. Cooper, R.T. Hood, K. Kennedy, and L.M. Torczon. ParaScope: a parallel programming environment. *International Journal of Supercomputer Applications*, 2(4), December 1988.

[4] Thomas F. Coleman. *Large Sparse Numerical Optimization*, volume 165 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1984.

[5] Thomas F. Coleman, Burton S. Garbow, and Jorge J. Moré. Software for estimating sparse Jacobian matrices. *ACM Transactions on Mathematical Software*, 10(3):329–345, 1984.

[6] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. An introduction to the structure of large scale nonlinear optimization problems and the LANCELOT project. Report 89-19, Namur University, Namur, Belgium, 1989.

[7] Wayne H. Enright and John D. Pryce. Two FORTRAN packages for assessing initial value methods. *ACM Trans. Math. Software*, 13(1):1–22, 1987.

[8] D. Goldfarb and P.L. Toint. Optimal estimation of Jacobian and Hessian matrices that arise in finite difference calculations. *Mathematics of Computation*, 43:69–88, 1984.

[9] Andreas Griewank. On automatic differentiation. In: *Mathematical Programming: Recent Developments and Applications*, Kluwer Academic Publishers, Amsterdam, 1989, pages 83–108.

[10] Andreas Griewank. The chain rule revisited in scientific computing. Preprint MCS–P227–0491, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[11] Andreas Griewank and Philippe L. Toint. On the unconstrained optimization of partially separable objective functions. In: *Nonlinear Optimization 1981*, M.J.D. Powell, editor, Academic Press, London, 1981, pages 301–312.

[12] Andreas Griewank and Philippe L. Toint. Partitioned variable metric updates for large structured optimization problems. *Numerische Mathematik*, 39:119–137, 1982.

[13] David Juedes. A taxonomy of automatic differentiation tools. In: *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Andreas Griewank and George Corliss, eds., SIAM, Philadelphia, 1991. To appear.

[14] M. Lescrenier. Partially separable optimization and parallel computing. *Ann. Oper. Res.*, 14:213–224, 1988.

[15] J. J. Moré. On the performance of algorithms for large-scale bound constrained problems. In: *Large-Scale Numerical Optimization*, F. Coleman and Y. Li, editors, SIAM, Philadelphia, 1991.

[16] J. M. Smith and H. C. Van Ness. *Introduction to Chemical Engineering.* McGraw-Hill, New York, 1975.