ADIFOR Working Note #3: ADIFOR Exception Handling

Christian H. Bischof
George Corliss
Andreas Griewank

CRPC-TR92234
January 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

ADIFOR Working Note #3:

ADIFOR Exception Handling

by

Christian Bischof, George Corliss, and Andreas Griewank

Abstract

Automatic differentiation uses recurrence relations based on the rules of calculus. Consequently, the results are guaranteed to be correct only if the relevant mathematical assumptions are satisfied at least in a neighborhood of the current argument. Computer programs may violate these conditions by branching or by calling intrinsic functions such as abs, max, sqrt, and asin at points where their derivative is undefined or infinite. The resulting dependence between the program's input and output variables may still be differentiable, because branch values fit together smoothly or nondifferentiabilities cancel each other out.

We have two objectives. First, we would like to assure the user that the function being evaluated is indeed locally differentiable because all intrinsics are evaluated at smooth arguments and none of the branching tests are critical. Second, the derivative program should run even when the assumptions of the chain rule are not strictly satisfied. In this case, the numerical results represent at least generalized derivatives under reasonable (but usually unverifiable) regularity assumptions. To achieve these two goals, we must take into account the effects of finite-precision arithmetic.

This paper addresses the detection and handling of exceptions. It is an exception in the ADIFOR-generated code to evaluate a function at a point at which the function may not be mathematically differentiable. When an exception is detected by tests written into the ADIFOR-generated code, an error handler is called. The error handler prints an error message (optionally), halts execution (optionally), and returns a value that allows the user's client program to detect that a requested derivative is not available.

Code is included for all of the necessary Fortran intrinsic functions and for the error handler.

1 Introduction

Automatic differentiation is based on the application of the chain rule. It gives the correct answer, provided that all operators and functions are applied at arguments interior to their domains, so that the operators and functions are smooth in a neighborhood of the point of application. If the relevant mathematical assumptions are not satisfied, the results computed by the ADIFOR-generated code cannot be guaranteed. That is, the code generated by ADIFOR computes the correct values of the derivatives almost all of the time. This paper discusses what happens in the remaining *rare* events.

The purpose of this paper is to make explicit the issues and alternatives associated with exception handling in ADIFOR. We assume that the reader is familiar with ADIFOR [1, 2]. The intended audience of this paper is the user of ADIFOR who wishes to better understand the error handling provided by ADIFOR and the rationale behind it.

We address three questions:

1. What is an "error?"

2. How can we detect that an error has occurred or is about to occur?

3. What action should we take when an error is detected?

We attempt to explicitly state reasonable alternatives.

The error-handling mechanisms must be powerful, flexible, portable, efficient, and easy to use. In the end, the effectiveness of the error handling is a major factor in the reliability of ADIFOR.

2 Definition of an Error

The (unreachable) goal of the error handling of ADIFOR is to provide a measure of reliability whenever the derivatives computed by the ADIFOR-generated code cannot be guaranteed to be correct. Hence, we consider it an "error" if we cannot assure the user that ADIFOR has computed the correct values for the derivatives. Unfortunately, it is too expensive to detect all errors in this broad class. We view it as our responsibility to detect and handle in a sensible manner attempts to evaluate derivatives at points of mathematical nondifferentiability. We do not attempt to handle all overflow errors.

If

- all arithmetic operations and intrinsic functions are differentiable on some neighborhood of the current argument,
- the sequence of these elementary functions (the flow of control) is the same at nearby points, and
- computations are performed in exact arithmetic,

then the overall function is locally differentiable, and ADIFOR computes the correct value of the derivatives.

If the elementary functions and their derivatives are evaluated to working accuracy, and no overflow or underflow occurs, then the Jacobian columns obtained in the forward mode represent the corresponding exact first derivatives for a function defined by the same sequence of elementary functions, but with their values and derivatives perturbed by a multipliers of the form $(1+\varepsilon)^2$, where ε is bounded by the relative machine precision.

If these assumptions are not satisfied, then the derivatives computed by ADIFOR may be correct, or they may be wrong. It is the function of the exception-handling mechanism to detect when these assumptions are violated, and to take "appropriate" action.

We discuss four classes of "error":

1. User function is not defined, for example, due to a division by 0. This is not our problem.
2. Differentiable functions — overflows, for example, exp (large number). Detecting this class of error is too expensive.
3. Nondifferentiable functions — $\lim f' = \pm\infty$, for example, the derivative of ASIN at 1. We return some special, user-defined value.
4. Nonsmooth functions — $\lim f'$ does not exist, for example, the derivative of ABS at 0. We return some special, user-defined value.

We discuss each of these classes of error in Sections 5 through 8. Error classes 3 and 4 constitute the core of this paper. In Sections 3 and 4, we discuss options for how we detect an error and what actions we take when one occurs.

Several conflicting principles were considered in designing the ADIFOR exception-handling mechanism:

Generalized Gradient: Many algorithms for optimizing nonsmooth functions use generalized gradient values. A generalized gradient is any value in the convex hull of derivative values in the neighborhood of the point of nondifferentiability. For univariant functions, one may obtain any value in the interval $[\liminf f', \limsup f']$. For example, a generalized gradient for $|x|$ at 0 is any number in $[-1, 1]$. The values we choose to return as "derivative" values at points of nondifferentiability are generalized gradient values, provided that the chain rule for generalized gradients holds as a set inequality, rather than as an inclusion [3].

Continuity of Catastrophe: The value at the point of nondifferentiability should in some sense be the limit of what happens in a neighborhood. For example, the derivative of asin (x) at 1 should be INFINITY. For some functions, the mathematical limit may be different from the computational limit, as a result of finite precision or denormalized numbers.

Extreme Point: A necessary condition for the existence of an extreme point is $f' = 0$. A point of nondifferentiability is usually at least a local extreme point, so returning a value of 0 as the derivative may signal an optimization algorithm that an extreme point has been found.

Scaling: It is critical to scale many applications appropriately before applying an optimization or ODE-solving algorithm. For example, one might scale by | something |, maximum, or minimum. The derivative is locally not defined, but the entire computation is globally differentiable. We have attempted to return derivative values that make sense in connection with commonly used scaling techniques.

Evaluation of Undefined Functions: In some computing environments, execution may continue after an attempt to evaluate a function at a point outside its domain (perhaps with a value of NaN). If the program has not crashed while evaluating $\sqrt{-2.0}$ (in real arithmetic), then our derivative evaluation should not crash, either.

These principles clearly conflict with one another. We made trade-off choices that we think can be justified.

The idea of scaling is common in scientific computation. For example, suppose that some numerical routine evaluates a function $y = f(x)$ that is homogeneous, namely, $f(cx) = cf(x)$ for any scalar multiplier c. Then one might prefer to compute numerically $y = ||x|| f(x/||x||)$, where $|| \cdot ||$ could denote any vector norm including the nonsmooth ones mentioned above. Now, we find that analytically

$$y' = ||x||' \cdot f(x/||x||) + ||x|| \cdot \nabla f(x/||x||) \cdot (x'/||x|| - x \cdot ||x||'/||x||^2).$$

The first and the last term cancel because $f(x) = \nabla f(x) \cdot x$ for homogeneous functions, where $*$ denotes the dot product. However, this is true only if the two derivative (vectors) $||x||'$ coincide, that is, are defined consistently. Even then, they will be multiplied by the difference of two numbers, which are theoretically identical but will cancel only up to round-off in finite precision. Therefore, we gain by setting the undefined values for the derivatives of abs and min or max to zero.

3 Detection of an Exception

We see the following alternatives for the detection of exceptions (in order of increasingly defensive posture):

Options:

I. Rely on the computing environment.

II. Provide tests and special handling for nondifferentiable functions.

III. Compute a measure of "relative safety" from undefined derivatives.

IV. Compute a measure of "relative safety" from undefined derivatives and overflows.

V. Compute a "trust region" in terms of the independent variables.

VI. Fully guard derivative computations against undefined derivatives.

VII. Fully guard derivative computations against undefined derivatives and overflows.

"Errors" may arise either from functions that are not differentiable or from effects of finite computer arithmetic. In this paper, we address Options I and II, which cover the handling of exceptional events. Options III–VII are concerned with quantifying how close an argument is to the boundary of the domain of differentiability. We shall address those issues in a later paper.

Issues related to error detection are discussed in this section. Issues related to the handling of the errors once they are detected are discussed in the next section.

An advantage of the Fortran-to-Fortran source translation nature of the ADIFOR tool is that the user has the Fortran source code and can edit it, if necessary to handle special situations.

3.1 Reliance on the Computing Environment

Arithmetic errors that occur in the evaluation of derivatives should be treated in exactly the same way as similar errors that arise in the evaluation of the user's original code. Depending on the host computing environment, the user may choose to set traps, test for IEEE arithmetic flags, or take other defensive measures against overflows, divisions by zero, or similar errors caused either by the user's function not being mathematically defined in exact arithmetic or else by effects of finite-precision arithmetic. Error detection of this type for the user's function is clearly the responsibility of the user. As a by-product, the same detection applies to operations performed in evaluating the derivatives.

This option is easy to implement, and it is a part of the error-handling strategy of ADIFOR. However, it is not portable, since the same program behaves differently in different computing environments. Worse, it ignores nondifferentiability caused by branches in the program or by non-differentiable functions such as abs.

The use of IEEE arithmetic fits into this class of error detection. If the host computing environment uses IEEE arithmetic and an error occurs in the evaluation of derivative values, then the IEEE arithmetic provides several different mechanisms for informing the user of that fact.

3.2 Tests and Special Handling for Nondifferentiable Functions

The Fortran intrinsic functions abs, sign, aint, max, min, and dim have points at which they are nondifferentiable. It is a simple matter for ADIFOR to detect calls to these functions and to generate special code to handle the points of nondifferentiability.

4 Exception-Handling Module

In the preceding section, we described options for *detecting* the presence of an error in the derivative evaluation. In this section, we describe a mechanism for taking action and what actions should be taken. In Sections 5–8, we specify that whenever an error is detected, we should call the exception-handling module described here. Fortran code for a simple implementation is given in Appendix A.

The error handler can

- initialize error handler (optional),
- print an error message (if desired),
- either STOP execution, or else return a value to let execution continue. On a machine with IEEE arithmetic, either NaN or INFINITY are logical choices. Then, the user's client program could detect that a requested derivative is not meaningful and take an appropriate action.

- report on the number of errors of each class (optional).

The user can control

ClassN: Class of error to which the error belongs

MsgTxt: Content of the message text

PrintF: Printing of the message

ErrFil: Logical unit number to which the error messages should be printed.

HaltFg: Halting execution

InfVal: Value returned if derivative limit is infinite

NoLmVl: Value returned if derivative has no limit

TieVal: Partial derivative of $\max(x, y)$ or $\min(x, y)$ with respect to x when $x = y$

If the user has complicated exception-handling requirements, we supply the code for `g$error` so that it can be customized as necessary.

We generate augmented code of the form

```
r$0 = funct (t)
if (t  .ne. BAD_POINTS) then
   g$r$0 = result of derivative calculations
else
   g$r$0 = g$error (n, 'Derivative of funct does not exist')
end if
y = r$0
```

The function `g$error` has the form

```
function g$error (Class_Number, Message_Text)
common / g$error_block / Print_Flag, Halt_Flag, Error_File,
             Infinite_Value, No_Limit_Value, Tie_Value
if (Print_Flag)
   write (Error_File) Message_Text
if (Halt_Flag)
  stop
if (Class_Number .lt. 10) then
   g$error = Infinite_Value
elseif (Class_Number .lt. 15) then
   g$error = Tie_Value
return
end
```

The variables `Print_Flag`, `Halt_Flag`, `Error_File`, `Infinite_Value`, `No_Limit_Value`, and `Tie_Value` belong to a common block `g$error_block`. Appropriate default values are provided.

Variable	Default Value
Print_Flag	TRUE
Halt_Flag	FALSE
Error_File	6
Infinite_Value	0
No_Limit_Value	0
Tie_Value	1/2

Several conflicting principles listed in Section 2 were considered in selecting these default values. We made choices for these default values that we think can be justified. The rationale is included with the discussions in the following sections where the values are used. A knowledgeable user may change the values directly or calling the routine we supply:

```
subroutine g$Init_Error (Print_F, Halt_F, Err_F, Infin, No_Lim, Max_V)
boolean Print_F, Halt_F, Err_F
real Infin, No_Lim
common / g$error_block / Print_Flag, Halt_Flag, Error_File,
             Infinite_Value, No_Limit_Value, Tie_Value
Print_Flag     = Print_F
Halt_Flag      = Halt_F
Error_File     = Err_F
Infinite_Value = Infin
No_Limit_Value = No_Lim
Tie_Value  = Max_V
return
end
```

The user who wishes even more control can write his own `function g$error` to handle errors in any way he sees fit.

A `subroutine g$ReptEr (LUnitN)` is provided to optionally report the cumulative number of errors of each class.

5 Error Class 1: Undefined User Function

5.1 Definition of an Error

According to the definition of a derivative, a function must have a finite real value in order to have a derivative. The first class of errors we consider is the case where the original program ADIFOR receives from the user cannot be evaluated at certain arguments. This may happen either because the mathematical function the user has described is undefined (e.g., Detecting $x/0$) or because the mathematical function is well defined, but it cannot be evaluated accurately in finite-precision arithmetic according to the algorithm the user has programmed (e.g., x/ε^2). Currently, ADIFOR can only handle real arithmetic. Hence, the square root is considered to be undefined at negative values.

5.2 Detection of an Error

Options:

I. Let the user's program crash.

II. Augment the user's program with tests to detect *a priori* erroneous conditions such as division by zero. ADIFOR would generate tests similar to that shown in the code fragment below.

5.3 Possible Actions

Recommendation: Let the user's program crash.
Rationale: We should not alter the workings of the original code supplied by the user.

However, the user's original code may have error-detection and error-handling features built in. If so, those features are carried over to the augmented code. If the user does not guard operations, those operations will not be guarded in the augmented code either. For example, the original code

```
IF (X .NE. 0.0) THEN
   Y = A / X
```

```
      ELSE
         CALL ERROR ('Please do not divide by zero.')
      END IF
```

produces the generated code

```
        if (x .ne. 0.0) then
C         y = a / x
          r$0 = a / x
          xbar = -r$0 / x
          do 99936 g$i$ = 1, g$p$
            g$y(g$i$) = xbar * g$x(g$i$)
99936     continue
          y = r$0
        else
          call g$error$0(g$p$, 'Please do not divide by zero.')
        endif
```

On the other hand, the original code

```
      Y = A / X
```

produces the generated code

```
C       y = a / x
        r$0 = a / x
        xbar = -r$0 / x
        do 99935 g$i$ = 1, g$p$
          g$y(g$i$) = xbar * g$x(g$i$)
99935   continue
        y = r$0
```

If the original code crashes, the augmented code crashes in exactly the same way. In either case, the original code and the augmented code behave the same way with respect to the erroneous condition `x = 0`.

The issue with respect to errors in the user's code is that the behavior of the original code is retained. In this class of error, we are not concerned that the evaluation of the derivative values may overflow. We are making no statements about what sort of test the user may determine to be appropriate. In particular, we are *not* objecting at this point in the discussion to the test for equality in this example. That objection comes later.

For the most part, the code generated by ADIFOR behaves exactly the same as the user's original code with respect to the values computed or with respect to errors that might occur. ADIFOR retains the parallelizability or vectorizability of the original code. It does not reorder statements. However, ADIFOR does assign some previously anonymous intermediate results to temporary storage locations. Assigning the results of intermediate computations may cause some compilers to compute answers that differ by one or two units in the last place. It is also possible that code relying on side effects and on a specific order of evaluation within an expression could produce a different value. Otherwise, the function values computed by ADIFOR's code agree with those computed by the user's original code.

6 Error Class 2: Differentiable Functions — Overflows

Many of the operators and intrinsic functions of Fortran (+, -, *, sin, cos, atan, sinh, cosh, exp) are everywhere differentiable. Some other operators and intrinsic functions (/, tan, log, log10, tanh) fail to be differentiable *only* at points where they fail to be defined. We refer to operators and functions in either set as *differentiable* because they are mathematically differentiable at each point in their domains.

If the user's original program evaluates differentiable functions without crashing, then we may also evaluate their derivatives. Hence, error handling for these operators and functions is completely within the domain of the user's original program, and we have no further error handling to do.

Properly speaking, the statement in the preceding paragraph is true *only if no overflow occurs.* For functions involving /, tan, log, log10, or tanh, the values of derivatives may be larger than the values of functions. Hence, the computations for derivatives may overflow, even when the evaluation of the function does not. If $f(t) = \ln(t)$ or $\tan(t)$, for example, then $|f'(t)| >> |f(t)|$ as $t \to 0^+$, or $\pi/2$, respectively.

6.1 Definition of an Error

An error belongs to this class if

- the function f is differentiable,
- the function f can be evaluated by the original user's program,
- the code generated by ADIFOR suffers overflow or underflow while computing the derivative, and hence
- the generated code crashes or computes the wrong values for the derivatives.

The "error" here is that finite-precision arithmetic cannot compute the value that is mathematically defined.

6.2 Detection of an Error

We settle for possibly detecting the occurrence of under/overflow in the derivative computations. It might be that the correct derivative values can be computed even in the presence of under/overflow, but we have no hope of recognizing that. Hence, we may "detect" under/overflow events that do not really belong to this class of error.

The detection of errors in this class requires careful definitions of the domains in which each operation and elementary function can be evaluated without under/overflow, and tests of each argument before each derivative is computed. Alternatively, on a machine supporting IEEE arithmetic, NaNs and INFINITYs generated during derivative computations signal overflow.

The cost of detecting this class of error in software is so high that we rely on the host computing environment.

6.3 Possible Actions

If we have some way of knowing that the generated code is running on a machine with IEEE arithmetic, we should take advantage of its capabilities.

As an example to illustrate the high cost of detecting and handling this class of error, consider the assignment statement

```
      Y = A / X + TAN (X) + LOG (X) + LOG10 (X) + TANH (X)
```

where `A` is a passive variable and `X` and `Y` are active. ADIFOR generates the code

```
C        y = a / x + tan(x) + log(x) + log10(x) + tanh(x)
         r$0 = a / x
         r$9 = cosh(x)
         xbar = 1.0 / (r$9 * r$9)
         xbar = xbar + 1.0 / (x * log(10.0))
```

```
        xbar = xbar + 1.0 / x
        r$10 = cos(x)
        xbar = xbar + 1.0 / (r$10 * r$10)
        xbar = xbar + (-r$0 / (x))
        do 99932 g$i$ = 1, g$p$
          g$y(g$i$) = xbar * g$x(g$i$)
99932   continue
        y = r$0 + tan(x) + log(x) + log10(x) + tanh(x)
```

which has many possible sources of overflow, depending on the value of `X`. We might define

```
      REAL LOGBIG, LOGSMALL, HALFBIG
      LOGBIG   = LOG (Max_Real) - epsilon
      LOGSMALL = LOG (Min_Positive_Real) + epsilon
               = - LOGBIG
      HALFBIG  = Max_Real / 2 - epsilon
```

and generate annotated code like this to detect and prevent overflow errors. The following code implements Option VII: Fully guard derivative computations against undefined derivatives and overflows. We give the generated code in full detail in order to communicate by example what code must be generated. We conclude that this option is too expensive at run time.

```
C       y = a / x + tan(x) + log(x) + log10(x) + tanh(x)
        r$0 = a / x
        r$9 = cosh(x)
        TMP = 2 * LOG (ABS (r$9))
        IF (TMP .GE. LOGBIG) THEN
           XBAR = G$ERROR (OVERFLOW_FLAG, 'tanh')
        ELSE IF (TMP .LE. LOGSMALL) THEN
           XBAR = G$ERROR (OVERFLOW_FLAG, 'tanh')
        ELSE
           xbar = 1.0 / (r$9 * r$9)
        END IF

        . . .

        do 99932 g$i$ = 1, g$p$
          TMP = LOG (ABS (XBAR)) + LOG (ABS (g$x(g$i$)))
          IF ((TMP .LE. LOGSMALL) .OR. (TMP .GE. LOGBIG)) THEN
             g$y(g$i$) = G$ERROR (OVERFLOW_FLAG, 'assignment')
          ELSE
             g$y(g$i$) = xbar * g$x(g$i$)
          END IF
99932   continue
        y = r$0 + tan(x) + log(x) + log10(x) + tanh(x)
```

It is more efficient in Fortran 77 to test the values rather than their logarithms. However, the in Fortran 90, it is as efficient to use the built-in functions to return the exponent of a number (to replace the calls to `LOG`) as it is to test the values themselves.

7 Error Class 3: Nondifferentiable Functions — Lim $f' = \pm\infty$

This section and the next form the core of this paper. Together, they describe how ADIFOR deals with Fortran intrinsic functions that are not globally differentiable.

Not all operators and intrinsic functions are differentiable. We call a function *nondifferentiable* if there are points *in its domain* for which its derivative does not exist. We are not concerned with points outside the domain of the functions because the augmented program will already behave the same way as the user's original program at such points. Our only concern is with points at which the function can be evaluated by using finite-precision arithmetic, but the derivative cannot.

There are only a few such points with which we must be concerned. The following table is an exhaustive listing.

Function	Points of nondifferentiability
sqrt (x)	$x = 0$
asin (x)	$x = \pm 1$
acos (x)	$x = \pm 1$
x**y	Depends on implementation
	$x \leq 0$
abs (x)	$x = 0$
sign $(x,\ y)$	$x = 0$, or $y = 0$
aint (x)	$x = \pm 1, \pm 2, \ldots$
max $(x,\ y)$	$x = y$
min $(x,\ y)$	$x = y$
dim $(x,\ y)$	$x = y$

We have divided these functions into two classes (by the horizontal line). For sqrt, asin, and acos, the derivatives approach $\pm\infty$ as $t \rightarrow$ point of nondifferentiability. These functions will be treated in this section. The power operator (**) is a special case. ADIFOR currently computes the derivative from t**u = exp $(u \ln (t))$. For a more sophisticated implementation, the power operator is not defined if $t \leq 0$ and u is fractional.

The second class of function (below the horizontal line) are nonsmooth as functions and will be treated in the next section.

7.1 Definition of an Error

An error belongs to this class if

- the function f is differentiable,
- the function f can be evaluated by the original user's program, and
- the function involves elementary functions sqrt, asin, acos, and **, evaluated at (or near) the point of nondifferentiability.

An error can belong to this class even when the mathematical function is well behaved, but intermediate results produce this error. For example (from H. Fischer [4]), let

$$f(x,z) := \sqrt{x^4 + z^4},$$

where x and z are both active variables. The function f is differentiable at the point $(x,z) = (0,0)$. However, a step-wise evaluation of f' forms $u(x,z) = x^4 + z^4$ first, then forms $f = \sqrt{u}$. Since $u(0,0) = 0$, the derivative of f is undefined and produces an error in this class when evaluating $f = \sqrt{u}$. To avoid this problem, one would have to examine the interaction between successive applications of elementary functions and operations. Such "symbolic" analysis is beyond the scope of ADIFOR and of automatic differentiation in general.

7.2 Possible Actions

For each function in this class, we generate augmented code of the form

```
r$0 = funct (t)
if (t is not near  BAD_POINTS) then
   g$r$0 = result of derivative calculations
else
   g$r$0 = g$error (1, 'Derivative of funct does not exist')
end if
y = r$0
```

The function **g$error** is described in Section 4.

For the reverse mode accumulation of adjoints within expressions, this class of error is handled in exactly the same manner as the previous class.

In the rest of this section, we discuss each of the functions sqrt, asin, acos, and **. Whenever a call to one of the first three functions appears in the original code, ADIFOR generates a call to a function **g$sqrt**, **g$asin**, or **g$acos**; ** is handled specially by in-line code as described in Section 7.5. Elementary functions described in this section that require error handling are treated by ADIFOR in the same manner as elementary functions like sin and cos which do not require error handling. The codes for each function are in Appendix A.

7.3 Sqrt

The function **g$sqrt** returns the derivative value for SQRT: $d(\sqrt{x})/dx$. The function **g$sqrt** is used like this in the code generated by ADIFOR:

```
C     Z = SQRT (X)
      r$1 = sqrt (x)
      temp = g$sqrt (x, r$1)
      do 99990 g$i$ = 1, g$p$
         g$z(g$i$) = temp * g$x(g$i$)
99990 continue
```

```
or
```

```
      xbar = xbar + zbar * g$sqrt (x, r$1)
```

Let $y := \sqrt{|x|}$. Then, the value returned by **g$sqrt** is

$$\mathtt{g\$sqrt}(x) := \begin{cases} 1/(2y) & \text{for } x > 0, \\ \text{InfVal from } \mathtt{g\$error} & \text{for } x = 0, \text{ and} \\ -1/(2y) & \text{for } x < 0. \end{cases}$$

Rationale: At the point of nondifferentiability $x = 0$, the default for InfVal = 0 is a generalized gradient value if we assume that SQRT (x) := SQRT (ABS (x)). Further, it makes expressions like SQRT (X*X*X*X + Y*Y*Y*Y) have the correct derivative. However, it violates the principle of continuity of catastrophe. Alternatively, the value of InfVal = INFINITY makes the one-sided limit correct.

Denormalized: If x is a denormalized number, then y is well into the range of normalized numbers. Hence, $1/(2y)$ cannot overflow. The computed value of y is zero if and only if x is zero.

7.4 Asin and acos

The functions **g$asin** and **g$acos** return the derivative values for ASIN: $d(\text{asin}(x))/dx$ and for ACOS: $d(\text{acos}(x))/dx$, respectively. The functions **g$asin** and **g$acos** are used in the code generated by ADIFOR in the same manner as **g$sqrt**.

The values returned by **g$asin** and **g$acos** are

$$\texttt{g\$asin}(x) := \begin{cases} 1/\sqrt{1-x^2} & \text{for } |x| < 1, \\ \text{InfVal from } \texttt{g\$error} & \text{for } |x| = 1, \text{ and} \\ \text{InfVal from } \texttt{g\$error} & \text{for } |x| > 1. \end{cases}$$

$$\texttt{g\$acos}(x) := \begin{cases} -1/\sqrt{1-x^2} & \text{for } |x| < 1, \\ \text{InfVal from } \texttt{g\$error} & \text{for } |x| = 1, \text{ and} \\ \text{InfVal from } \texttt{g\$error} & \text{for } |x| > 1. \end{cases}$$

Rationale: At the points of nondifferentiability $x = \pm 1$, the default for InfVal = 0 indicates an extreme point. However, it violates the principle of continuity of catastrophe. Alternatively, the value of InfVal = INFINITY makes the one-sided limit correct. If $|x| > 1$, usually the user's original code will have already crashed while evaluating ASIN (x). If it has continued execution (perhaps with value NaN), we should continue execution also. No value is reasonable since the function is not defined, so we choose to return the same value at at $x = \pm 1$. Alternatively, we could return whatever was assigned to the value of ASIN (x).
Denormalized: No matter how close x is to ± 1, neither $1 - x$ nor $1 + x$ can be very small relative to the machine epsilon. Hence, the derivative evaluated at a machine-representable number cannot overflow.

7.5 Power: **

The power operator is treated differently from the other elementary functions. For $x^y = x$**y, we must be prepared to handle separate cases for either or both x and y being active variables. We must be able to compute derivatives with respect to either or both of them.

Following the general philosophy of the other exception-handling routines, we try to catch the situations where the function value itself is at least mathematically defined, but the derivatives are not. In contrast to the other intrinsic functions, we prefer to generate the necessary code in-line, except for a call to **g$error** in the exceptional cases. "In-line" here means that we branch directly on the bar quantity assignments.

Depending on whether we wish to differentiate x^y with respect to x or with respect to y, we should consider it as a "power" or as an "exponential." Correspondingly, the error classification number should be 5 or 10, respectively, in the call to **g$error**. When both x and y are active reals, we consider x^y simultaneously as a power and as an exponential.

We assume that x^y has a well-defined value if $x \geq 0$ or y is an integer with $0^0 = 1$. On particular systems, the values may be defined differently if $x = 0$ or $y = 0$, and there will be overflow when $y < 1$ and x is sufficiently small. We will do nothing about this because we would otherwise also have to safeguard simple divisions.

For fixed y, the derivative of x^y with respect to x is mathematically defined except when $x = 0$, and $0 < y < 1$. This case is a generalization of the square root situation. Therefore, we set the derivative to InfVal. When $y = 0$, we set the derivative with respect to x to zero and do not call **g$error**, even if $x = 0$.

For fixed x, the derivative of x^y with respect to y is mathematically defined except when $x \leq 0$, and the value of y is an integer. When x is negative, x^y is not defined for any fractional y. Therefore, we set the value of the derivative to NoLmVl at integral values of y. When $x = 0$, the derivative is zero for all $y > 0$. For $y = 0$, we may again use NoLmVl.

Note that **ybar** remains unchanged if $x = 0$ and $y > 0$.

When y is not active, there is no **ybar**, and the second part of the calculation can be omitted. If y is of type integer, the first part can be reduced to the single statement

```
      if (x .ne. 0.0) xbar = xbar + y * zbar * r$0 / x
```

If x is passive, there is no **xbar**, and the first part can be omitted. Finally, if either a or y are constants that can be evaluated at compile time, further simplifications are possible.

ADIFOR should generate code like this:

```
C     z = x**y
      r$0 = x**y
      zbar = 1.0
      xbar = 0.0
      ybar = 0.0
C
C     First, do the derivative with respect to x.
C
      if (x .ne. 0.0) then
         xbar = xbar + y * zbar * r$0/x
      else
         if ((y .gt. 0.0) .and. (y .lt. 1.0)) then
            xbar = xbar + zbar
     +              * g$error (5, 'Fractional power of zero')
         end if
      end if

C     Second, do the derivative with respect to y.
      if (x.gt.0.0) then
         ybar = ybar + zbar * r$0*log(x)
      else
         if ((x .lt. 0.0) .or. (y .eq. 0.0)) then
            ybar = ybar + zbar
     +              * g$error (10, 'Negative basis or 0**0')
         end if
      end if
      z = r$0
      do 99990 g$i$ = 1, g$p$
        g$z(g$i) = xbar*g$x(g$i) + ybar*g$y(g$i)
99990 continue
```

8 Error Class 4: Nonsmooth Functions — Lim f' does not exist

The functions abs (x), sign (x, y), aint (x), max (x, y), min (x, y), and dim (x, y) are not smooth. Although the user's original program can evaluate the functions, they are not differentiable at certain points. As for functions whose derivatives have infinite limits, we have the same alternatives as before, but now it is less clear what value should be returned.

8.1 Definition of an Error

It is an error in this class to evaluate one of the functions abs (x), sign (x, y), aint (x), max (x, y), min (x, y), or dim (x, y) at a point of nondifferentiability.

8.2 Possible Actions

In many calculations, variable vectors are scaled by their L_1 norm or L_∞ norm (i.e., the sum or maximum of the component moduli). Later on, this scaling is undone so that the overall calculation is mathematically smooth, even when some of the components are zero or their absolute values are tied at the maximum. Then the automatic differentiation should go through and yield the right results.

For the rest of this section, we give the code to be generated for each of the functions in this class.

8.3 Abs

The function **g$abs** returns the derivative value for ABS: $d(|x|)/dx$. The function **g$abs** is used like this in the code generated by ADIFOR:

```
C     Z = ABS (X)
      r$1 = abs (x)
      temp = g$abs (x, r$1)
      do 99990 g$i$ = 1, g$p$
         g$z(g$i$) = temp * g$x(g$i$)
99990 continue

or

      xbar = xbar + zbar * g$abs (x, r$1)
```

Then, the value returned by **g$abs** is

$$\texttt{g\$abs}(x) := \begin{cases} -1 & \text{for } x < 0, \\ \text{NoLmVl from } \texttt{g\$error} & \text{for } x = 0, \text{ and} \\ 1 & \text{for } x > 0. \end{cases}$$

Rationale: At the point of nondifferentiability $x = 0$, the default for NoLmVl $= 0$ is a generalized gradient value that indicates an extreme point.

8.4 Sign

The function **g$sign** returns the derivative value for SIGN: d(SIGN $(x, y)/dx$. It is not necessary to differentiate with respect to y because d(SIGN $(x, y)/dy = 0$. Fortran's SIGN (x, y) $:= |x| \cdot$ signum (y). The function **g$sign** is used by ADIFOR in the same way as **g$abs**. The value returned by **g$sign** depends on the signs of both x and y:

x/y	$-$	0	$+$
$-$	1	NoLmVl	-1
0	NoLmVl	NoLmVl	NoLmVl
$+$	-1	NoLmVl	1

Rationale: At the point of nondifferentiability $x = 0$, the default for NoLmVl $= 0$ is a generalized gradient value equal to the average of the two limits from each side.

8.5 Aint and anint

The functions **g$aint** and **g$anint** return the derivative values for AINT and ANINT, respectively. Fortran's AINT (x) truncates toward 0, so it is not differentiable at $x = \pm 1, \pm 2, \ldots$. ANINT rounds to the nearest integer, so it is not differentiable at $x =$ odd multiples of $1/2$. The functions **g$aint** and **g$anint** return InfVal at the points of nondifferentiability, and 0 elsewhere. An alternative choice is NoLmVl $= 0$, the limit from each side.

8.6 Mod

The function **g$mod** returns the derivative value for MOD. Fortran's MOD (x, y) $= x -$ aint $(x/y) \cdot y$, so it is not differentiable at $x =$ multiples of y. The function **g$mod** returns InfVal at the points of nondifferentiability, and 1 elsewhere. An alternative choice is NoLmVl $= 0$ to signal an extreme value, or 1, the limit from both sides.

8.7 Dim

The function **g$dim** returns the derivative value for DIM: $d(\text{DIM }(x, y))/dx$. It is sufficient to compute the derivative with respect to x because $d(\text{DIM }(x, y))/dx = d(\text{DIM }(x, y))/dy$. Fortran's DIM $(x,\ y) = \max\ (x - y,\ 0)$, so it is not differentiable at $x = y$. The function **g$dim** returns

$$\texttt{g\$dim}(x, y) := \begin{cases} 1 & \text{for } y < x, \\ \text{NoLmVl from } \texttt{g\$error} & \text{for } y = x, \text{ and} \\ 0 & \text{for } y > x. \end{cases}$$

Rationale: At the points of nondifferentiability $x = y$, the default for NoLmVl = 0 is a generalized gradient. An alternative choice is 1/2, the average of the limits from both sides.

8.8 Max and Min

The functions **g$max** and **g$min** return the derivative values for MAX and MIN, respectively. MAX and MIN are not necessarily differentiable at points where their arguments are equal. In that case, we return the average of the two derivatives. At the points of nondifferentiability, the default for TieVal = 1/2 is a generalized gradient. It is sufficient to compute $d(\text{MAX }(x, y))/dx$ because $d(\text{MAX }(x, y))/dy = 1 - d(MAX(x, y))/dx$, and similarly for MIN. Hence, the values returned by the functions **g$max** and **g$min** are

$$\texttt{g\$max}(x, y) := \begin{cases} 1 & \text{for } x > y, \\ \text{TieVal from } \texttt{g\$error} & \text{for } x = y, \text{ and} \\ 0 & \text{for } x < y. \end{cases}$$

$$\texttt{g\$min}(x, y) := \begin{cases} 1 & \text{for } x < y, \\ \text{TieVal from } \texttt{g\$error} & \text{for } x = y, \text{ and} \\ 0 & \text{for } x > y. \end{cases}$$

Fortran's MAX and MIN functions accept more than two arguments. If ADIFOR encounters such calls, it translates them into a sequence of binary calls to MAX or MIN and applies the exception handling described here to each binary call. This procedure has the unfortunate consequence that if many arguments are equal, their slopes are weighted 1/2, 1/4, 1/8, ... Hence, we are considering more sophisticated ways to handle MAX and MIN.

In some applications, especially to univariate functions, one might prefer a different formulation. If $f := \max(x, y)$, we might define

$$f' := \begin{cases} x' & \text{for } x > y, \\ y' & \text{for } x < y, \\ (x' + y')/2 & \text{if } x = y \text{ and } x' \times y' > 0 \\ 0 & \text{if } x = y \text{ and } x' \times y' \le 0. \end{cases}$$

This definition is attractive because the derivative value of zero is taken by many optimization codes as a signal for a local optimum. This definition has the disadvantage, however, that its value is not always appropriate in the context of multidimensional optimization. If this interpretation of the derivatives of max and min is desired, calls to MAX and MIN can be replaced by calls to **my_max** and **my_min**, respectively, and the user can supply subroutines **g$my_max** and **g$my_min** which implement the alternative definition.

9 Future Directions

So far in this paper, we have described the ADIFOR exception-handling mechanism. We are continuing to work on several related issues which we outline briefly in this section.

9.1 Quantifying Distance to Danger

In Section 3, we listed options based in relative safety or trust region approaches to indicate when the function is being evaluated at or near a point of nondifferentiability. The relative safety measure is inexpensive, but hard to interpret. The trust region is easy to interpret in terms of the original independent variables, but it is expensive to compute. We are exploring the nature of that cost and ways to economize by combining the relative safety measure with the trust region approach. The results will appear in a later paper.

9.2 Branching

The user's original program may contain IF statements which have the effect of defining functions that are not differentiable or are not even continuous. The augmented code executes the appropriate branches in the manner described by Kedem [5]. However, the value of the derivatives computed at points at which equality holds are suspect. The derivatives computed are those that would result from taking limits of points for which inequality holds. The result may appear to be a derivative value at a point for which the mathematical derivative does not exist. The following example of possible user's code to compute the absolute value illustrates some of the dangers.

Suppose that the user's original code includes the following code to compute an absolute value:

```
if (t .ge. 0.0) then
   abs = t
else
   abs = -t
end if
```

Then, we would conclude that

$$\mathrm{abs}'(t) = \begin{cases} 1 & \text{if } t \geq 0 \\ -1 & \text{if } t < 0. \end{cases}$$

An equally reasonable programmer might write

```
if (t .gt. 0.0) then
   abs = t
else
   abs = -t
end if
```

or

```
if (t .gt. 0.0) then
   abs = t
else if (t .lt. 0.0) then
   abs = -t
else
   abs = 0.0
end if
```

The three different equivalent programs for abs give values for the derivative at 0 as $+1$, -1, and 0, respectively.

Another unintended consequence of an IF statement is illustrated by another example from Fischer [4]. If the function $f(x) := x^2$ is programmed as

```
if (x .eq. 1.0) then
   f = 1.0
else
   f = x * x
end if
```

then automatic differentiation of this program would incorrectly compute $f'(1) = 0$. While few programmers would implement x^2 in this manner, we have encountered similar formulations in production codes. It is a reasonable way to program when function values are known explicitly for special points and evaluation of the formula is expensive.

The "fault" in the abs and x^2 examples is not with automatic differentiation; the results are unavoidable consequences of the style of the original program supplied by the user. ADIFOR currently handles programs with IF statements. The flow of control in the derivative code is the same as the flow of control in the original code. If tests do not occur at equality, the point of evaluation is interior to the domain of differentiability, and ADIFOR computes the correct derivative values. If tests at equality *do* occur, the results are usually appropriate for some one-sided limit, but they can be incorrect.

The user of ADIFOR should be aware of the possibility that IF statements can be used to compute derivative values for nondifferentiable functions. Relative safety or trust region techniques will allow us to alert the user to danger.

9.3 Mathematical Pitfalls

Automatic programming is no substitute for mathematical insight. Automatic differentiation is no exception. The following examples are from Fischer [4].

Example 1: Let $f_2(x) := x \cdot \exp(-x^2)$ and $f_k(x) = f_{k-1}(x) \cdot \exp(f_2(x))$. For $x = 1$, $\lim_{k \to \infty} f_k'(0) = 1$, while $f'(0) = 0$.

Example 2: Let $f_n(x) := \frac{1}{2^n \pi} \cdot \sin(2^n \pi x)$, for $n = 1, 2, \ldots$. Then the sequence of function values $\{f_n\}$ converges everywhere pointwise to $f \equiv 0$, but the sequence of derivative values $\{f_n'\}$ converges to 1 for infinitely many values of x.

The "problem" here is that differentiation and limits are not interchangeable. The use of ADIFOR to perform the differentiation does not change that mathematical fact. Automatic differentiation correctly computes the requested derivative, but it remains the responsibility of the user to interpret the derivative correctly.

Acknowledgments

We thank Alan Carle and Paul Hovland for their helpful suggestions on exception handling and for their essential roles in the ADIFOR development project.

Appendix A. Exception-Handing Code

In Section 4, we described the functionality of the error-handling module. Here we give the Fortran code for a simple implementation.

```
C Purpose:  Simple error handler for ADIFOR
C Authors:  George F. Corliss and Andreas Griewank

C Description:
C    Initialize error handler (optional).
C    When an error occurs:
C       Conditionally print an error message
C       Conditionally STOP execution,
C          or else return a value to let execution continue.
C    Report the number of errors in each class (optional).
C
C    The user can control:
C       ClassN  Class of error to which the error belongs
C       MsgTxt  Content of the message text
C       PrintF  Printing of the message
C       ErrFil  Logical unit number to which the error messages
C               should be printed.
C       HaltFg  Halting execution
C       InfVal  Value returned if derivative limit is infinite
C       NoLmVl  Value returned if derivative has no limit
C       TieVal  Partial derivative of Max (x, y) or Min (x, y)
C               with respect to x when x = y
C       g$error Source code can be customized, if necessary
C
C Contents:
C     block data
C     function g$error (ClassN, MsgTxt)
C     subroutine g$InitEr (PFlag, HFlag, ErrF, Infin, NoLim, MaxV)
C     subroutine g$ReptEr (LUnitN)
C     real function g$sqrt (x, y)
C     real function g$asin (x)
C     real function g$acos (x)
C     real function g$aint (x, y)
C     real function g$anint (x, y)
C     real function g$mod (x, y, z)
C     real function g$abs (x)
C     real function g$sign (x, y)
C     real function g$dim (x, y)
C     real function g$max (x, y)
C     real function g$min (x, y)
C
C Usage:
C     Initialize error handler
C     Optional.  These happen to be the default parameters.
C     call g$InitEr (.True., .False., 6, 0.0, 0.0, 0.5)
C     . . .
C     if (All is fine) then
C        Answer = Normal processing
C     else
C        Answer = g$error (1, 'Something is wrong!')
C     end if
```

```
C     . . .
C     Optional.  If you want to know how many errors of each type.
C     call g$ReptEr (6)
C Reference:
C     Christian Bischof, Alan Carle, George Corliss, Andreas Griewank,
C     Paul Hovand, Generating Derivative Codes from Fortran Programs,
C     Preprint No. MCS--P263--0991, Mathematics and Computer Science
C     Division, Argonne National Laboratory, 1991.  Also appeared as
C     Technical Report No. 91185, Center for Research in Parallel
C     Computation, Rice University, Houston, TX., 1991.


      block data

C Purpose:  Initialize default values for error handling.
C        PrintF  Print Flag
C                   0  No printing
C                   1  Print error message
C               Default: Print
C        HaltFg  Halt Flag
C                   0  Continue execution
C                   1  Halt execution
C               Default: Continue
C        ErrFil  Error File
C                   Logical unit number to which error messages
C                   (if any) should be written.
C               Default: Standard output
C        InfVal  Infinite Value
C                   In case HaltFg = 0 so that execution continues,
C                   this is the value returned in cases like sqrt (0)
C                   for which the function is defined, but the
C                   derivative has an infinite limit.  On a
C                   machine with IEEE arithmetic, INFINITY or NaN
C                   would be good choices.
C               Default: 0.0
C        NoLmVl  No Limit Value
C                   In case HaltFg = 0 so that execution continues,
C                   this is the value returned in cases like abs (0)
C                   for which the function is defined, but the
C                   derivative has no limit.
C               Default: 0.0
C        TieVal  Value returned for Max (x, x) or Min (x, x)
C                   In case HaltFg = 0 so that execution continues,
C                   this is the value returned for Max (x, x) or
C                   Min (x, x).  The average is a generalized gradient.
C               Default: 0.5

      common / g$ErrBlk / PrintF, HaltFg, ErrFil, InfVal, NoLmVl,
     +                    TieVal, KtErr
      logical PrintF, HaltFg
      integer ErrFil, KtErr(20)
      real    InfVal, NoLmVl, TieVal
      data PrintF / 1 /,
     +     HaltFg / 0 /,
     +     ErrFil / 6 /,
     +     InfVal / 0.0 /,
     +     NoLmVl / 0.0 /,
     +     TieVal / 0.5 /,
     +     KtErr  / 20 * 0 /
      end


      function g$error (ClassN, MsgTxt)

C Purpose:  Conditionally print message, conditionally STOP.
C Input parameters:
C    ClassN   Class Number
C               Derivative limit is infinite
C                  1  sqrt (0)
C                  2  asin (+-1), acos (+-1)
C                  3  Fractional power of zero
C                  4  Negative basis or 0**0
C                  5  aint (integer)
C                  6  mod (n*y, y)
C               Derivative limit does not exist
C                 11  abs (0)
C                 12  sign (0, x) or sign (x, 0)
C                 13  dim (x, x)
C               15    Max, Min
C               20    Values hit equality in IF tests
C    MsgTxt   Message Text

      integer      ClassN
      character*40 MsgTxt
      common / g$ErrBlk / PrintF, HaltFg, ErrFil, InfVal, NoLmVl,
     +                   TieVal, KtErr
      logical PrintF, HaltFg
      integer ErrFil, KtErr(20)
      real    InfVal, NoLmVl, TieVal

C Increment error counter.
      KtErr(ClassN) = KtErr(ClassN) + 1
C Conditionally print an error message.
      if (PrintF) then
         write (ErrFil, 1010) MsgTxt
 1010    format (/ 'ERROR: ', A40 /)
      end if
C Conditionally halt execution.
      if (HaltFg) then
C!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!     S T O P

        STOP

C!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!     S T O P
      end if
C What value should we return?
      if (ClassN .le. 9) then
         g$error = InfVal
      else if (ClassN .le. 14) then
         g$error = NoLmVl
      else
         g$error = TieVal
      end if
      return
      end


      subroutine g$InitEr (PFlag, HFlag, ErrF, Infin, NoLim, MaxV)

C Purpose:  Optionally called to override error handler defaults
C Input parameters:
C     PFlag   Print Flag
C     HFlag   Halt Flag
C     Err_F   Error File
C     Infin   Infinite Value
C     No_Lim  No Limit Value
C     TieVal  Maximum/Minimum value

      logical PFlag, HFlag
      integer ErrF
      real Infin, NoLim, MaxV
      common / g$ErrBlk / PrintF, HaltFg, ErrFil, InfVal, NoLmVl,
     +                    TieVal, KtErr
      logical PrintF, HaltFg
      integer ErrFil, KtErr(20)
      real    InfVal, NoLmVl, TieVal

      PrintF = PFlag
      HaltFg = HFlag
      ErrFil = ErrF
      InfVal = Infin
      NoLmVl = NoLim
      TieVal = MaxV
      do 10 i = 1, 20
         KtErr(i) = 0
 10   continue
      return
      end


      subroutine g$ReptEr (LUnitN)

C Purpose:  Optionally report the cumulative number of errors
C           of each class.
C Input parameter:
C     LUnitN  Logical Unit Number to which the report should be written.
C Output:   Report on LUnitN

      integer LUnitN
      common / g$ErrBlk / PrintF, HaltFg, ErrFil, InfVal, NoLmVl,
     +                    TieVal, KtErr
      logical PrintF, HaltFg
      integer ErrFil, KtErr(20)
      real    InfVal, NoLmVl, TieVal
      write (LUnitN, 1010) KtErr(1), KtErr(2), KtErr(3), KtErr(4),
     +    KtErr(5), KtErr(6), KtErr(11), KtErr(12),  KtErr(13),
     +    KtErr(15)
 1010 format (// 'How many errors did the ADIFOR-generated code ',
     +           'detect?',
     +       /  'Error: Function is defined,',
     +       /  '       but not differentiable in exact arithmetic.'
     +       // '  Class Number                Number of errors',
     +        / '    Derivative limit is infinite:    ',
     +        / '       1  sqrt (0)                   ', i4,
     +        / '       2  asin (+-1), acos (+-1)     ', i4,
     +        / '       3  Fractional power of zero   ', i4,
     +        / '       4  Negative basis or 0**0     ', i4,
     +        / '       5  aint (integer)             ', i4,
     +        / '       6  mod (n*y, y)               ', i4,
     +        / '    Derivative limit does not exist: ',
     +        / '      11  abs (0)                    ', i4,
     +        / '      12  sign (0, x) or sign (x, 0) ', i4,
     +        / '      13  dim (x, x)                 ', i4,
     +        / '    15 Max (x, x) or Min (x, x)      ', i4 //)
      return
      end


      real function g$sqrt (x, y)

C Purpose:  Return the derivative value for SQRT.
C          d (SQRT (x)) / d x.
C Input parameter:
C     x   Argument to SQRT
C     y   Result of SQRT (x)
C         Instead, we could compute y = SQRT (x) locally.
C Returned value:
C     if x < 0   g$sqrt = -1 / (2 y)
C                Usually will have previously crashed while
C                evaluating SQRT (x).  If it did not crash,
C                we assume SQRT (x) := SQRT (ABS (x)).
C     if x = 0   g$sqrt = InfVal from g$error
C     if x > 0   g$sqrt =  1 / (2 * y)
C Rationale:  At the point of nondifferentiability, the default
C     for InfVal = 0 is a generalized gradient value if we assume
C     SQRT (x) := SQRT (ABS (x)).  Further, it makes expressions
C     like SQRT (X*X*X*X + Y*Y*Y*Y) have the correct derivative.
C     However, it violates the principle of continuity of catastrophe.
C     Alternatively, the value of InfVal = INFINITY makes the
C     one-sided limit correct.
C Denormalized:  If x is a denormalized number, then y is well into
C     the range of normalized numbers.  Hence, 1 / (2 y) cannot
C     overflow.  The value of y is zero if and only if x is zero.
C Usage in ADIFOR-generated code:
C C     Z = SQRT (X)
C       r$1 = sqrt (x)
C       temp = g$sqrt (x, r$1)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$)
```

```
C  99990 continue
C or
C       xbar = xbar + zbar * g$sqrt (x, r$1)

      real x, y

      if (x .gt. 0.0) then
         g$sqrt =  1.0 / (2.0 * y)
      else if (x .lt. 0.0) then
         g$sqrt = -1.0 / (2.0 * y)
      else
         g$sqrt = g$error (1,
     +            'Computed the derivative of SQRT (0)')
      end if
      return
      end


      real function g$asin (x)

C Purpose:  Return the derivative value for ASIN.
C          d (ASIN (x)) / d x.
C Input parameter:
C     x   Argument to ASIN
C Returned value:
C     if |x| < 1  g$asin = 1 / sqrt (1 - x*x)
C     if |x| = 1  g$asin = InfVal from g$error
C     if |x| > 1  g$asin = InfVal from g$error
C                 Usually will have previously crashed while
C                 evaluating ASIN (x).
C Rationale:  At the points of nondifferentiability, the default
C     for InfVal = 0 indicates an extreme point.
C     However, it violates the principle of continuity of catastrophe.
C     Alternatively, the value of InfVal = INFINITY makes the
C     one-sided limit correct.
C Denormalized:  No matter how close x is to +- 1, neither 1 - x
C     nor 1 + x can be very small.  Hence, the derivative evaluated
C     at a machine-representable number cannot overflow.
C Usage in ADIFOR-generated code:
C C     Z = ASIN (X)
C       r$1 = asin (x)
C       temp = g$asin (x)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$)
C  99990 continue
C or
C       xbar = xbar + zbar * g$asin (x)

      real x

      if (abs (x) .lt. 1.0) then
         g$asin =  1.0 / sqrt ((1.0 - x) * (1.0 + x))
      else
         g$asin = g$error (2,
     +            'Computed the derivative of ASIN (1)')
```

```
      end if
      return
      end


      real function g$acos (x)

C Purpose:  Return the derivative value for ACOS.
C          d (ACOS (x)) / d x.
C Input parameter:
C     x   Argument to ACOS
C Returned value:
C     if |x| < 1  g$acos = -1 / sqrt (1 - x*x)
C     if |x| = 1  g$acos = InfVal from g$error
C     if |x| > 1  g$acos = InfVal from g$error
C                 Usually will have previously crashed while
C                 evaluating ACOS (x).
C Rationale:  At the points of nondifferentiability, the default
C     for InfVal = 0 indicates an extreme point.
C     However, it violates the principle of continuity of catastrophe.
C     Alternatively, the value of InfVal = INFINITY makes the
C     one-sided limit correct.
C Denormalized:  No matter how close x is to +- 1, neither 1 - x
C     nor 1 + x can be very small.  Hence, the derivative evaluated
C     at a machine-representable number cannot overflow.
C Usage in ADIFOR-generated code:
C C     Z = ACOS (X)
C       r$1 = acos (x)
C       temp = g$acos (x)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$)
C  99990 continue
C or
C       xbar = xbar + zbar * g$acos (x)

      real x

      if (abs (x) .lt. 1.0) then
         g$acos = -1.0 / sqrt ((1.0 - x) * (1.0 + x))
      else
         g$acos = g$error (2,
     +             'Computed the derivative of ACOS (1)')
      end if
      return
      end


      real function g$abs (x)

C Purpose:  Return the derivative value for ABS.
C          d (ABS (x)) / d x.
C Input parameter:
C     x   Argument to ABS
C Returned value:
C     if x < 0   g$abs = -1
```

```
C     if x = 0   g$abs = NoLmVl from g$error
C     if x > 0   g$abs =  1
C Rationale:  At the point of nondifferentiability, the default
C     for NoLmVl = 0 is a generalized gradient value equal to the
C     average of the two limits from each side.  It satisfies the
C     principle of continuity of catastrophe, and indicates an
C     extreme point.
C Usage in ADIFOR-generated code:
C       Z = ABS (X)
C       r$1 = abs (x)
C       temp = g$abs (x)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$)
C  99990 continue
C or
C       xbar = xbar + zbar * g$abs (x)

      real x

      if (x .gt. 0.0) then
         g$abs =  1.0
      else if (x .lt. 0.0) then
         g$abs = -1.0
      else
         g$abs = g$error (11,
     +           'Computed the derivative of ABS (0)')
      end if
      return
      end


      real function g$sign (x, y)

C Purpose:  Return the derivative value for SIGN (x, y).
C           d (SIGN (x, y)) / d x.
C           d (SIGN (x, y)) / d y = 0.
C Input parameter:
C     x, y   Arguments to SIGN
C
C     SIGN (x, y) := ABS (x) * signum (y).  sign (0) := +1.
C
C        y            -              0             +
C       x
C       -          -x * (-1)      -x * 1        -x * 1
C                   x'           -x' --> 0        -x'
C       0           0 * (-1)       0 * 1         0 * 1
C                   NE --> 0     NE --> 0       NE --> 0
C       +           x * (-1)       x * 1         x * 1
C                    -x'          x' --> 0        x'
C
C Returned value:
C    g$sign =
C       x \  y     -        0       +
C       -          1     NoLmVl    -1
C       0       NoLmVl  NoLmVl  NoLmVl
```

```
C       +         -1     NoLmVl     1
C
C Rationale:  At the point of nondifferentiability, the default
C     for NoLmVl = 0 is a generalized gradient value equal to the
C     average of the two limits from each side.
C Usage in ADIFOR-generated code:
C       Z = SIGN (X, Y)
C       r$1 = sign (x, y)
C       temp = g$sign (x, y)
C       do 99990 g$i$ = 1, g$p$
C          g$x(g$i$) = temp * g$x(g$i$)
C  99990 continue
C or
C       xbar = xbar + zbar * g$sign (x, y)

      real x, y

      if ((x .eq. 0.0) .or. (y .eq. 0.0)) then
         g$sign = g$error (12,
     +            'Computed the derivative of SIGN (0, or 0)')
      else if (x .gt. 0.0) then
         if (y .gt. 0.0) then
            g$sign =  1.0
         else
            g$sign = -1.0
         end if
      else
         if (y .gt. 0.0) then
            g$sign = -1.0
         else
            g$sign =  1.0
         end if
      end if
      return
      end


      real function g$aint (x, y)

C Purpose:  Return the derivative value for AINT.
C          d (AINT (x)) / d x.
C Input parameter:
C     x   Argument to AINT
C     y   Result of AINT (x)
C         Instead, we could compute y = AINT (x) locally.
C Returned value:
C     AINT truncates toward zero.
C     if x  = +-1, +-2, ... g$aint = InfVal from g$error
C     otherwise  g$aint =  0
C Rationale:  At the points of nondifferentiability, the limit
C     of the derivative is infinite.  An alternative choice is
C     NoLmVl = 0, the limit from each side.
C Usage in ADIFOR-generated code:
C       Z = AINT (X)
C       r$1 = aint (x)
```

```
C       temp = g$aint (x, r$1)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$)
C  99990 continue
C or
C       xbar = xbar + zbar * g$aint (x, r$1)

      real x, y

      if ((y .eq. 0) .or. (y .ne. x)) then
         g$aint =  0.0
      else
         g$aint = g$error (5,
     +            'Computed the derivative of AINT (integer)')
      end if
      return
      end


      real function g$anint (x, y)

C Purpose:  Return the derivative value for ANINT.
C           d (ANINT (x)) / d x.
C Input parameter:
C     x   Argument to ANINT
C     y   Result of ANINT (x)
C         Instead, we could compute y = ANINT (x) locally.
C Returned value:
C     ANINT rounds to the nearest whole number.
C     if x  = +-1/2, +-3/2, ... g$anint = InfVal from g$error
C     otherwise  g$anint =  0
C Rationale:  At the points of nondifferentiability, the limit
C     of the derivative is infinite.  An alternative choice is
C     NoLmVl = 0, the limit from each side.  Determination of
C     the points of nondifferentiability is problematic.
C Usage in ADIFOR-generated code:
C       Z = ANINT (X)
C       r$1 = anint (x)
C       temp = g$anint (x, r$1)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$)
C  99990 continue
C or
C       xbar = xbar + zbar * g$anint (x, r$1)

      real x, y

      if ((x .eq. y - 0.5) .or. (x .eq. y + 0.5)) then
         g$anint = g$error (5,
     +             'Computed the derivative of ANINT (integer)')
      else
         g$anint =  0.0
      end if
      return
      end
      real function g$mod (x, y, z)

C Purpose:  Return the derivative value for MOD.
C           d (MOD (x, y)) / d x.
C           d (MOD (x, y)) / d y = 0.
C Input parameters:
C     x, y   Arguments to MOD
C     z   Result of MOD (x, y)
C         Instead, we could compute z = MOD (x, y) locally.
C Returned value:
C     MOD (x, y) := x - int (x / y) * y
C     if x = n * y  g$mod = - InfVal from g$error
C     otherwise  g$mod = 1
C Rationale:  At the points of nondifferentiability, the limit of
C     the derivative is - infinity.  Alternate choices would be
C     NoLmVl = 0 to signal an extreme value, or 1, the limit from
C     both sides.
C Usage in ADIFOR-generated code:
C       Z = MOD (X, Y)
C       r$1 = mod (x, y)
C       temp = g$mod (x, y, r$1)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$) - temp * g$y(g$i$)
C  99990 continue
C or
C       temp = g$mod (x, y, r$1)
C       xbar = xbar + zbar * temp
C       ybar = ybar - zbar * temp

      real x, y, z

      if (z .eq. 0) then
         g$mod = - g$error (6,
     +             'Computed the derivative of MOD (n*y, y)')
      else
         g$mod =  1.0
      end if
      return
      end


      real function g$dim (x, y)

C Purpose:  Return the derivative value for DIM.
C           d (DIM (x, y)) / d x.
C           d (DIM (x, y)) / d y = - d (DIM (x, y)) / d x.
C Input parameters:
C     x, y   Arguments to DIM
C Returned value:
C     DIM (x, y) := max (x - y, 0)
C                 = x - y  if x ge y
C                 = 0      otherwise
C     d. / dx =  1  if x gt y, 0 if x lt y, else NoLmVl from g$error
```

```
C     d. / dy = -1  if x gt y, 0 if x lt y, else NoLmVl from g$error
C Rationale:  At the points of nondifferentiability, the default
C     for NoLmVl = 0 is a generalized gradient.  An alternate choice
C     would be 1/2, the average of the limits from both sides.
C Usage in ADIFOR-generated code:
C       Z = DIM (X, Y)
C       r$1 = dim (x, y)
C       temp = g$dim (x, y)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$) - temp * g$y(g$i$)
C  99990 continue
C or
C       temp = g$dim (x, y, r$1)
C       xbar = xbar + zbar * temp
C       ybar = ybar - zbar * temp

      real x, y

      if (x .gt. y) then
         g$dim = 1.0
      else if (x .lt. y) then
         g$dim = 0.0
      else
         g$dim = g$error (13,
     +           'Computed the derivative of DIM (x, x)')
      end if
      return
      end


      real function g$max (x, y)

C Purpose:  Return the derivative value for MAX.
C           d (MAX (x, y)) / d x.
C           d (MAX (x, y)) / d y = 1 - d (MAX (x, y)) / d x.
C Input parameters:
C     x, y   Arguments to MAX
C Returned value:
C     g$max = 1       if x gt y
C             0       if x lt y
C             TieVal  if x = y
C Rationale:  At the points of nondifferentiability, the default
C     for TieVal = 1/2 is a generalized gradient.
C Usage in ADIFOR-generated code:
C       Z = MAX (X, Y)
C       r$1 = max (x, y)
C       temp = g$max (x, y)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$) + (1.0 - temp) * g$y(g$i$)
C  99990 continue
C or
C       temp = g$max (x, y, r$1)
C       xbar = xbar + zbar * temp
C       ybar = ybar + zbar * (1.0 - temp)
```

```
      real x, y

      if (x .gt. y) then
         g$max = 1.0
      else if (x .lt. y) then
         g$max = 0.0
      else
         g$max = g$error (15,
     +           'Computed the derivative of MAX (x, x)')
      end if
      return
      end


      real function g$min (x, y)

C Purpose:  Return the derivative value for MIN.
C           d (MIN (x, y)) / d x.
C           d (MIN (x, y)) / d y = 1 - d (MIN (x, y)) / d x.
C Input parameters:
C     x, y   Arguments to MIN
C Returned value:
C     g$min = 1       if x lt y
C             0       if x gt y
C             TieVal  if x = y
C Rationale:  At the points of nondifferentiability, the default
C     for TieVal = 1/2 is a generalized gradient.
C Usage in ADIFOR-generated code:
C       Z = MIN (X, Y)
C       r$1 = min (x, y)
C       temp = g$min (x, y)
C       do 99990 g$i$ = 1, g$p$
C          g$z(g$i$) = temp * g$x(g$i$) + (1.0 - temp) * g$y(g$i$)
C  99990 continue
C or
C       temp = g$min (x, y, r$1)
C       xbar = xbar + zbar * temp
C       ybar = ybar + zbar * (1.0 - temp)

      real x, y

      if (x .lt. y) then
         g$min = 1.0
      else if (x .gt. y) then
         g$min = 0.0
      else
         g$min = g$error (15,
     +           'Computed the derivative of MIN (x, x)')
      end if
      return
      end
```

Appendix B. Error Handling for the ** Operator

```
C     Purpose: Error Handling for the ** operator
C     Author:  A. Griewank  Oct. 24
C
C     Approach:
C     We try to catch the situations where the function
C     value itself is at least mathematically defined but the
C     derivatives are not. In contrast to the other intrinsics,
C     we prefer to do everything in-line, except for a call to
C     g$error in the exceptional cases.  In line here means that
C     we branch directly to the bar quantity assignments.
C
C     Depending on whether we wish to differentiate x**y with respect
C     to x or with respect to y, we should consider it as a 'power'
C     or as an 'exponential', respectively.  Correspondingly, the error
C     classification number should be 5 or 10, respectively, in the
C     call to g$error.  When both x and y are active reals, we consider
C     x**y simultaneously as a power and as an exponential.
C
C     We assume that x**y has a well-defined value if x >= 0 or y is
C     an integer with 0**0 = 0.  On particular systems, the values may
C     be defined differently if x = 0 or y = 0, and there will be
C     overflow when y < 1 and x is sufficiently small.  We will do
C     nothing about this because we would otherwise also have to
C     safeguard simple divisions.
C
C     For fixed y, the derivative of x**y with respect to x is
C     mathematically defined except when x = 0, and 0 < y < 1. This
C     case is a generalization of the square root situation.
C     Therefore, we set the derivative to InfVal.  When y = 0, we set
C     the derivative with respect to x to zero and do not call g$error,
C     even if x = 0.
C
C     For fixed x, the derivative of x**y with respect to y is
C     mathematically defined except when x <= 0, and the value of y
C     is an integer.  When x is negative, x**y is not defined for
C     any fractional y.  Therefore, we set the value of the derivative
C     to NoLmVl at integral values of y.  When x = 0, the derivative
C     is zero for all y > 0.  For y = 0, we may again use NoLmVl.
C
C
C     Note that ybar remains unchanged if x = 0 and y > 0.
C
C     When y is not active, there is no ybar, and the second part
C     of the calculation can be omitted.  If y is of type integer, then
C     the first part can be reduced to the single statement
C
C     if(x .ne. 0.0d0) xbar = xbar + y * zbar * r$0/x
C
C     If x is passive, there is no xbar, and the first part can
C     be omitted.  Finally, if either a or y are constants that can
C     be evaluated at compile time, further simplifications are
C     possible.
C
C*****************************************************************
```

```
C
C     Original code segment:
C
C      real x,y,z
C      x = ..
C      y = ..
C      z = x**y
C
C
C*****************************************************************
C
C     Processed code with x, y, and thus z active:

      program main
      real y
      call test(0.1,3.0)
      call test(-0.1,3.0)
      call test(-0.1,4.0)
      call test(-0.1,5.0)
      call test(-0.1,-1.0)
      call test(-0.1,-2.0)
      call test(0.0,0.3)
      end


      subroutine test(x,y)
      integer g$pmax, g$p$, g$i
      parameter (g$pmax = 50)
      real x,y,z
      real g$x(g$pmax)
      real g$y(g$pmax)
      real g$z(g$pmax)
      real dummy,xbar,ybar,r$0

      dummy = 0.1
      g$p$  = 3
      do 10 g$i$ = 1, g$p$
        g$x(g$i) = dummy
        g$y(g$i) = dummy
10    continue
C     z = x**y
      r$0 = x**y
      zbar = 1.0
      xbar = 0.0
      ybar = 0.0
C
C     First, do the derivative with respect to x.
C
      if (x .ne. 0.0) then
 xbar = xbar + y * zbar * r$0/x
      else
         if ((y .lt. 1.0) .and. (y .gt. 0.0)) then
            xbar = xbar + zbar
     +              * g$error (5, 'Fractional power of zero')
         end if
      end if

C     Note that xbar remains unchanged if x = 0 and y >= 1
C
C     Second, do the derivative with respect to y.

      if (x.gt.0.0) then
         ybar = ybar + zbar * r$0*log(x)
      else
         if ((x .lt. 0.0) .or. (y .eq. 0.0)) then
            ybar = ybar + zbar
     +               * g$error (10, 'Negative basis or 0**0')
         end if
      end if
      z = r$0
      write(6,*) z, xbar, ybar
      do 20 g$i$ = 1, g$p$
        g$z(g$i) = xbar*g$x(g$i) + ybar*g$y(g$i)
20    continue
      return
      end
```

Appendix C. Code to Test ADIFOR Error Handling

```
c  File:  WHAT_ERR1.f                 16-OCT-1991

c  Author:  George Corliss
c  Purpose: Look at generated code to consider what error detection
c           tests are required.

      program what

      real x, y

      x = 0.0
      call all (x, y)
      stop
      end


      SUBROUTINE ALL (X, Y)
      REAL X, Y, Z, A, B, C, D(2,2), F(2), R(2)
      REAL wc, w, gammar, one,gammai, zero, d1, d2
      REAL tgd1r, tgd1i, tgd2r, tgd2i
      INTEGER I, J, K

c  IT IS NOT INTENDED THAT THIS ROUTINE EXECUTE MEANINGFULLY!

      A = 3.0
      B = 10.0**50
      D(1,1) = X
      R(1) = X


c  Error Class 1:  User Function Is not Defined.
c  =============================================

c  Example 1-1.  User guards error.
      IF (X .NE. 0.0) THEN
         Y = A / X
      ELSE
         CALL ERROR ('Please do not divide by zero.')
      END IF

c  Example 1-2.  Error is unguarded.
      Y = A / X
      Y = asin (A * X)

c  Example 1-3.  Error is overflow.
      Y = X / B


c  Error Class 2:  Differentiable Functions -- Overflows.
c  ======================================================

c  Example 2-1.  Derivative evaluation can overflow.
      Y = A / X + TAN (X) + LOG (X) + LOG10 (X) + TANH (X)
      Z = Y
      Y = -X / (Z * Z * Z)


c  Error Class 3:  Nondifferentiable Functions -- lim f' = \pm \infty.
c  =======================================================================

c Example 3-1.  At points of nondifferentiability.
      Z = X
      Z = 1.0
      Y = SQRT (1.0 - Z) + ASIN (Z) + ACOS (Z) + Z ** X

c Example 3-2.  Near points of nondifferentiability.
      Z = X
      Z = 0.999999
      Y = SQRT (1.0 - Z) + ASIN (Z) + ACOS (Z) + Z ** X

c Example 3-3.  Fischer's root problem.  Function is differentiable,
c              but intermediate results are not.
      Y = SQRT (X*X*X*X + Z*Z*Z*Z)


c  Error Class 4:  Nonsmooth Functions -- lim f' does not exist.
c  ==================================================================

c Example 4-1.  At points of nondifferentiability.
      Z = 0.0
      Y = ABS (Z) + SIGN (X, Z) + AINT (Z) + MAX (Z, X) + MIN (Z, X)
     +    + DIM (X, X)

c Example 4-2.  Near points of nondifferentiability.
c              Can we detect how near we are?
      Z = 10.0**(-30)
      Y = ABS (Z) + SIGN (X, Z) + AINT (Z) + MAX (Z, X) + MIN (Z, X)
     +    + DIM (X, X)


c  Error Class 5:  Problems of Domains -- Branchings.
c  ==================================================

c  Example 5-1.  ABS using if statements.
      IF (X .GE. 0.0) THEN
         Y = X
      ELSE
         Y = - X
      END IF

      IF (X .GT. 0.0) THEN
         Y = X
      ELSE
         Y = - X
      END IF

      IF (X .GT. 0.0) THEN
         Y = X
      ELSE IF (X .LT. 0.0) THEN
         Y = - X
      ELSE
         Y = 0.0
      END IF

c Example 5-2.  Fischer's branch
      IF (X .EQ. 1.0) THEN
         Y = 1
      ELSE
         Y = X*X
      END IF

c Example 5-3.  Fischer's 2 by 2 Gaussian elimination
      IF (D(1,1) .EQ. 0.0) THEN
         F(2) = R(1) / D(1,2)
         F(1) = (R(2) - D(2,2) * F(2)) / D(2,1)
      ELSE
         C = D(2,1) / D(1,1)
         D(2,2) = D(2,2) - C * D(1,2)
         R(2) = R(2) - C * R(1)
         F(2) = R(2) / D(2,2)
         F(1) = (R(1) - D(1,2) * F(2)) / D(1,1)
      END IF
      Y = F(1) * F(2)

c Example 5-4.  Janet Rogers -- Interplay of IF and ABS
         if (wc.le.w) then
            gammar = sqrt(abs((one-(wc/w)**2)))*w/c
            gammai = zero
            tgd1r   = sin(gammar*d1)/cos(gammar*d1)
            tgd1i   = zero
            tgd2r   = sin(gammar*d2)/cos(gammar*d2)
            tgd2i   = zero
         else
            gammar = zero
            gammai = sqrt(abs((one-(wc/w)**2)))*w/c
            tgd1r   = zero
            tgd1i   = -(exp(-gammai*d1)-exp(gammai*d1)) /
     +                 (exp(-gammai*d1)+exp(gammai*d1))
            tgd2r   = zero
            tgd2i   = -(exp(-gammai*d2)-exp(gammai*d2)) /
     +                 (exp(-gammai*d2)+exp(gammai*d2))
         end if


c  Error Class 6:  Mathematical Pitfalls.
c  ======================================


c  List of all elementary functions
c  ================================

c Example 7-1.  As simple assignments.
      Y = ABS (X)
      Y = ACOS (X)
      Y = AINT (X)
      Y = LOG (X)
      Y = ALOG (X)
      Y = LOG10 (X)
      Y = ALOG10(X)
      Y = MAX   (X, Z)
      Y = MIN   (X, Z)
      Y = ATAN (X)
      Y = ATAN2 (X, Z)
      Y = COS (X)
      Y = COSH (X)
      Y = EXP (X)
      Y = MOD (X, Z)
      Y = SIGN (X, Z)
      Y = SIN (X)
      Y = SINH (X)
      Y = SQRT (X)
      Y = TAN (X)
      Y = TANH (X)

c Example 7-2.  In complicated assignments.
      Y = ABS(X) + ACOS(X) + AINT(X) + ALOG(X) + ALOG10(X)
      Y = ASIN  (X) + ATAN  (X) + ATAN2 (X, Z)
      Y = COS   (X) + COSH  (X) + EXP   (X) + MAX   (X, Z)
      Y = MIN   (X, Z) + SIGN  (X, Z)
      Y = SIN   (X) + SINH  (X) + SQRT  (X) + TAN   (X) + TANH  (X)

      RETURN
      END


      SUBROUTINE ERROR (MSG)
      CHARACTER*1 MSG
      RETURN
      END
```

References

[1] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *Generating derivative codes from Fortran programs*, Preprint MCS–P263–0991, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 1991. Also appeared as Technical Report 91185, Center for Research in Parallel Computation, Rice University, Houston, Texas.

[2] C. Bischof and P. Hovland, *Using ADIFOR to compute dense and sparse Jacobians*, Technical Memorandum MCS–TM–158, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, October 1991.

[3] F. Clark, *Optimization and Nonsmooth Analysis*, John Wiley and Sons, New York, 1983.

[4] H. Fischer, *Special problems in automatic differentiation*, In Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. Corliss (eds.), SIAM, Philadelphia, Pennsylvania, 1991, to appear.

[5] G. Kedem, *Automatic differentiation of computer programs*, ACM Transactions on Mathematical Software, 6 (1980), pp. 150–165.