Extending Compile-Time Reverse Mode and Exploiting Partial Separability in ADIFOR

Christian H. Bischof
Moe EL-Khadiri

CRPC-TR92428
October, 1992
ADIFOR Working Note #7:
Extending Compile-Time Reverse Mode and
Exploiting Partial Separability in ADIFOR

by
Christian H. Bischof and Moe EL-Khadiri

Argonne Technical Memorandum MCS-TM-163, 1992

Abstract. The numerical methods employed in the solution of many scientific computing problems require the computation of the gradient of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$. ADIFOR is a source translator that, given a collection of subroutines to compute $f$, generates Fortran 77 code for computing the derivative of this function. Using the so-called torsion problem from the MINPACK-2 test collection as an example, this paper explores two issues in automatic differentiation: the efficient computation of derivatives for partial separable functions and the use of the compile-time reverse mode for the generation of derivatives. We show that orders of magnitudes of improvement are possible when exploiting partial separability and maximizing use of the reverse mode.

1 Introduction

Differentiation is one of the most fundamental mathematical concepts. In system analysis and control, the investigation into the effect of a disturbance or a change in design parameters on the performance of the overall system is essential. Mathematically, the change can be modeled by the derivative of the system output with respect to a design parameter. Another application is the numerical solution of initial value problems in stiff ordinary differential equations (see, for example [7, 18]). Methods such as implicit Runge-Kutta and backward differentiation formula (BDF) methods require a Jacobian which is either supplied by the user or approximated by finite differences. In the context of optimization, one needs the derivatives of the objective function. For example, given a function

$$f : \mathbb{R}^n \rightarrow \mathbb{R},$$

one can find a minimizer $x_*$ of $f$ using variable metric methods that involve the iteration

$$\text{for } i = 1, 2, \ldots \text{ do}$$
$$\quad \text{Solve } B_i s_i = -\nabla f(x_i)$$
$$\quad x_{i+1} = x_i + \alpha_i s_i$$
$$\text{end for}$$

where $B_i$ is a symmetric positive definite matrix approximating the Hessian of $f$ at $x_i$.

These methods are examples of typical methods applied in numerical computations, where the computation of the derivative is a crucial step in the numerical solution process ([6, 9, 12, 8, 21]). One particular optimization problem is the elastic plastic torsion problem, which arises from the determination of the stress field on an infinitely long cylindrical bar. The infinite-dimensional version of this problem is of the form

$$\min \{ g(v) : v \in K \},$$

1
where \( q : K \rightarrow R \) is the quadratic

\[
q(v) = \frac{1}{2} \int_D \| \nabla v(x) \|^2 dx - c \int_D v(x) dx
\]

for some constant \( c \), and \( D \) is a bounded domain with smooth boundary. The convex set \( K \) is defined by

\[
K = \{ v \in H^2_0(D) : | v(x) | \leq \text{dist}(x, \partial D), x \in D \},
\]

where \( \text{dist}(x, \partial D) \) is the distance function to the boundary of \( D \), and \( H^2_0(D) \) is the Hilbert space of all functions with compact support in \( D \) such that \( v \) and \( \| \nabla v \|^2 \) belong to \( L^2(D) \). This formulation and the physical interpretation of the torsion problem are discussed in the test problem collection of MINPACK-2 \[1\]. A finite element approximation of the torsion problem leads to

\[
q(v) = \frac{1}{2} \sum q_{i,j}^L(v) + \frac{1}{2} \sum q_{i,j}^U(v) - h_x h_y \sum w_i(z_{i,j}) v_{i,j},
\]

where

\[
q_{i,j}^L(v) = \mu_{i,j} \left\{ \left( \frac{v_{i+1,j} - v_{i,j}}{h_x} \right)^2 + \left( \frac{v_{i,j+1} - v_{i,j}}{h_y} \right)^2 \right\},
\]

\[
q_{i,j}^U(v) = \lambda_{i,j} \left\{ \left( \frac{v_{i-1,j} - v_{i,j}}{h_x} \right)^2 + \left( \frac{v_{i,j+1} - v_{i,j}}{h_y} \right)^2 \right\},
\]

and \( \mu_{i,j}, \lambda_{i,j}, h_x, \) and \( h_y \) are constants.

Note that \( q_{i,j}^L(v) \) and \( q_{i,j}^U(v) \) are quadratics which depend only on \( v_{i+1,j}, v_{i,j+1}, v_{i,j} \), and on \( v_{i-1,j}, v_{i,j-1}, v_{i,j} \), respectively. The third contribution to \( q(v) \), which is the linear part, depends only on \( v_{i,j} \). So if we define

\[
S_1 = \sum q_{i,j}^L(v),
\]

\[
S_2 = \sum q_{i,j}^U(v),
\]

\[
S_3 = \sum w_i(z_{i,j}) v_{i,j},
\]

then

\[
f(x) = \frac{1}{2} S_1 + \frac{1}{2} S_2 - h_x h_y S_3.
\]

In the MINPACK-2 code for the torsion problem shown in Appendix A, LOOP1, LOOP2, and LOOP3 correspond to the computation of \( S_1, S_2, \) and \( S_3 \), respectively.

The torsion problem is a particular instance of a particular class of functions that arises often in optimization contexts, the so-called partially separable functions \[11,17,19\]. These are functions \( f : R^n \rightarrow R \) which can be expressed as

\[
f(x) = \sum_{i=1}^{n\lambda} \alpha_i f_i(x).
\]
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 $f$.

ADIFOR (Automatic Differentiation of Fortran) is a source translator that augments Fortran codes with statements for the computation of derivatives [3, 2]. ADIFOR employs a mixed forward/reverse mode paradigm. The forward mode propagates derivatives of intermediate variables with respect to the input variables; the reverse mode propagates derivatives of the final values with respect to intermediate variables [14]. The forward mode follows the flow of execution of the original program, whereas the reverse mode of automatic differentiation requires the ability to access values generated in the execution of a program in reverse order, which is usually achieved by logging all values on a so-called tape, and then interpreting the tape in reverse order [14, 16, 15]. ADIFOR pioneered the use of the compile-time reverse mode where, instead of logging values at run time, we apply the reverse mode at compile time, thereby eliminating the storage requirements and run-time overhead of the tape scheme.

In this paper, we are concerned with the efficient generation of derivative code through the reverse mode of automatic differentiation, and the efficient use of the generated derivative code for computing gradients of partially separable functions. We use the torsion problem as a case study and explore how to improve the current ADIFOR-generated code and decrease the time and storage complexity of computing derivatives.

The paper is structured as follows. In the next section, we recall the key points about the method that is currently used in ADIFOR to generate derivatives. In Section 3, we then illustrate extensions of the compile-time reverse mode from basic blocks all the way to generating an adjoint code for the whole program. In Section 4, we explore the use of partial separability in computing derivatives. We present experimental results on Sparc-2 and IBM RS6000/590 workstations in Section 5.

2 Current ADIFOR Strategy

Automatic differentiation techniques rely on the fact that every function, no matter how complicated, is executed on a computer as a (potentially very long) sequence of elementary operations such as additions, multiplications, and elementary functions such as sin and cos. By applying the chain rule

$$\frac{\partial}{\partial t} f(g(t))|_{t=t_0} = \left( \frac{\partial}{\partial s} f(s)|_{s=g(t_0)} \right) \left( \frac{\partial}{\partial t} g(t)|_{t=t_0} \right)$$

(1)

over and over again to the composition of those elementary operations, one can compute derivative information of $f$ exactly and in a completely mechanical fashion [5]. ADIFOR transforms Fortran 77 programs using this approach.

To illustrate automatic differentiation with current ADIFOR, we differentiate the subroutine `torfcn` for the torsion problem that maps an $n$-vector $\mathbf{x}$ into a scalar $f$. The vector $\mathbf{x}$ contains the independent variables, and the scalar $f$ contains the dependent variable. The full code for `torfcn` can be found in the appendix.

The first loop (LOOP1) is shown in Figure 1. It computes $S_1$, whose value is stored in $f\text{quad}$. Currently, ADIFOR generates the code shown in Figure 2 for computing $\frac{\partial f\text{quad}}{\partial \mathbf{x}}$. In accordance
\begin{verbatim}
  fquad = 0.0
  do 20 j = 0, ny
    do 10 i = 0, nx
      k = nx*(j-i) + i
      vr = 0.0
      vu = 0.0
      if (i .ge. 1 .and. j .ge. 1) v = x(k)
      if (i .lt. nx .and. j .gt. 0) vr = x(k+1)
      if (i .gt. 0 .and. j .lt. ny) vu = x(k+nx)
      fquad=fquad + hxy*(vr-v)**2 + hxy*(vu-v)**2
  10 continue
  20 continue
\end{verbatim}

Figure 1: Code for LOOP1

with the specification of ADIFOR (see [3]), $g\$p$ denotes the actual length of the derivative objects in a call to derivative code. Since Fortran 77 does not allow dynamic memory allocation, derivative objects for local variables are statically allocated with leading dimension $p_{\text{max}}$. $p_{\text{max}}$ is specified by the user when ADIFOR processes the Fortran code for $\text{torfcn}$. A variable and its associated objects are treated in analogous manner; that is, if $x$ is function parameter, so is $g\$x$. Derivative objects corresponding to locally declared variables or variables in common blocks are declared as local variables or variables in common blocks. Given $x$ and $g\$x$, the derivative code computes

$$g\$fquad(1:g\$p) = \left( \left( \frac{df\text{quad}}{dx} \right) g\$x(1:g\$p,1:n)^T \right)^T.$$

In particular, if $g\$p$ equals $n$ and $g\$x$ is the $n \times n$ identity matrix, it computes the gradient of $f\text{quad}$ with respect to $x$.

An active variable is one that is on the computational path from independent to dependent variables (see [4]). Notice that in the ADIFOR-generated code, a loop of length $g\$p$ is associated with every assignment statement involving an active variable. Therefore the cost of floating-point operations can be approximated as $(g\$p \times \text{function evaluation})$. The storage requirement for ADIFOR-generated code is $(g\$p \times \text{number of active variables})$. We note two key points about the current ADIFOR:

- ADIFOR uses the forward mode overall to compute derivatives. That is, ADIFOR code maintains the derivatives of intermediate variables with respect to all input variables. So, for example, $g\$vu = \frac{du}{dx}$.
- ADIFOR uses the reverse mode to preaccumulate “local” derivatives for assignments statements.

The reverse mode is best understood with an example. For example, in the torsion problem, we have the assignment

$$f\text{quad} = hxy * (vr - v) ** 2 + hxy * (vu - v) ** 2,$$
FQUAD = 0.0
DO $IS = 1,$PS
$FQUAD($IS) = 0.0d0
END DO
DO 99998 J = 0, N Y
   DO 99999 I = 0, N X
      K = N X* (J-1) + I
      V = 0.0
      DO $IS = 1, $PS
         G$V($IS) = 0.0d0
      END DO
      VR = 0.0
      DO $IS = 1, $PS
         G$V$R($IS) = 0.0d0
      END DO
      VU = 0.0
      DO $IS = 1, $PS
         G$V$U($IS) = 0.0d0
      END DO
      IF (I .GE. 1 .AND. J .GE. 1) THEN
         V = x(k)
         DO $IS = 1, $PS
            G$V($IS) = G$X($IS,K)
         END DO
         V = x(k)
      END IF
      IF (I .LT. N X .AND. J .GT. 0) THEN
         VR = x(k + 1)
         DO $IS = 1, $PS
            G$V$R($IS) = G$X($IS,K+1)
         END DO
         VR = x(k + 1)
      END IF
      IF (I .GT. 0 .AND. J .LT. N Y) THEN
         VU = x(k + N X)
         DO $IS = 1, $PS
            G$V$U($IS) = G$X($IS,K+N X)
         END DO
         VU = x(k + N X)
      END IF
      IF (I .GT. 0 .AND. J .LT. N Y) THEN
         VU = x(k + N X)
         DO $IS = 1, $PS
            G$V$U($IS) = G$X($IS,K+N X)
         END DO
         VU = x(k + N X)
         f$quad = f$quad + h$x * (v$r - v) ** 2 + h$y * (v$u - v) ** 2
         D$O = v$r - v
         D$4 = v$u - v
         DO $IS = 1, $PS
            G$FQUAD($IS) = G$FQUAD($IS) + H$X*2*D$O*G$V$R($IS) +
                           (- (H$X*2*D$4)* (H$X*2*D$O)) +
                           G$V($IS) + H$X*2*D$4*G$V$U($IS)
         END DO
      END IF
      FQUAD = FQUAD + H$Y*D$O**2 + H$Y*D$4**2
   10   CONTINUE
99999 CONTINUE
20 CONTINUE
99998 CONTINUE

Figure 2: ADIFOR-generated Derivative Code for LOOP1
where $hxy$ and $hxy$ are constants. The chain rule tells us that

$$
\nabla f_{\text{quad}} = \frac{\partial f_{\text{quad}}}{\partial v}\nabla v + \frac{\partial f_{\text{quad}}}{\partial v}\nabla u + \frac{\partial f_{\text{quad}}}{\partial v}\nabla v.
$$

Hence, if we know the “local” derivatives ($\frac{\partial f_{\text{quad}}}{\partial v}$, $\frac{\partial f_{\text{quad}}}{\partial v}$, $\frac{\partial f_{\text{quad}}}{\partial v}$) of $f_{\text{quad}}$ with respect to $v$, $u$, and $u$, we can easily compute $\nabla w$, the derivatives of $w$ with respect to $x$. The “local” derivatives ($\frac{\partial f_{\text{quad}}}{\partial v}$, $\frac{\partial f_{\text{quad}}}{\partial v}$, $\frac{\partial f_{\text{quad}}}{\partial v}$) can be computed efficiently by using the reverse mode of automatic differentiation. Here we maintain the derivative of the final result with respect to an intermediate quantity. These quantities are usually called adjoints. They measure the sensitivity of the final result with respect to some intermediate quantity. In the reverse mode, let $t_{\bar{v}}$ denote the adjoint object corresponding to $t$. The goal is for $t_{\bar{v}}$ to contain the derivative $\frac{\partial f_{\text{quad}}}{\partial v}$. We know that $w_{\bar{v}} = \frac{\partial f_{\text{quad}}}{\partial v} = 1.0$. We can compute $y_{\bar{v}}$ and $z_{\bar{v}}$ by applying the following simple rule to the statements executed in computing $f_{\text{quad}}$, but in reverse order:

```plaintext
if s = f(t), then t_{\bar{v}} += s_{\bar{v}} * (df/dt)
if s = f(t,u), then t_{\bar{v}} += s_{\bar{v}} * (df/dt)
    u_{\bar{v}} += s_{\bar{v}} * (df/du)
```

Using this simple recipe (see [14,20]), we generate the code shown in Figure 3 for computing $v_{\bar{v}}$, $v_{\bar{v}}$, and $v_{\bar{v}}$. One can easily convince oneself that

$$
\begin{align*}
v_{\bar{v}} &= 2 * hxy * (v_{\bar{v}} - v) \\
v_{\bar{v}} &= 2 * hxy * (v_{\bar{v}} - v) \\
v_{\bar{v}} &= -2 * hxy * (v_{\bar{v}} - v) - 2 * hxy * (v_{\bar{v}} - v)
\end{align*}
$$

so that we have in fact computed the correct “local” derivatives. The code shown in Figure 2 has been generated by applying this same technique to all other assignment statements involving active variables and by optimizing the resulting code by removing additions with 0 and multiplications with 1. The ADIFOR-generated code for the whole subroutine is shown in Appendix B. ADIFOR is currently not consistent about pulling loop invariant subexpressions out of the loop, but will do so reliably in the future.

3 Extending the Scope of the Compile-Time Reverse Mode

In this section, we explore extensions of the compile-time reverse mode to

- a sequence of assignment statements,
- a nested loop, and
- the whole program.

A closer look at the current ADIFOR-generated code in the preceding sections reveals a substantial time and space overhead associated with the computations of auxiliary gradients such as $g_{\bar{v}}$, $g_{\bar{v}}$, and $g_{\bar{v}}$. In this section, we explore different ways for improving the overall computation of the gradient by extending the scope of the reverse mode.
```c
/* Compute function values */
d$0 = (v*r-v)
d$4 = (v*r-v)
t1 = d$0 * d$0
t2 = t1 * hxy
t3 = d$4 + d$4
t5 = t3 * hxy
w = t2 + t5

/* Initialize adjoint quantities */
  wbar = 1.0; t3bar = 0.0; t2bar = 0.0;
  t1bar = 0.0; d$0bar = 0.0; d$4bar = 0.0;

/* Adjoints for w = t2 + t5 */
t2bar = t2bar + wbar * 1
  t5bar = t5bar + wbar * 1
/* Adjoints for t5 = t3 * hxy */
t3bar = t3bar + t5bar * hxy
/* Adjoints for t3 = d$4 * d$4 */
d$4 = d$4 + t3bar * d$4
  d$4 = d$4 + t3bar * d$4
/* Adjoints for t2 = t1 * hxy */
t1bar = t1bar + t2bar * hxy
/* Adjoints for t1 = d$0 * d$0 */
d$0 = d$0 + t1bar * d$0
  d$0 = d$0 + t1bar * d$0
/* Adjoints for d$4 = v*r - v */
vubar = vubar + d$4bar * 1
  vbar = vbar + d$4bar * (-1)
/* Adjoints for d$0 = (v*r-v) */
vbar = vbar + d$0bar * 1
  vbar = vbar + d$0bar * (-1)
```

Figure 3: Unoptimized Reverse Mode Computation
3.1 Case 1: Reverse Mode for Basic Blocks inside the Loop

In the program for the torsion problem, there are three loops: two for the computation of the quadratic part of the function and one for the computation of the linear part. Consider, for example, LOOP1. Each loop iteration can be viewed as a mapping

\[ [x(k), x(k+1), x(k+nx), f_{\text{quad old}}] \rightarrow f_{\text{quad new}}. \]

We use the notation \( f_{\text{quad old}} \) and \( f_{\text{quad new}} \) to distinguish between the original and updated value of the variable \( f_{\text{quad}} \). Hence, if we know

\[
\frac{\partial f_{\text{quad new}}}{\partial x(k)}, \frac{\partial f_{\text{quad new}}}{\partial x(k+1)}, \frac{\partial f_{\text{quad new}}}{\partial x(k+nx)} \quad \text{and} \quad \frac{\partial f_{\text{quad new}}}{\partial f_{\text{quad old}}} \quad (2)
\]

then we can update \( \nabla f_{\text{quad}} \) as follows:

\[
\nabla f_{\text{quad}} = \frac{\partial f_{\text{quad new}}}{\partial x(k)} \nabla f_{\text{quad}} + \frac{\partial f_{\text{quad new}}}{\partial x(k+1)} \nabla x(k+1)
+nabla x(k+nx) + \frac{\partial f_{\text{quad new}}}{\partial f_{\text{quad old}}} \nabla f_{\text{quad}}.
\]

The derivatives in the equation (2) can easily be computed by applying the reverse mode to the loop body. The resulting code is shown in Figure 4. Note that each variable is assigned only once in each loop iteration. If this had not been the case, we would have had to save the sequence of values of variables that are overwritten by allocating some extra temporary variables. This extension of the scope of the reverse mode saved us 3 derivative vectors \( g^tv, g^vrv, \) and \( g^vu \), and decreased the number of derivative vector operations from 10 to 4.

In general, we can apply this technique in a straightforward fashion to any piece of code that has only one entry and exit point and does not contain subroutine or function calls or loops. We call such a piece of code a basic block. We may have to introduce some temporaries to make sure that each variable is assigned only once (i.e., represents a unique value) in a basic block, but this requires at most as many scalar temporaries as there are lines of code, an insignificant increase of storage. The savings achieved by this technique depend on the particular code at hand, but, in general, will be more pronounced the more statements a basic block contains. The code that results from applying this technique to the whole subroutine is shown in Appendix C.

3.2 Case 2: Reverse Mode for the Whole Loop

In order to expand the scope of the compile-time reverse mode, the special structure of the torsion problem is important. Defining

\[ t_k := -hxy * (v_r - v) * *2 + hxy * (v_u - v) * *2 \]

to be the value computed in loop iteration \( k \) to upgrade \( f_{\text{quad}} \), we can express

\[
f_{\text{quad}} = \sum_{k=1}^{(nx+1)(ny+1)} t_k. \quad (3)
\]
FQUAD = 0.0
DO G$1$ = 1, G$P$
G$FQUAD(G$1$) = 0, O$D$
END DO
DO 99998 J = 0, N
DO 99999 I = 0, N
K = N$1$ + 1
  C  
  C  compute new contribution to sum  
  C  
  V = 0.0
  VR = 0.0
  VU = 0.0
  IF (I.GE.1 .AND. J.GE.1) THEN
    V = X(K)
  END IF
  IF (I.LT.N .AND. J.GT.0) THEN
    VR = X(K+1)
  END IF
  IF (I.GT.0 .AND. J.LT.1) THEN
    VU = X(K+N)
  END IF
  C  
  C  reverse mode computation for computing derivatives of  
  C  x(k), x(k+1), x(k+nx). We know that the deriv. of fquad_new  
  C  with respect to fquad_old is 1.  
  C  
  D8O = VR - V
  D84 = VU - V
  FQUAD = FQUAD + HYX*D8O**2 + HYX*D84**2
  VBAR = -2*HYX*D84 - 2*HYX*D8O
  VUBAR = 2*HYX*D84
  VRBAR = 2*HYX*D8O
  XKBAR = 0.0
  XK1BAR = 0.0
  XKXBAR = 0.0
  IF (I.GE.1 .AND. J.GE.1) XKBAR = VBAR
  IF (I.LT.N .AND. J.GT.0) XK1BAR = VBAR
  IF (I.GT.0 .AND. J.LT.1) XKXBAR = VUBAR
  C  
  C  Chain Rule to update derivatives of fquad w.r.t. x  
  C  
  D0 PP = 1, G$P$
  G$FQUAD(PP) = G$FQUAD(PP) + XKBAR*G$X(PP, K) +
  + XK1BAR*G$X(PP,K+1) +
  + XKXBAR*G$X(PP,K+N)
END DO
99999 CONTINUE
20 CONTINUE
99998 CONTINUE

Figure 4: Reverse Mode for Basic Block in LOOP1
Since \( \mathbf{v}, \mathbf{v}_\mathbf{u}, \) and \( \mathbf{v}_\mathbf{r} \) are defined in terms of \( \mathbf{x}(k+1), \mathbf{x}(k+n\mathbf{x}), \) and \( \mathbf{x}(k), \) \( t_k \) is a function of these values, that is,

\[
t_k = t_k(x(k+1), x(k+n\mathbf{x}), x(k)).
\]

Since no entry of \( \mathbf{x} \) is overwritten in any of the loop iterations, \( t_k \) and \( t_1 \) do not depend on each other for \( k \neq 1, \) and we can compute the sum (3) in any order. In compiler terms, there are no loop-carried dependencies and this loop is a parallel loop.

Remember that the reverse mode implicitly assumes that we are able to trace the values computed during some computation in the reverse order. Hence, a loop that is not parallel would require us to save some intermediate values. However, for a parallel loop, it is sufficient simply to generate the reverse mode code for the loop body. But this is exactly what we already did in the preceding section, where we computed

\[
\frac{\partial t_k}{\partial x(k+1)} \frac{\partial t_k}{\partial x(k)} \frac{\partial t_k}{\partial x(k+n\mathbf{x})}.
\]

Now, since \( t_1 \) and \( t_k \) do not depend on each other for \( l \neq k, \) the associativity of addition allows us to compute

\[
\frac{df_{\text{quad}}}{dx(j)} = \frac{\partial t_{j-1}}{\partial x(j)} + \frac{\partial t_{j-n\mathbf{x}}}{\partial x(j)} + \frac{\partial t_j}{\partial x(j)}
\]

in a piecemeal fashion, as each of the iterations \( j, j-1, \) and \( j-n\mathbf{x} \) contributes to \( \frac{df_{\text{quad}}}{dx(j)} \). The resulting code is shown in Figure 5. The \( \mathbf{fbar} \) vector contains \( \frac{df_{\text{quad}}}{dx} \) and components \( k+1, k, \) and \( k+n\mathbf{x} \) are updated in iteration \( k. \) After the loop, we apply the chain rule to compute

\[
\nabla f_{\text{quad}} = \frac{df_{\text{quad}}}{dx} \cdot \nabla x.
\]

This matrix-vector multiplication is performed using the BLAS routine DGEMV [13].

To summarize, we exploited the fact that

- loop iterations do not depend on each other, and
- the result of each loop enters into the dependent variable (fquad) in an additive fashion.

This allows us to generate reverse mode code for the whole loop by simply generating reverse mode code for the loop body, and the forward mode propagation of the global derivatives could be moved outside of the loop.

Compared with the code in the previous section, we now have a multiplication of an \( \mathbf{g}_{\mathbf{p}} \times (n\mathbf{x}+1) \) \( (n\mathbf{y}+1) \) matrix by a vector outside the loop instead of \( (n\mathbf{x}+1)(n\mathbf{y}+1) \) multiplications of an \( \mathbf{g}_{\mathbf{p}} \times 1 \) matrix by a vector multiplication inside a loop that is executed \( (n\mathbf{x}+1)(n\mathbf{y}+1) \) times, requiring roughly one-fourth the number of operations. Applying this technique to the whole subroutine results in the code shown in Appendix D.

### 3.3 Case 3: The Full Reverse Mode

So far we exploited only the particular structure of the code in LOOP1, LOOP2, and LOOP3. On the other hand, \( f(x) \) is the sum of the contributions computed in LOOP1, LOOP2, LOOP3,
FQUAD = 0.0
DO G$S$ = 1,G$S$
   G$S$FQUAD(G$S$) = 0.0
END DO
DO I = 1,XBARSIZE
   XBAR(I) = 0.0
END DO
DO J = 0,MY
   DO 99999 I = 0,IX
      K = IX* (J-1) + I
      V = 0.0
      VR = 0.0
      VU = 0.0
      IF (I.GE.1 .AND. J.GE.1) V = X(K)
      IF (I.LT.NX .AND. J.GT.JY) VR = X(K+1)
      IF (I.GT.NO .AND. J.LT.JY) VU = X(K+IX)
      C
      FQUAD = FQUAD + HYX * (VR - V) * 2 + HXY * (VU - V) * 2
      D80 = VR - V
      D84 = VU - V
      FQUAD = FQUAD + HYX*D80**2 + HXY*D84**2
      VBAR = -2*HYX*D84 - 2*HYX*D80
      VUBAR = 2*HYX*D84
      VBAR = 2*HYX*D80
      IF (I.GE.1 .AND. J.GE.1) XBAR(K) = XBAR(K) + VBAR
      IF (I.LT.NX .AND. J.GT.JY) XBAR(K+1) = XBAR(K+1) + VBAR
      IF (I.GT.NO .AND. J.LT.JY) XBAR(K+IX) = XBAR(K+IX) + VBAR
    99999 CONTINUE
  99998 CONTINUE
  99992 CONTINUE
  99991 CONTINUE

99991 CONTINUE

CALL DGEMV('N',G$S$,XBARSIZE,1.0,D$S$,LD$S$,XBAR,1,1.0,D$S$,G$S$FQUAD,
* + 1)

Figure 5: Reverse Mode for the Whole Loop
and, in addition to being parallel loops themselves, these loops do not depend on each other. So, instead of computing

\[
x_{\text{bar}}(1:n) = 0; \\
/* \text{Update xbar in LOOP1 */} \\
df_\text{quad} = g\mathbf{x} \times \mathbf{x}_{\text{bar}}
\]

\[
x_{\text{bar}}(1:n) = 0; \\
/* \text{Update xbar in LOOP2 */} \\
df_\text{quad} = df_\text{quad} + g\mathbf{x} \times \mathbf{x}_{\text{bar}}
\]

\[
x_{\text{bar}}(1:n) = 0; \\
/* \text{Update xbar in LOOP3 */} \\
df_\text{quad} = df_\text{quad} + g\mathbf{x} \times \mathbf{x}_{\text{bar}}
\]

we could simply keep on updating xbar in LOOP1, LOOP2, and LOOP3 and perform compute \( df\text{quad} = g\mathbf{x} \times \mathbf{x}_{\text{bar}} \) once at the end. This is possible since none of these loops updates the vector \( \mathbf{x} \), and hence \( g\mathbf{x} \) remains unchanged. But we can go even further: Since in the forward mode, \( g\mathbf{x} \) is initialized to the identity, we can eliminate the final multiplication \( g\text{quad} = g\mathbf{x} \times \mathbf{x}_{\text{bar}} \) and simply assign return \( x_{\text{bar}} \). In this fashion, we have generated adjoint code for the whole subroutine, and the code for computing the gradient does not contain any vector operations.

It is important to note that we were able to do the full implementation of the reverse mode because

- each of the three loops is a parallel loop,
- the three loops do not depend on each other,
- the contribution computed inside each loop enters in the final result in an additive fashion, and
- the results of each of the three loops are added to achieve the final result.

The resulting reverse mode code for the torsion problem is shown in Appendix E. While we did not decrease the storage requirement any further compared with the preceding section, we saved another three loops of size \( g\mathbf{p} \times \mathbf{x} \times \mathbf{y} \), and the run time of this program no longer depends on \( g\mathbf{p} \).

4 Exploring Partial Separability

As was mentioned in the introduction, the torsion problem is a partially separable function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \), in that it can be expressed as

\[
f(x) = \sum_{i=1}^{n} f_i(x).
\]

This structure can also be used advantageously in computing the (usually dense) gradient \( \nabla f \) of \( f \) (see [9]). 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 fi(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$sfn(n,x,g$sx,1dg$sx,fval,g$sfval,1dg$sfval).

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

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

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

we have the identities

$$ f(x) = e^T g(x), $$

and hence $\nabla f(x) = e^T J_g$, where $e$ is the vector of all ones, and $J_g$ is the Jacobian of $g$. However, if most of the component
functions $f_i$ depend only on a few parameters $x_j$, the Jacobian of $g$ is sparse, and this fact can be
exploited advantageously.

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

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

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

$$ J = \begin{pmatrix} \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \\ \bigcirc & \bigcirc & \bigcirc \end{pmatrix}. $$
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 computing sparse Jacobians is to identify so-called structurally orthogonal columns $j_i$ of $J$ (see [10]), that is, columns whose inner product is always zero, independent of the numerical values of their nonzero entries. 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 $x_1$, namely $\{f_1, f_2\}$, and the set of functions that depend nontrivially on $x_2$, namely $\{f_3, f_4, f_5\}$, are disjoint. Because of the graph-coloring approaches that are used to reveal this structure, one usually associates a “color” with every set of structurally orthogonal columns.

To exploit this sparsity structure, we recall that ADIFOR (ignoring transposes) computes $J \cdot S$, where $S$ is a matrix with $g$ 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

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

the running time for the ADIFOR code is roughly halved. The ADIFOR-generated code remains unchanged.

This idea can readily be applied to the torsion problem. By storing the contribution of iteration $k$ to $f\text{quad}$ in the $k$-th element of separate vectors $FQ$, $FQQ$, and $FP$ (for the LOOP1, LOOP2, and LOOP3, respectively), the derivative of $f\text{quad}$ is the sum of the row sums of the Jacobians of $FQ$, $FQQ$, and $FP$.

For example, the code for the loop corresponding to $FQ$ is

```fortran
SUBROUTINE TORFCW1(N,X,X,F,NX,NY,HX,HY,FORCE,FQ)
C .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HXHY
INTEGER N,NX,NY
C ..
C .. Array Arguments ..
DOUBLE PRECISION FQ(*),X(N)
C ..
C .. Local Scalars ..
DOUBLE PRECISION FQUAD,HXY,HYX,V,VR,VU
INTEGER FQK,I,J,K
C
HXY = HX/HY
HYX = HY/HX
FQUAD = 0.0
DO 20 J = 0,NY
  DO 10 I = 0,NX
    K = NX* (J-1) + I
    V = 0.0
    VR = 0.0
    DO 10 K = 0,NX
      FQK = FQ(I+K,J+1)
      IF (FQK .NE. 0.0) THEN
        V = V + FQUAD
      END IF
10    CONTINUE
20 CONTINUE
RETURN
END
```
Figure 6: Sparsity Structures of Component Jacobians

```
VU = 0.0
IF (I. GE. 1 . AND. J. GE. 1) THEN
  V = X(K)
END IF
IF (I. LT. NX . AND. J. GT. 0) THEN
  VR = X(K+1)
END IF
IF (I. GT. 0 . AND. J. LT. NY) THEN
  VU = X(K+NX)
END IF
FQK = (J* (NX+1)) + I + 1
FQ(FQK) = HYX* (VR-V)**2 + HXY* (VU-V)**2
```

The only change (compared with the corresponding code fragment in Appendix A) is that we replaced the accumulation of fquad by an assignment to FQ. Subroutines torfcn2 and torfcn3 to compute FQQ and FP, respectively, are generated in the same fashion. For these codes, ADIFOR then generates the derivative codes shown in Appendix F.

For n = 40, the structures of \( \frac{\partial F}{\partial x} \) and \( \frac{\partial FQQ}{\partial x} \) are shown in Figure 6, and \( \frac{\partial FP}{\partial x} \) is diagonal. The Jacobian for FQ and FQQ can be grouped into three sets of structurally orthogonal columns, independent of the size of the problem. And in the case of the function FP, the Jacobian can be compressed into only one column.

Exploiting this structure, we can now initialize the gradient vector as follows:
* * find sparsity pattern and compute compressed Jacobian pattern *
* * ***************************************************************************

CALL SPARSITY(N,X,F,NX,NY,NX,NY,INDCOLQ,INDROWQ,
+ INDCOLQQ,INDROWQQ)

DO I = 1,NNZQ
INDROWQ(I) = INDCOLQQ(I)
END DO

CALL DSM((NY+1)*(NX+1),N,NNZQ,INDROWQ,INDCOLQ,MGRPQ,MAXGRPQ,
+ MINGRPQ,INFO,IPMTRQ,JPMTQQ,IWA,LIWA)

CALL DSM((NY+1)*(NX+1),N,NNZQQ,INDROW QQ,INDCOL QQ,MGRPQQ,+
+ MAXGRPQQ,MINGRPQQ,INFO,IPMTRQQ,JPMTQQQ,IWA,LIWA)

* * **************************************************************************
* * compute Jacobians for the individual loops *
* * **************************************************************************

c---- calc g$FQ
   DO I = 1,N
      DO J = 1,MGRPQ
         G$X(I,J) = 0
      END DO
      G$X(MGRPQ(I),I) = 1.0
   END DO
   CALL REVOA(MAXGRPQ,N,X,G$X,PQMAX,F,NX,NY,NX,NY,FORC,FQ,
+ G$FQ,MAXCOLOR)

c----- calc g$FQQ
   DO I = 1,N
      DO J = 1,MGRPQQ
         G$X(I,J) = 0
      END DO
      G$X(MGRPQQ(I),I) = 1.0
   END DO
   CALL REVOB(MAXGRPQQ,N,X,G$X,PQMAX,F,NX,NY,NX,NY,FORC,FQQ,
+ G$FQQ,MAXCOLOR)

c---- calc G$FP
c ------ ngrpfp =1 as Jacobian is diagonal
   MAXGRPFP = 1
   DO I = 1,N
       G$X(I) = 1,0
   END DO
   CALL REVOC(MAXGRPFP,N,X,G$X,F$MAX,F,FX,NX,HX,HY,FORCE,FP,
           +     G$FP,MXCOLOR)

*  ***************************************
*  * Assemble final gradient value *
*  ***************************************

   DO I = 1,N
       SPARSEGF(I) = 0.0e0
   END DO

   DO I = 1,NWQ
       ROW = IDROWQQS(I)
       COL = IDCOLOQS(I)
       SPARSEGF(COL) = SPARSEGF(COL) +
           + 0.25*G$FQ(NGRPQ(COL),ROW)
   END DO

   DO I = 1,NWQQ
       ROW = IDROWQQS(I)
       COL = IDCOLOQS(I)
       SPARSEGF(COL) = SPARSEGF(COL) +
           + 0.25*G$FQ(NGRPQQ(COL),ROW)
   END DO

   TEMP = -FORCE*HX*HY
   DO K = 1,N
       SPARSEGF(K) = SPARSEGF(K) + TEMP*G$FP(I,K)
   END DO

After we have initialized some arrays determining the sparsity pattern of the Jacobian, we call the
MINPACK subroutine DSM [9] to determine the proper coloring for the Jacobians of $FQ$ and $FQQ$. Having thus determined $NGRPQ(i)$, the “color” of column $i$ and $MAXGRPQ$, the number of colors for
the Jacobian of $FQ$, we initialize $g$x and calls revoa (a renamed version of the ADIFOR-generated
subroutine for torfcm1) to compute the compressed Jacobian of $FQ$. The same idea is applied to
calculate $g$FQ and $g$FP. Lastly, the derivative values of the subfunctions are all added into a
sparse vector, without ever expanding the compressed component Jacobians, as shown below. For
the Jacobian of $FQ$, the index arrays $IDROWQQS$ and $IDCOLOQS$ indicate the row and column index of
nonzero entries, and the $NGRPQ$ array indicates the group (corresponding to one particular color) of a
certain column. The Jacobian of $FQQ$ is dealt with accordingly. The uncompression of the Jacobian
of $F_Q$ is trivial, since it was diagonal — we just add the $i$-th diagonal entry (properly scaled) to the $i$-th entry of the gradient accumulation vector SPARSEGF. The MINPACK documentation contains details on the particular data structures used to represent the sparse derivative matrices.

We note that we could of course apply the idea of the “basic block reverse mode” to generate improved derivative code for trofcn1, etc. This code is shown in Appendix G. We would expect much less spectacular savings in this case, since the length of the derivative objects was not more than three for our sparse Jacobians (whereas it was $n$ when we did not exploit partial separability).

5 Experimental Results

We tested the performance of our various derivative codes on a Sun Sparcstation iPX with 48 Mbytes of memory and an IBM RS6000/550 with 128 Mbytes of memory. We computed gradients for $n = 10 \times 10, 20 \times 20, \ldots, 100 \times 100$. For the alternatives described in Sections 2, 3.1, and 3.2, we computed gradients in slices of 10 elements (i.e., the gradient was computed by calling the derivative code $\lceil n/10 \rceil$ times). Figure 7 shows the ratio of the run time of a gradient to a function evaluation obtained for these derivative codes. As expected, the run time is linear in $n$, but the slope decreases as we expand the scope of the reverse mode.

In Figure 8 we show the ratio of the run time of a gradient to a function evaluation obtained by the full reverse mode (Section 3.3) and by exploiting the partial separability of the torsion problem. These graphs also show the run time of the hand-coded derivative subroutine supplied in the MINPACK-2 test suite. We see that by exploiting partial separability, we can achieve very good performance for computing the gradient of the torsion problem. This is particularly noteworthy as we do not need to know anything more about the structure of the problem than that it is partially separable. In contrast, intimate knowledge of the code is required to develop the full reverse mode and the hand-coded versions.

Acknowledgments

We thank Alan Carle, George Corliss, and Andreas Griewank for their comments. We are especially grateful to George Corliss for his thoughtful and thorough comments on an earlier draft of this paper.

References


Figure 7: Performance of Current ADIFOR Code and Enhanced Versions
Figure 8: Exploiting Partial Separability, the Full Reverse Mode and Handcoded Derivatives


APPENDICES: Code Listings for the Torsion Problem

A Minpack-2 Code for the Torsion Problem

SUBROUTINE TORFCN(N,X,F,HX,NY,HY,FORCE)
  
  This subroutine computes the function of the torsion problem.
  
  The spacing parameters hx and hy are for a rectangle with
  nx points on the x-axis and ny points on the y-axis

C .. Scalar Arguments..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER N,NX,NY
C ..
C .. Array Arguments..
DOUBLE PRECISION X(N)
C ..
C .. Local Scalars..
DOUBLE PRECISION FQUAD,HXY,HYX,V,VD,YL,VR,VU
INTEGER I,J,K
C ..
HXY = HX/NY
HYX = HY/NX

C Computation of the quadratic part of the function.
C
C LOOP1:
C
FQUAD = 0.0
DO 20 J = 0,NY
  DO 10 I = 0,NX
    K = NX* (J-1) + I
    V = 0.0
    VR = 0.0
    VU = 0.0
    IF (I .GE. 1 .AND. J .GE. 1) V = X(K)
    IF (I .LT. NX .AND. J .GT. 0) VR = X(K+1)
    IF (I .GT. 0 .AND. J .LT. NY) VU = X(K+NX)
    FQUAD = FQUAD + HXY* (VR-V)**2 + HYX* (VU-V)**2
  10 CONTINUE
20 CONTINUE
C
C LOOP2:
C
DO 40 J = 1,NY + 1

23
DO 30 I = 1, NX + 1
   K = NX* (J-1) + I
   V = 0.0
   VL = 0.0
   VD = 0.0
   IF (I.LE.NX .AND. J.LE.NY) V = X(K)
   IF (I.GT.1 .AND. J.LE.NY) VL = X(K-1)
   IF (I.LE.NX .AND. J.GT.1) VD = X(K-NX)
   FQUAD = FQUAD + HXY* (VL-V)**2 + HXY* (VD-V)**2
30    CONTINUE
40    CONTINUE

c    Computation of the linear part of the function.

c LOOP 3:

FLIN = 0.0
DO 50 K = 1,NX*NY
   FLIN = FLIN + X(K)
50    CONTINUE
F = 0.25*FQUAD - FORCE*HX*HY*FLIN

END
B  Current ADIFOR Code for Torsion Problem

SUBROUTINE REVO(G$P$,N,X,G$X$,LDG$X$,F,G$F$,LDG$F$,NX,NY,HX,HY,FORCE)

******************************************************************************
* * Generated by current ADIFOR for computing gradient of
* torsion problem, x independent, f dependent.
* ******************************************************************************
C
C  ADIFOR: runtime gradient index
C  ADIFOR: translation time gradient index
C  ADIFOR: gradient iteration index
C
C  **********
C  This subroutine computes the function of the torsion problem.
C  **********
C  The spacing parameters hx and hy are for a rectangle with
C  nx points on the x-axis and ny points on the y-axis
C
C  ADIFOR: gradient declarations
C
C  .. Parameters ..
INTEGER G$P$MAX$
PARAMETER (G$P$MAX$=4900)$
C
C  .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER G$P$,LDG$F$,LDG$X$,NX,NY
C
C  .. Array Arguments ..
DOUBLE PRECISION G$F$(LDG$F$),G$X$(LDG$X$,N),X(N)
C
C  .. Local Scalars ..
DOUBLE PRECISION D$O$,D$4$,FLIN,FQUAD,HXY,HYX,V,VD,VL,VR,VU
INTEGER G$S$I$,I,J,K
C
C  .. Local Arrays ..
DOUBLE PRECISION G$F$LIN(G$P$MAX$)$,G$F$QUAD(G$P$MAX$)$,G$S$V(G$P$MAX$)$,
  + G$S$VD(G$P$MAX$)$,G$S$VL(G$P$MAX$)$,G$S$VR(G$P$MAX$)$,
  + G$S$VU(G$P$MAX$)$
C
C  IF (G$P$,GT,G$P$MAX$) THEN
   PRINT *, 'Parameter g$p is greater than g$pmx.'
   STOP
END IF
HXY = HX/HY
HYX = HY/HX
C  Computation of the quadratic part of the function.
FQUAD = 0.0
DO G$IS$ = 1,G$SP$
    G$SFQUAD(G$IS$) = 0.0d0
END DO
DO 99998 J = 0,NY
   DO 99999 I = 0,NX
      K = NX* (J-1) + I
      V = 0.0
      DO G$IS$ = 1,G$SP$
         G$SV(G$IS$) = 0.0d0
      END DO
      VR = 0.0
      DO G$IS$ = 1,G$SP$
         G$SVR(G$IS$) = 0.0d0
      END DO
      VU = 0.0
      DO G$IS$ = 1,G$SP$
         G$SVU(G$IS$) = 0.0d0
      END DO
      IF (I.GE.1 .AND. J.GE.1) THEN
         V = x(k)
         DO G$IS$ = 1,G$SP$
            G$SV(G$IS$) = G$Sx(G$IS$,$k$)
         END DO
         V = x(k)
         END IF
      IF (I.LT.NX .AND. J.GT.0) THEN
         VR = x(k+1)
         DO G$IS$ = 1,G$SP$
            G$SVR(G$IS$) = G$Sx(G$IS$,$k+1$)
         END DO
         VR = x(k+1)
         END IF
      IF (I.GT.0 .AND. J.LT.NY) THEN
         VU = x(k+nx)
         DO G$IS$ = 1,G$SP$
            G$SVU(G$IS$) = G$Sx(G$IS$,$k+nx$)
         END DO
         VU = x(k+nx)
         END IF
      IF (Q2 = 0) THEN
         fquad = fquad + hyx * (vr - v)**2 + hxy * (vu - v)**2
      END IF
      D$O = VR - V
      D$4 = VU - V
      DO G$IS$ = 1,G$SP$
         G$SFQUAD(G$IS$) = G$SFQUAD(G$IS$) + HYX*2*D$O*G$SVR(G$IS$) +
            (HYX*2*D$4)**2 +
            G$SV(G$IS$) + HXY*2*D$4*G$SVU(G$IS$)
END DO
FQUAD = FQUAD + HXY*D$O**2 + HXY*D$4**2
10 CONTINUE
99999 CONTINUE
20 CONTINUE
99998 CONTINUE
DO 99996 J = 1,NY + 1
DO 99997 I = 1,NX + 1
K = NX* (J-1) + I
V = 0.0
DO G$IS$ = 1,G$SP$
G$SV(G$IS$) = 0.0d0
END DO
VL = 0.0
DO G$IS$ = 1,G$SP$
G$SVL(G$IS$) = 0.0d0
END DO
VD = 0.0
DO G$IS$ = 1,G$SP$
G$SVD(G$IS$) = 0.0d0
END DO
IF (I.LE.NX .AND. J.LE.NY) THEN
  C
  v = x(k)
  DO G$IS$ = 1,G$SP$
    G$SV(G$IS$) = G$SV(x(k),K)
  END DO
  V = x(k)
END IF
IF (I.GT.1 .AND. J.LE.NY) THEN
  C
  vl = x(k - 1)
  DO G$IS$ = 1,G$SP$
    G$SVL(G$IS$) = G$SV(x(k-1),K-1)
  END DO
  VL = x(k-1)
END IF
IF (I.LE.NX .AND. J.GT.1) THEN
  C
  vd = x(k - nx)
  DO G$IS$ = 1,G$SP$
    G$SVD(G$IS$) = G$SV(x(k-nx),K-NX)
  END DO
  VD = x(k-nx)
END IF
END IF
C
  fquad = fquad + hxy * (vl - v) ** 2 + hxy * (vd - v) ** 2
D$O = VL - V
D$4 = VD - V
DO G$IS$ = 1,G$SP$
G$SFQUAD(G$IS$) = G$SFQUAD(G$IS$) + HXY*2*D$O*G$SVL(G$IS$) +


```fortran
+ (- (HXY*2*D$4) - (HXY*2*D$0))*
+ G$V(G$S$I$) + HXY*2*D$4*G$SVD(G$S$I$)

END DO
FQUAD = FQUAD + HXY*D$0**2 + HXY*D$4**2

30 CONTINUE

99997 CONTINUE

40 CONTINUE

99996 CONTINUE

C Computation of the linear part of the function,
FLIN = 0.0
Do G$S$I$ = 1,G$S$P$
   G$S$FLIN(G$S$I$) = 0.0d0
End Do
Do 99995 K = 1,NX*NY
   FLIN = FLIN + X(K)
   Do G$S$I$ = 1,G$S$P$
      G$S$FLIN(G$S$I$) = G$S$FLIN(G$S$I$) + G$X(G$S$I$,K)
   End Do
   FLIN = FLIN + X(K)
50 CONTINUE

99995 CONTINUE

C f = 0.25 * fquad - force * hx * hy * flin
Do G$S$I$ = 1,G$S$P$
   G$S$F(G$S$I$) = 0.25*G$S$FQUAD(G$S$I$) - (FORCE*HX*HY)*G$S$FLIN(G$S$I$)
End Do
F = 0.25*FQUAD - FORCE*HX*HY*FLIN
END
```
Reverse Mode for Basic Blocks

SUBROUTINE REV1(G$P$,$N,X,G$X,LDG$X,F,G$F,LDG$F,NX,NY,HX,HY,FORCE)

*******************************************************************************
*                                                                *
* reverse mode at basic block level                                 *
*                                                                            *
*******************************************************************************

C ADIFOR: runtime gradient index
C ADIFOR: translation time gradient index
C ADIFOR: gradient iteration index

********
C This subroutine computes the function of the torsion problem.
C
********
C The spacing parameters hx and hy are for a rectangle with
C nx points on the x-axis and ny points on the y-axis
C
ADIFOR: gradient declarations

.. Parameters ..
INTEGER G$P$MAX$
PARAMETER (G$P$MAX$=10000)

.. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER G$P$,LDG$F,LDG$X,N,NX,NY

.. Array Arguments ..
DOUBLE PRECISION G$F(LDG$F),G$X(LDG$X,N),X(N)

.. Local Scalars ..
DOUBLE PRECISION D$0,D$4,FLIN,FLINBAR,FQUAD,HXY,HYX,V,VBAR,VD,
+ VDBAR,VL,VLBAR,VR,VRBAR,VU,VUBAR,XK1BAR,XK2BAR,
+ XK3BAR
INTEGER G$S$I$,$I,J,K,PP

.. Local Arrays ..
DOUBLE PRECISION G$S$FLIN(G$P$MAX$),G$S$QUAD(G$P$MAX$)

.. IF (G$S$P$.GT,G$P$MAX$) THEN
       PRINT *,’Parameter g$P is greater than g$P$MAX.’
       STOP
END IF
HXY = HX/HY
HYX = HY/HX

Computation of the quadratic part of the function.
FQUAD = 0.0
DO G$IS = 1,G$PS
   G$FQUAD(G$IS) = 0.0d0
END DO
DO 99998 J = 0,NY
   DO 99999 I = 0,NX
      K = NX* (J-1) + I
C
C   compute new contribution to sum
C
      V = 0.0
      VR = 0.0
      VU = 0.0
      IF (I.GE.1 .AND. J.GE.1) THEN
         V = X(K)
      END IF
      IF (I.LT.NX .AND. J.GT.0) THEN
         VR = X(K+1)
      END IF
      IF (I.GT.0 .AND. J.LT.NY) THEN
         VU = X(K+NX)
      END IF
C
      fquad = fquad + hyx * (vr - v) ** 2 + hxy * (vu - v) ** 2
C
C   reverse mode computation for computing derivatives of
C   x(k), x(k+1), x(k+nx). We know that the deriv. of fquad, with
C   respect to fquad, is 1.
C
      D$0 = VR - V
      D$4 = VU - V
      FQUAD = FQUAD + HYX*D$0**2 + HXY*D$4**2
      VBAR = -2*HXY*D$4 - 2*HYX*D$0
      VUBAR = 2*HXY*D$4
      VRBAR = 2*HYX*D$0
      XXBAR = 0.0
      XX1BAR = 0.0
      XXNBAR = 0.0
      IF (I.GE.1 .AND. J.GE.1) XXBAR = VBAR
      IF (I.LT.NX .AND. J.GT.0) XX1BAR = VRBAR
      IF (I.GT.0 .AND. J.LT.NY) XXNBAR = VUBAR
C
C   Chain Rule to update derivatives of fquad w.r.t. x
C
DO PP = 1,G$PS
   G$FQUAD(PP) = G$FQUAD(PP) + XXBAR*G$X(PP,K) +
! - X1BAR*G$X(PP,K+1) +
! + XNBAR*G$X(PP,K+NX)
!
END DO

99999  CONTINUE
20  CONTINUE
99998  CONTINUE

DO 99996 J = 1,NY + 1
   DO 99997 I = 1,NX + 1
      K = NX* (J-1) + I
      
      V = 0.0
      VL = 0.0
      VD = 0.0
      IF (I,LE,NX .AND. J,LE,NY) V = X(K)
      IF (I,GT.1 .AND. J,LE,NY) VL = X(K-1)
      IF (I,LE,NX .AND. J,GT.1) VD = X(K-NX)
      
      C  
      fquad = fquad + hyx * (vl - v) ** 2 + hxy * (vd - v) ** 2
      D$0 = VL - V
      D$4 = VD - V
      FQUAD = FQUAD + HYX*D$0**2 + HXY*D$4**2
      VBAR = -2*HYX*D$4 - 2*HXY*D$0
      VDBAR = 2*HYX*D$4
      VLBAR = 2*HXY*D$0
      XKBAR = 0.0
      X1BAR = 0.0
      XNBAR = 0.0
      IF (I,LE,NX .AND. J,LE,NY) XKBAR = VBAR
      IF (I,GT.1 .AND. J,LE,NY) X1BAR = VLBAR
      IF (I,LE,NX .AND. J,GT.1) XNBAR = VDBAR
      
      DO PP = 1,G$P$ 
         G$FQUAD(PP) = G$FQUAD(PP) + XKBAR*G$X(PP,K) +
         + X1BAR*G$X(PP,K-1) +
         + XNBAR*G$X(PP,K-NX)
      END DO

99997  CONTINUE
99996  CONTINUE

C  
Computation of the linear part of the function.
FLIN = 0.0
DO G$S$I$ = 1,G$P$
   G$S$FLIN(G$S$I$) = 0.040
END DO
DO 99995 K = 1,NX*NY
   G$FLIN(K) = G$FLIN(K) + 1
FLINBAR = 1.0
DO I = 1,G$PS
   GSFLIN(I) = GSFLIN(I) + FLINBAR*G$X(I,K)
END DO
FLIN = FLIN + X(K)
99995 CONTINUE
C     f = 0.25 * fquad - force * hx * hy * flin
DO G$IS = 1,G$PS
   GSF(G$IS) = 0.25*GSFQUAD(G$IS) - (FORCE*HX*HY)*GSFLIN(G$IS)
END DO
F = 0.25*FQUAD - FORCE*HX*HY*FLIN
END
D Reverse Mode for Loop Bodies

SUBROUTINE REV2(G$P$,N,X,G$X$,LDG$X$,F,G$F$,LDG$F$,HX,NX,NY,HY,FORCE,
+ XBAR,XBARSIZE)

***************
* reverse mode at individual loop level
***************
C
  ADIFOR: runtime gradient index
C  ADIFOR: translation time gradient index
C  ADIFOR: gradient iteration index
C
**********
C This subroutine computes the function of the torsion problem.
C
**********
C The spacing parameters hx and hy are for a rectangle with
C nx points on the x-axis and ny points on the y-axis
C
C  ADIFOR: gradient declarations

C .. Parameters ..
INTEGER G$PMAX$
PARAMETER (G$PMAX$=10000)
C ..
C .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER G$P$,LDG$F$,LDG$X$,N,NX,NY,XBARSIZE
C ..
C .. Array Arguments ..
DOUBLE PRECISION G$F$(LDG$F$),G$X$(LDG$X$),X(N),XBAR(*)
C ..
C .. Local Scalars ..
DOUBLE PRECISION D$0$,D$4$,FLIN,FLINBAR,FQUAD,HXY,HYX,V,VBAR,VD,
+ VBAR,VL,VLBAR,VR,VRBAR,VU,VUBAR
INTEGER G$S$I$,$I,$J$,$K
C ..
C .. Local Arrays ..
DOUBLE PRECISION G$FLIN$(G$PMAX$),G$FQUAD$(G$PMAX$)
C ..
C .. External Subroutines ..
EXTERNAL D$G$M$V$
C ..
IF (G$P$.GT.G$PMAX$) THEN
   PRINT *,’Parameter g$p is greater than g$pmx.’
   STOP
END IF

33
HXY = HX/HY
HYX = HY/HX

C Computation of the quadratic part of the function.
FQUAD = 0.0
DO G$S$ = 1,G$F$S
   G$FQUAD(G$S$) = 0.0d0
END DO
DO I = 1,XBARSIZE
   XBAR(I) = 0.0
END DO

DO 99998 J = 0,NY
   DO 99999 I = 0,NX
      K = NX* (J-1) + I
      V = 0.0
      VR = 0.0
      VU = 0.0
      IF (I.GE.1 .AND. J.GE.1) V = X(K)
      IF (I.LT.NX .AND. J.GT.0) VR = X(K+1)
      IF (I.GT.0 .AND. J.LT.NY) VU = X(K+NX)
      C fquad = fquad + hyx * (v - v) ** 2 + hxy * (vu - v) ** 2
      D$S = VR - V
      D$S4 = VU - V
      FQUAD = FQUAD + HYX*D$S**2 + HXY*D$S4**2
      VBAR = -2*HYX*D$S4 - 2*HYX*D$S
      VBARY = 2*HYX*D$S4
      VRBAR = 2*HYX*D$S
      IF (I.GE.1 .AND. J.GE.1) XBAR(K) = XBAR(K) + VBAR
      IF (I.LT.NX .AND. J.GT.0) XBAR(K+1) = XBAR(K+1) + VRBAR
      IF (I.GT.0 .AND. J.LT.NY) XBAR(K+NX) = XBAR(K+NX) + VRBAR
      99999 CONTINUE
   20 CONTINUE
99998 CONTINUE

C xbar is the vector of partial derivatives of the contribution
C to fquad with respect to x. Since fquad was zero before this
C loop, the derivative d$quad = g$x * xbar.

C CALL DGMEM('n',G$P$,XBARSIZE,1.0d0,G$X$,LDBG$X$,XBAR,1,1.0d0,G$FQUAD,
   + 1)

DO I = 1,XBARSIZE
   XBAR(I) = 0.0
END DO

DO 99996 J = 1,NY + 1
   DO 99997 I = 1,NX + 1
      K = NX* (J-1) + I
V = 0.0
VL = 0.0
VD = 0.0
IF (I.LE.NX .AND. J.LE.NY) V = X(K)
IF (I.GT.1 .AND. J.LE.NY) VL = X(K-1)
IF (I.LE.NX .AND. J.GT.1) VD = X(K-NX)
C                  fquad = fquad + hx * (vl - v) ** 2 + hxy * (vd - v) ** 2
D$0 = VL - V
D$4 = VD - V
FQUAD = FQUAD + HYX*D$0**2 + HXY*D$4**2
VBAR = -2*HXY*D$4 - 2*HYX*D$0
VDBAR = 2*HXY*D$4
VLBAR = 2*HYX*D$0
IF (I.LE.NX .AND. J.LE.NY) XBAR(K) = XBAR(K) + VBAR
IF (I.GT.1 .AND. J.LE.NY) XBAR(K-1) = XBAR(K-1) + VLBAR
IF (I.LE.NX .AND. J.GT.1) XBAR(K-NX) = XBAR(K-NX) + VDBAR
99997  CONTINUE
99996  CONTINUE
C
C xbar is the vector of partial derivatives of the contribution
C to fquad with respect to x. Since fquad was already initialized
C before this loop, the derivative d$fquad = d$fquad + g$x * xbar.
C
CALL DGEVM('n',G$P$X,XBARSZ1,1.0d0,G$X,LDG$X,XBAR,1,1.0d0,G$FQUAD,+
                1)
C
Computation of the linear part of the function,
FLIN = 0.0
DG G$S$I$ = 1,G$S$P$X
G$FLIN(G$S$I$) = 0.0d0
END DO
DG I = 1,XBARSZ1
XBAR(I) = 0.0
END DO

DG 99995  K = 1,NX*NY
C
C g$flin(k) = g$flin(k) + 1
FLINBAR = 1.0
XBAR(K) = XBAR(K) + FLINBAR
FLIN = FLIN + X(K)
99995  CONTINUE
C
C again, d$fquad = d$fquad + g$x * xbar.
C
CALL DGEVM('n',G$P$X,XBARSZ1,1.0d0,G$X,LDG$X,XBAR,1,1.0d0,G$FLIN,+
                1)
C
f = 0.25 * fquad - force * hx * hy * flin

35
DO G\$I\$ = 1,G\$F
    G\$F(G\$I\$) = 0.25*G\$FQUAD(G\$I\$) - (FORCE*HX*HY)*G\$FLIN(G\$I\$)
END DO
F = 0.25*FQUAD - FORCE*HX*HY*F\$LIN
END
E Reverse Mode for the Whole Program

SUBROUTINE REV3(N,X,F,NX,NY,NX,HY,FORCE,XBAR,XBARSIZE)
C ADIFOR: runtime gradient index
C ADIFOR: translation time gradient index
C
**********
C This subroutine computes the function of the torsion problem.
C **********
C The spacing parameters hx and hy are for a rectangle with
C nx points on the x-axis and ny points on the y-axis
C
C ADIFOR: gradient declarations
C
.. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER N,NX,NY,XBARSIZE
C
.. Array Arguments ..
DOUBLE PRECISION X(N),XBAR(*)
C
.. Local Scalars ..
DOUBLE PRECISION D$0,D$4,FLIN,FLINBAR,FQUAD,HYX,T,V,VBAR,VD,
+ VDBAR,VL,VLBAR,VR,VRBAR,VU,VUBAR
INTEGER I,J,K
C
.. HX = HX/HY
HYX = HY/HX
C
Computation of the quadratic part of the function.
FQUAD = 0.0
DO I = 1,XBARSIZE
  XBAR(I) = 0.0
END DO
DO 99998 J = 0,NY
  DO 99999 I = 0,NX
    K = NX* (J-1) + I
    V = 0.0
    VR = 0.0
    VU = 0.0
    IF (I.GE.1 .AND. J.GE.1) V = X(K)
    IF (I.LT.NX .AND. J.GT.0) VR = X(K+1)
    IF (I.GT.0 .AND. J.LT.NY) VU = X(K+NX)
    fquad = fquad + hyx * (vr - v) ** 2 + hyx * (vu - v) ** 2
    D$0 = VR - V
    D$4 = VU - V
    FQUAD = FQUAD + HYX*D$0**2 + HYX*D$4**2
    VBAR = -2*HYX*D$4 - 2*HYX*D$0

37
VUBAR = 2*HXY*D$4
VRBAR = 2*HXY*D$0
IF (I,GE,1 .AND. J,GE,1) XBAR(K) = XBAR(K) + VBAR
IF (I,LT,NX .AND. J,GT,0) XBAR(K+1) = XBAR(K+1) + VRBAR
IF (I,GT,0 .AND. J,LT,NY) XBAR(K+NX) = XBAR(K+NX) + VUBAR

99999  CONTINUE
99998  CONTINUE

DO 99996 J = 1,NY + 1
  DO 99997 I = 1,NX + 1
    K = NX* (J-1) + I
    V = 0.0
    VL = 0.0
    VD = 0.0
    IF (I,LE,NX .AND. J,LE,NY) V = X(K)
    IF (I,GT,1 .AND. J,LE,NY) VL = X(K-1)
    IF (I,LE,NX .AND. J,GT,1) VD = X(K-NX)
    C
    fquad = fquad + hyx * (VL - V) ** 2 + hxy * (VD - V) ** 2
    D$0 = VL - V
    D$4 = VD - V
    FQUAD = FQUAD + HXY*D$0**2 + HXY*D$4**2
    VBAR = -2*HXY*D$4 - 2*HXY*D$0
    VBAR = 2*HXY*D$4
    VLBAR = 2*HXY*D$0
    IF (I,LE,NX .AND. J,LE,NY) XBAR(K) = XBAR(K) + VBAR
    IF (I,GT,1 .AND. J,LE,NY) XBAR(K-1) = XBAR(K-1) + VLBAR
    IF (I,LE,NX .AND. J,GT,1) XBAR(K-NX) = XBAR(K-NX) + VDBAR
  99997  CONTINUE
99998  CONTINUE
C  Computation of the linear part of the function.
  FLIN = 0.0
  T = - (FORCE*HX*HY)
  all the flinbar(k)'s are equal to 1.0
  FLINBAR = 1.0
  DO 99995 K = 1,NX*NY
    XBAR(K) = (0.25*XBAR(K)) + (T*FLINBAR)
    FLIN = FLIN + X(K)
  99995  CONTINUE
F = 0.25*FQUAD - FORCE*HX*HY*FLIN
END
F Derivative Code for Component Functions

SUBROUTINE REVOA(G$P$,N,X,G$X$,LDG$X$,F,NX,NY,HX,HY,FORCE,FQ,G$FQ$, + LDG$FQ$)

**********
* derivative of first loop -- current ADIFOR
**********
C
C Formal fq is active.
C Formal x is active.
C
C .. Parameters ..
INTEGER G$P$MAX$
PARAMETER (G$P$MAX$=100)
C ..
C .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER G$P$,$LDG$FQ$,LDG$X$,N,NX,NY
C ..
C .. Array Arguments ..
DOUBLE PRECISION FQ(*),G$FQ$(LDG$FQ$,*)\(,G$X$(LDG$X$,N),X(N)
C ..
C .. Local Scalars ..
DOUBLE PRECISION D$0$,D$\bar{0}$BAR,D$3$,D$3$BAR,FQUAD,HXY,HXY,V,VR,VU
INTEGER FK,G$B$1,J,I,K
C ..
C .. Local Arrays ..
DOUBLE PRECISION G$V$(G$P$MAX$),G$VR$(G$P$MAX$),G$VU$(G$P$MAX$)
C ..
IF (G$P$$.GT.$G$P$MAX$) THEN
   PRINT *,'Parameter g$p$ is greater than g$pmax.'
   STOP
END IF
HXY = HX/HY
HXY = HY/HX
C Computation of the quadratic part of the function.
FQUAD = 0.0
DO 99998 J = 0,NY
   DO 99999 I = 0,NX
      K = NX* (J-1) + I
      V = 0.0
      DO 99990 G$B$1 = 1,G$P$
         G$V$(G$B$1$) = 0.0d0
 99990       CONTINUE
      VR = 0.0
      DO 99989 G$B$1 = 1,G$P$
         G$V$R(G$B$1$) = 0.0d0
 99989   CONTINUE
 99999   CONTINUE
 99998   CONTINUE

39
99989 CONTINUE
VU = 0.0
DO 99988 G$IS$ = 1,G$PS$
     G$SU(G$IS$) = 0.0d0
99988 CONTINUE
IF (I.GE.1 .AND. J.GE.1) THEN
   V = x(k)
   DO 99987 G$IS$ = 1,G$PS$
     G$SV(G$IS$) = G$X(G$IS$),K)
99987 CONTINUE
END IF
IF (I.LT.NX .AND. J.GT.0) THEN
   VR = x(k+1)
   DO 99986 G$IS$ = 1,G$PS$
     G$SVR(G$IS$) = G$X(G$IS$),K+1)
99986 CONTINUE
VR = x(K+1)
END IF
IF (I.GT.0 .AND. J.LT.NY) THEN
   RU = x(k + nx)
   DO 99985 G$IS$ = 1,G$PS$
     G$SUV(G$IS$) = G$X(G$IS$),K+NX)
99985 CONTINUE
RU = x(K+NX)
END IF
C m fquad = fquad + hyx*(vr-v)**2 + hxy*(vu-v)**2
     FQK = (J* (NX+1)) + I + 1
C fq(fqk) = hyx* (vr - v) ** 2 + hxy * (vu - v) ** 2
     D$G = VR - V
     D$S = VU - V
     D$SBAR = HXY* (2*D$S)
     D$OBAR = HXY* (2*D$G)
     DO 99984 G$IS$ = 1,G$PS$
     G$FQ(G$IS$,FQK) = D$OBAR*G$SVR(G$IS$) +
       + (-D$3BAR + (-D$OBAR))*G$SV(G$IS$) +
         D$3BAR*G$SU(G$IS$)
99984 CONTINUE
FQ(FQK) = HXY*D$S**2 + HXY*D$3**2
10 CONTINUE
99999 CONTINUE
20 CONTINUE
99998 CONTINUE
END
SUBROUTINE REVOB(G$PS,N,X,G$S,X,LDG$X,F,NX,NY,HY,FORCE,FQQ,G$FQQ, 
  + 
LDG$FQQ)

*******
* derivs for second loop, current ADIFOR
*******
C
C  Formal fqq is active.
C  Formal x is active.
C
C .. Parameters ..
INTEGER G$P$MAX$
PARAMETER (G$P$MAX$=100)
C ..
C .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,NX,NY
INTEGER G$PS,LDG$FQQ,LDG$X,N,NX,NY
C ..
C .. Array Arguments ..
DOUBLE PRECISION FQQ(*),G$FQQ(LDG$FQQ,*),G$S(X(LDG$X,X),X(N))
C ..
C .. Local Scalars ..
DOUBLE PRECISION D$0,D$OBAR,D$3,D$3BAR,HXY,HYX,V,VD,VL
INTEGER FQK,G$IS$,I,J,K
C ..
C .. Local Arrays ..
DOUBLE PRECISION G$V(G$P$MAX$),G$VD(G$P$MAX$),G$VL(G$P$MAX$)
C ..
IF (G$PS.GT.G$P$MAX$) THEN
  PRINT *,'Parameter g$P$ is greater than g$P$max.'
  STOP
END IF
HXY = HX/NY
HYX = HY/NX
DO 99998 J = 1,NY + 1
  DO 99999 I = 1,NX + 1
    K = NX* (J-1) + I
    V = 0.0
    DO 99990 G$IS$ = 1,G$PS$
      G$V(G$IS$) = 0.0d0
  99990 CONTINUE
    VL = 0.0
    DO 99989 G$IS$ = 1,G$PS$
      G$VL(G$IS$) = 0.0d0
  99989 CONTINUE
    VD = 0.0
    DO 99988 G$IS$ = 1,G$PS$
  99999 CONTINUE
  99998 CONTINUE
G$VD(G$IS$) = 0.0d0
C
99988 CONTINUE
IF (I.LE.NX .AND. J.LE.NY) THEN
C
99987 CONTINUE
v = x(k)
   DO 99987 G$IS$ = 1, G$PS$
   G$Y(G$IS$) = G$X(G$IS$,K)
99987 CONTINUE
V = X(K)
END IF
IF (I.GT.1 .AND. J.LE.NY) THEN
C
99986 CONTINUE
vl = x(k - 1)
   DO 99986 G$IS$ = 1, G$PS$
   G$VL(G$IS$) = G$X(G$IS$,K-1)
99986 CONTINUE
VL = X(K-1)
END IF
IF (I.LE.NX .AND. J.GT.1) THEN
C
99985 CONTINUE
vd = x(k - nx)
   DO 99985 G$IS$ = 1, G$PS$
   G$VD(G$IS$) = G$X(G$IS$,K-NX)
99985 CONTINUE
VD = X(K-NX)
END IF
C
m = fquad = fquad + hxy*(v1-v)**2 + hxy*(vd-v)**2
FQK = ((J-1)*(NX+1)) + I
C
99984 CONTINUE
fql(fqk) = hxy * (v1 - v) ** 2 + hxy * (vd - v) ** 2
D$0 = VL - V
D$3 = VD - V
D$3BAR = HXY* (2*D$3)
D$0BAR = HXY* (2*D$0)
   DO 99984 G$IS$ = 1, G$PS$
   G$FQG(G$IS$,FQK) = D$0BAR*G$VL(G$IS$) +
   + (-D$3BAR* (-D$0BAR))*G$V(G$IS$) +
   + D$3BAR*G$VD(G$IS$)
99984 CONTINUE
FQQ(FQK) = HXY*D$0**2 + HXY*D$3**2
30 CONTINUE
99999 CONTINUE
40 CONTINUE
99998 CONTINUE
END
SUBROUTINE REV1C(GS$N,X,G$X,LDG$X,F,NX,N,Y,HX,HY,FORCE,FP,RGFP,+
     LDG$FP)

***************
* third loop, basic block reverse mode
***************

C .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER GS$N,LDG$FP,LDG$X,N,NX,NY
C ..
C .. Array Arguments ..
DOUBLE PRECISION FP(*),G$(LDG$X,*),RGFP(LDG$FP,*),X(N)
C ..
C .. Local Scalars ..
DOUBLE PRECISION FPBAR,HXY,HYX
INTEGER I,K
C ...
HXY = HX/HY
HYX = HY/HX
C Computation of the linear part of the function.
DO 50 K = 1,NX*NY
     FPBAR = 1.0
     RGFP(I,K) = 0.0
     DO I = 1,GS$N
          RGFP(I,K) = RGFP(I,K) + FPBAR*G$(I,K)
     END DO
     FP(K) = X(K)
50 CONTINUE

END
G Enhanced Derivative Code for Component Functions

SUBROUTINE REV1A(G$P$,X,X,LDG$X,F,NX,NY,NX,NY,FORCE,FQ,REFQ,
                  LDG$FQ$)
******
* first loop contribution, basic block reverse mode
******
C ADIFOR: runtime gradient index
C ADIFOR: translation time gradient index
C ADIFOR: gradient iteration index
C The spacing parameters hx and hy are for a rectangle with
C nx points on the x-axis and ny points on the y-axis
C ADIFOR: gradient declarations
C .. Parameters..
INTEGER G$P$MAX$
PARAMETER (G$P$MAX$=49000)
C ..
C .. Scalar Arguments..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER G$P$,LDG$FQ$,LDG$X$,N,NX,NY
C..
C .. Array Arguments..
DOUBLE PRECISION FQ(*),G$X$(LDG$X$,N),RGFQ(LDG$FQ$,*)X(N)
C..
C .. Local Scalars..
DOUBLE PRECISION D$O,D$4,FQUAD,HX,HY,V,VR,VRBAR,VU,VUBAR,
                  XXXBAR,XXBAR,XXNXBAR
INTEGER FQK,I,J,K,FP
C..
IF (G$P$,GT,G$P$MAX$) THEN
   PRINT *, 'Parameter g$P$ is greater than g$P$max.'
   STOP
END IF

HX = HX/HY
HY = HY/HX

c Computation of the quadratic part of the function, a
c the following is not needed
c do g$i$ = 1, g$P$
   do j = 1, ((nx+1)*(ny+1))
      rgfQ(g$i$,j) = 0.0d0
   enddo
enddo
FQUAD = 0.0
DO 20 J = 0,NY
   DO 10 I = 0,NX
      K = NX* (J-1) + I
      V = 0.0
      VR = 0.0
      VU = 0.0
      IF (I,GE,1 .AND. J,GE,1) V = X(K)
      IF (I,LT,NX .AND. J,GT,0) VR = X(K+1)
      IF (I,GT,0 .AND. J,LT,NY) VU = X(K+NX)
      FQK = (J* (NX+1)) + I + 1
      FQ(FQK) = HYX* (VR-V)**2 + HXY* (VU-V)**2
      D$0 = VR - V
      D$4 = VU - V
      FQ(FQK) = HYX*D$0**2 + HXY*D$4**2
      VBAR = -2*HXY*D$4 - 2*HYX*D$0
      VUBAR = 2*HXY*D$4
      VRBAR = 2*HYX*D$0
      XBAR = 0.0
      XBAR = 0.0
      XNBAR = 0.0
      IF (I,GE,1 .AND. J,GE,1) XBAR = VBAR
      IF (I,LT,NX .AND. J,GT,0) XBAR = VRBAR
      IF (I,GT,0 .AND. J,LT,NY) XNBAR = VUBAR
   END DO
   DO PP = 1,GSP$2
      RGFQ(P,P,FQK) = XBAR*G$X(PP,K) + XBAR*G$X(PP,K+1) +
                       XNBAR*G$X(PP,K+NX)
   END DO
20 CONTINUE
END
SUBROUTINE REV1B(G$P$,N,X,G$X,LDG$X,F,NX,NY,HY,HX,FORCE,FQQ,RGFQQ,
+ LGDG$FQQ)

************
* second loop, basic block reverse mode
************
C
C ADIFOR: runtime gradient index
C ADIFOR: translation time gradient index
C ADIFOR: gradient iteration index
C
C The spacing parameters hx and hy are for a rectangle with
C nx points on the x-axis and ny points on the y-axis
C
C ADIFOR: gradient declarations
C
C .. Parameters ..
INTEGER G$P$MAX$
PARAMETER (G$P$MAX$=4900)
C ..
C .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER G$P$,LDG$FQQ,LDG$X,NX,NY
C ..
C .. Array Arguments ..
DOUBLE PRECISION FQQ(*),G$X(LDG$X,N),RGFQQ(LDG$FQQ,*)X(N)
C ..
C .. Local Scalars ..
DOUBLE PRECISION D$0,D$4,FQUAD,HXY,HYX,VBAR,VDBAR,VL,VLBAR,
+ X1BAR,X2BAR,XN$X$BAR
INTEGER FQK,I,J,K,FP
C ..
IF (G$P$,GT,G$P$MAX$) THEN
  PRINT *, 'Parameter g$P$ is greater than g$P$max.'
  STOP
END IF
HXY = HX/HY
HYX = HY/HX
C
Computation of the quadratic part of the function.
C the following is NOT needed
C do g$i$ = 1, g$P$
C do j = 1, (nx+1)*(ny+1))
C rgfQQ(g$i$,j) = 0.0d0
C enddo
C enddo
FQUAD = 0.0
DO 40 J = 1,NY + 1
DO 30 I = 1,NX + 1
   **K = NX* (J-1) + I**
   V = 0.0
   VL = 0.0
   **VD = 0.0**
   IF (I.LE.NX .AND. J.LE.NY) V = X(K)
   IF (I.GT.1 .AND. J.LE.NY) VL = X(K-1)
   IF (I.LE.NX .AND. J.GT.1) VD = X(K-NX)

   fquad = fquad + hyx*(vl-v)**2 + hxy*(vd-v)**2
   FQK = ((J-1)* (NX+1)) + I
   FQQ(FQK) = HYX* (VL-V)**2 + HXY* (VD-V)**2
   D$s0 = VL - V
   D$s4 = VD - V
   FQQ(FQK) = HYX*D$s0**2 + HXY*D$s4**2
   VBAR = -2*HYX*D$s4 - 2*HYX*D$s0
   VLBAR = 2*HYX*D$s0
   VDBAR = 2*HYX*D$s4
   XKBAR = 0.0
   XX1BAR = 0.0
   **XXNXBAR = 0.0**
   IF (I.LE.NX .AND. J.LE.NY) XKBAR = VBAR
   IF (I.GT.1 .AND. J.LE.NY) XX1BAR = VLBAR
   IF (I.LE.NX .AND. J.GT.1) XXNXBAR = VDBAR

   DO PP = 1,GS$+$
      RGFQ(PP,FQK) = XKBAR*GSX(PP,K) + XX1BAR*GSX(PP,K-1) +
                      XXNXBAR*GSX(PP,K-NX)
   **END DO**

30 CONTINUE
40 CONTINUE
END
SUBROUTINE REV1C(G$SP$N,X,G$X,LDG$X,F,NX,NY,HX,HY,FORCE,FP,RGFP, 
   +   LDG$FP)

***************
* third loop, basic block reverse mode
***************

C .. Scalar Arguments ..
DOUBLE PRECISION F,FORCE,HX,HY
INTEGER G$SP$,LDG$FP,LDG$X,N,NX,NY
C ..
C .. Array Arguments ..
DOUBLE PRECISION FP(*),G$(LDG$X,*) ,RGFP(LDG$FP,*) ,X(N)
C ..
C .. Local Scalars ..
DOUBLE PRECISION FPBAR,HX,YX
INTEGER I,K

C Computation of the linear part of the function.

DO 50 K = 1,NX*NY
   FPBAR = 1.0
   RGFP(I,K) = 0.0
   DO I = 1,G$SP$
      RGFP(I,K) = RGFP(I,K) + FPBAR*G$(I,K)
   END DO
   FP(K) = X(K)
50 CONTINUE

END