ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-164

ADIFOR Working Note #9

Getting Started with ADIFOR

by

Christian Bischof, Alan Carle,* George Corliss,
Andreas Griewank, and Paul Hovland

Mathematics and Computer Science Division

Technical Memorandum No. 164

June 1993

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, through NSF Cooperative Agreement No. CCR-8809615, by the W. M. Keck Foundation, and by NASA Purchase Order L25935D.

*Address: Center for Research on Parallel Computation, Rice University, P.O. Box 1892, Houston, TX 77251

Contents

ABSTRACT

ADIFOR is a source translator that, given a collection of subroutines to compute a function f, generates Fortran 77 code for computing the derivatives of this function. This paper describes step by step how to use ADIFOR to generate derivative code. It also describes common misunderstandings as well as workarounds for current shortcomings. Familiarity with UNIX™ and Fortran 77 is assumed. Also desirable is a basic understanding of automatic differentiation (see [3, 7, 9] for an introduction). ADIFOR is a research project, and thus likely to be subject to many changes. This document will change to reflect the ADIFOR changes. New users are advised to make sure that the manual version in hand is the current up-to-date one (see Section 8).

1 Introduction

Many problems in computational science require the evaluation of a mathematical function, as well as the derivatives of that function with respect to certain independent variables. ADIFOR provides a mechanism for the automatic generation of Fortran code for the computation of derivatives, using the Fortran code for the evaluation of the function as input. More information on ADIFOR can be found in [2, 3, 4, 5].

The organization of this paper is as follows. The next section is devoted to a step-by-step description of how to process a code using ADIFOR, and an explanation of how ADIFOR-generated code may be incorporated into a program. This example is intentionally simple, and ignores many subtle issues. Section 3 is devoted to describing the criteria that must be satisfied prior to processing; i.e., the restrictions imposed by ADIFOR on the user code. Section 4 covers some of the mathematical pitfalls associated with automatic differentiation. Section 5 describes some of the shortcuts and workarounds in using ADIFOR, while Section 6 covers a few advanced topics. Section 7 gives a more advanced example of processing a program with ADIFOR. Section 8 deals with running ADIFOR at Argonne. The final section presents some common problems encountered when using ADIFOR.

2 A Simple Example

We demonstrate the use of ADIFOR using the simple program shown in Figures 1 and 2. It shows a simple Newton iteration being used to minimize Rosenbrock's function. The routines `DLANGE` and `DGESV` from the LAPACK package [1] are used to compute the norm of `y` and to solve the linear system $\frac{dy}{dx}s = -y$. This program and all of the auxillary files mentioned in this section can be found in `/usr/local/adifor/examples/rosenbrock` at Argonne. For details on using ADIFOR at Argonne, see section 8. Rosenbrock's function is used only for illustrative purposes. It is not indicative of the power of ADIFOR, which has processed programs over 12,000 lines in length. Our goal will be to replace the subroutine `fprime` which approximates $\frac{\mathtt{dy}}{\mathtt{dx}}$ using central divided differences, with an ADIFOR-generated derivative code.

Step 1: Create an ADIFOR Script File

In order to create the ADIFOR script file, the user must first identify the function to be differentiated. In most cases, the function to be differentiated corresponds to a subroutine, like `func` in the example program. This subroutine is referred to as the *top-level routine*. The user must also identify the variables that correspond to the independent and dependent variables of the function with respect to differentiation. In the example, `y` is a dependent variable, and `x` is the independent variable.

```
      PROGRAM NEWTON

C     .. Local Scalars ..
      DOUBLE PRECISION DUMMY,TOL
      INTEGER INFO
C     ..
C     .. Local Arrays ..
      DOUBLE PRECISION X(2),Y(2),YPRIME(2,2)
      INTEGER IPIV(2)
C     ..
C     .. External Subroutines ..
      EXTERNAL DGESV,FPRIME,FUNC
C     ..
C     .. External Functions ..
      DOUBLE PRECISION DLANGE
      EXTERNAL DLANGE
C     ..
      TOL = 1.0E-12

      WRITE (*,FMT=*) 'Input 2-element starting vector '
      READ (*,FMT=*) X(1),X(2)

      CALL FUNC(X,Y)
c
c check for convergence
c      (very simplistic, based only on norm of Y)
c
   10 IF (DLANGE('1',2,1,Y,2,DUMMY).LT.TOL) GO TO 20
c
c compute function and Jacobian at current iterate
c
      CALL FPRIME(X,Y,YPRIME)
c
c solve J * s = - f
c      and update x = x + s
c
      Y(1) = -Y(1)
      Y(2) = -Y(2)
      CALL DGESV(2,1,YPRIME,2,IPIV,Y,2,INFO)
      X(1) = X(1) + Y(1)
      X(2) = X(2) + Y(2)
c
c compute new function value
c
      CALL FUNC(X,Y)
      WRITE (*,FMT=1000) 'Current Function Value:',Y(1),Y(2)
      GO TO 10

   20 CONTINUE
      WRITE (*,FMT=1000) 'Minimum is approximately:',X(1),X(2)
 1000 FORMAT (a,1x,2 (d15.8,2x))
      END
```

Figure 1. A Simple Implementation of Newton's Method

```
      SUBROUTINE FUNC(X,Y)
      DOUBLE PRECISION X(2),Y(2)

      Y(1) = 10.0* (X(2)-X(1)*X(1))
      Y(2) = 1.0 - X(1)
      RETURN
      END



      SUBROUTINE FPRIME(X,Y,YPRIME)
c
c approximates derivatives of Func by central differences.
c
      .. Array Arguments ..
      DOUBLE PRECISION X(2),Y(2),YPRIME(2,2)
C     .. Local Scalars ..
      DOUBLE PRECISION H
C     .. Local Arrays ..
      DOUBLE PRECISION XH(2),YM(2),YP(2)
C     .. External Subroutines ..
      EXTERNAL FUNC
C     ..
      IF (X(1).EQ.0.0) THEN
          H = 1.0e-7
      ELSE
          H = X(1)*1.0e-7
      END IF
      XH(1) = X(1) - H
      XH(2) = X(2)
      CALL FUNC(XH,YM)
      XH(1) = X(1) + H
      XH(2) = X(2)
      CALL FUNC(XH,YP)
      YPRIME(1,1) = (YP(1)-YM(1))/ (2.0*H)
      YPRIME(2,1) = (YP(2)-YM(2))/ (2.0*H)

      IF (X(2).EQ.0.0) THEN
          H = 1.0e-7
      ELSE
          H = X(2)*1.0e-7
      END IF
      XH(1) = X(1)
      XH(2) = X(2) - H
      CALL FUNC(XH,YM)
      XH(1) = X(1)
      XH(2) = X(2) + H
      CALL FUNC(XH,YP)
      YPRIME(1,2) = (YP(1)-YM(1))/ (2.0*H)
      YPRIME(2,2) = (YP(2)-YM(2))/ (2.0*H)

      RETURN
      END
```

Figure 2. Rosenbrock's Function and Divided-Difference Approximations of the Jacobian

The ADIFOR script file communicates this information to ADIFOR; it identifies the top-level routine, the independent variables, the dependent variables, the upper bound on the size of gradient objects, and (optionally) the separation parameter ADIFOR uses for generating derivative names. There is no formal requirement for the name of the script file, but our informal convention is to use the name of the subroutine to be differentiated (the top-level subroutine) with a `adf` extension. For our example, we might create the file `func.adf`:

```
TOP func
IVARS x
OVARS y
PMAX 2
SEP $
```

The meaning of the various entries is as follows:

TOP:

The directive `TOP` denotes the name of the top-level subroutine in the program composition.

IVARS and OVARS:

The comma-separated lists `IVARS` (Input variables) and `OVARS` (Output variables) denote which variables are *independent* or *dependent* with respect to differentiation. All variables in these lists must be of type real or double precision, since complex, integer and character variables are not eligible for differentiation in ADIFOR. A variable may be designated as independent, dependent, or both. The current ADIFOR version does not allow spaces around commas separating the names of dependent or independent variables. Multiple `IVARS` (or `OVARS`) lines may appear in the script file. Any variable appearing in an IVARS (or OVARS) statement is treated as an input (or output) variable. For our example, we wish to compute the derivatives of `y` with respect to `x`. Thus, the list of `OVARS` has only one item, `y`. Similarly, the list of `IVARS` is simple `x`.

PMAX:

Because Fortran 77 does not allow dynamic memory allocation and because of complex issues surround storage association in Fortran, ADIFOR requires that the user specify an upper bound on the size of the gradient objects, `PMAX`. The value `PMAX` must be greater than zero. `PMAX` is an upper bound on the number of independent variables in an invocation of the ADIFOR-generated code. As an example, one might want to compute derivatives with respect to an array `x(1:1000)`, but, because of storage limitations, the ADIFOR-generated code will be used to compute only 10 Jacobian columns at a time. Then a value of 10 for `PMAX` is suitable choice. Also for sparse Jacobians, `PMAX` need not be the total number of independent variables. These issues are discussed in detail in [5], which is available on-line (see Section 8). For our example program, we choose a value of 2 for `PMAX`, because `x` is the only independent variable, and it has 2 elements.

SEP:

ADIFOR uses the character specified by `SEP` for generating names for Fortran variables used in derivative computations. The default for the separator character is '$' and this character will be chosen if there is no `SEP` directive in the script file. Users can override this choice by specifying a different character with the `SEP` directive. If the dollar-sign $ was chosen as the separator, the compiler on your target system must accept variable names containing a dollar-sign (e.g. Sun systems). You must specifiy an alternative value for `SEP` if the compiler on your system does not accept dollar-signs (e.g. Cray systems).

Step 2: Create a Composition File

The next step in processing a program using ADIFOR is to create a "composition file," i.e. a file listing the files to be processed by ADIFOR. These files must make up an entire program, which we refer to as a composition. As with the ADIFOR script file, there is no formal requirement for the name of the composition file, but our informal convention is to use the name of the subroutine to be differentiated (the top-level subroutine) with a `comp` extension.

If source code for the entire program is available and does not violate any of the restrictions enumerated in Section 3, it is best to submit the entire program to ADIFOR (i.e., specify every file in the program as part of the composition), so that a complete analysis is possible. However, it is often the case that parts of the program which are not part of the function evaluation (such as the LAPACK routines in our example) are not available as source code or contain constructs which violate the restrictions on code to be processed by ADIFOR. In this situation, we can trick ADIFOR into believing it has a complete program and process only the top-level subroutine and those subroutines below it in the call-tree. By creating a file called `dummy.f` consisting of the lines

```
program dummy
end
```

and including `dummy.f` in the list of files in the composition, we can convince ADIFOR that it is working with a complete program. In addition to `dummy.f`, the composition file should list the file or files containing the top-level subroutine and all subroutines called by it. If the program in our example is split into `func.f`, `newton.f`, and `fprime.f`, then `func.f` should be included in the composition file, since it contains subroutine `func`, the top-level subroutine. Thus, the composition file for our example (which we will call `func.comp`) would look like:

```
func.f
dummy.f
```

ADIFOR expects one name per line, *with no leading or trailing blanks.* If a composition file does include trailing blanks, ADIFOR will interpret those blanks as part of the filename, and report that the specified file can not be found. The composition file must also not contain any blank lines.

Step 3: Invoke ADIFOR

Two parameters must be specified when running ADIFOR: the first parameter is the name of the script file, and the second parameter is the name of the composition file. So we would invoke ADIFOR by issuing the command

```
adifor <script file> <composition file>
```

In our example, if we call our composition file `func.comp` and our script file `func.adf`, the command

```
adifor func.adf func.comp
```

should be used to invoke ADIFOR. Consequently, `ADIFOR` creates

- two subdirectories `ADDIR` and `adtmp`, which contain internal information to be used during the translation phase, and
- a file called `ADMakefile`, to be used with the UNIX `make` utility. This file allows the user to specify certain parameters to the translator program (see Section 6.2 for details) and is used to invoke the translator (called ADTRANS).

The `ADMakefile` generated for our example program is:

```
ADMISC = # SAXPY
ADFLAVORS = UNOPT OPT
WHICH :sh = adwhich adtrans
all: $(ADFLAVORS)
UNOPT: func.3.unopt.f
OPT:   func.3.f
func.3.unopt.f func.3.f: adtmp/func.0 $(WHICH) /home/bischof/newton/ADDIR/func/source
adtrans adtmp/func.0 $(ADFLAVORS) $(ADMISC)
```

Next, we issue the command

```
make -f ADMakefile
```

to invoke the ADIFOR translator ADTRANS. ADTRANS generates derivative code for the top-level subroutine and the subroutines called (directly or indirectly) by it. In our example, it only generates new code for `func`, namely `func.3.unopt.f` and `func.3.f`. The 3 in the names of these files and in the name of the generated subroutine is a hexidecimal encoding of the active variables in the subroutine (see Section 4 for an explanation of active variables). Both are versions of the ADIFOR-generated subroutine, `g$func$3`, but the latter has been "cleaned up" to eliminate unnecessary computations, such as multiplications by 1.0 and additions of 0.0. The "optimized" version of `g$func$3` is shown in Figure 3.

We mention, that, in general, ADTRANS generates variable names that are longer than six characters. We also mention that it is a good idea to make sure that the files specified in the program composition file compile correctly and adhere to the Fortran 77 standard (see 8.3) before submitting them to ADIFOR. ADIFOR will complain about syntax errors, but its error messages are likely to be less refined (more cryptic).

Step 4: Incorporate ADIFOR-generated Subroutine

Now we wish to incorporate the ADIFOR-generated subroutine into our program. In order to do this, we need to perform three steps.

1. **Allocate the gradients in the calling module.**

   The user should carefully check the ADIFOR-generated top-level subroutine to determine which gradient objects must be passed to the top-level routine through parameters or common blocks. The user must then declare and allocate each of these gradient objects. For our small example, the declarations are:

   ```
   double precision g$x(pmax,2), g$y(pmax,2)
   ```

   where `pmax` is an integer constant (Fortran `PARAMETER`) whose value is at least the length of the gradient objects (2, in this case).

2. **Initialize the seed matrix.**

   ADIFOR produces code to compute the original function, as well as the matrix-matrix product `J*S`, where `J` is the Jacobian of the "function" with respect to the user-defined independent variables, and the seed matrix `S` is the composition of derivative objects corresponding to the independent variables. This interface is flexible; by setting `S=x`, one can compute the matrix-vector product `Jx`, or by setting `S=I`, with `I` the identity matrix, one can compute the whole Jacobian `J`. The former functionality if of particular importance, because the cost of computing derivatives using the forward mode of automatic differentiation is directly proportional to the number of columns in the seed matrix (and consequently in the product `J*S`). Thus, the cost of computing `Jx`, where `x` is an n-element column vector, by computing `J` then multiplying by `x` is n times as expensive

```
      subroutine g$func$3(g$p$, x, g$x, ldg$x, y, g$y, ldg$y)
C
C       Formal y is active.
C       Formal x is active.
C
        integer g$p$
        integer g$pmax$
        parameter (g$pmax$ = 2)
        integer g$i$
        double precision d$3
        double precision d$3bar
        double precision d$2
        double precision d$2bar
C

C       .. Array Arguments ..
        double precision x(2), y(2)
        double precision g$x(ldg$x, 2), g$y(ldg$y, 2)
        integer ldg$x
        integer ldg$y
C       ..
        if (g$p$ .gt. g$pmax$) then
          print *, 'Parameter g$p is greater than g$pmax.'
          stop
        endif
C       y(1) = 10.0 * (x(2) - x(1) * x(1))
        d$2 = x(1)
        d$3 = x(1)
        d$2bar = -10.0 * d$3
        d$3bar = -10.0 * d$2
        do 99997 g$i$ = 1, g$p$
          g$y(g$i$, 1) = 10.0 * g$x(g$i$, 2) + d$2bar * g$x(g$i$, 1) + d
     *$3bar * g$x(g$i$, 1)
99997   continue
        y(1) = 10.0 * (x(2) - d$2 * d$3)
C       y(2) = 1.0 - x(1)
        do 99996 g$i$ = 1, g$p$
          g$y(g$i$, 2) = -g$x(g$i$, 1)
99996   continue
        y(2) = 1.0 - x(1)
        return
      end
```

Figure 3. The ADIFOR-generated Code for Subroutine `func`

as computing `Jx` directly (by setting `S=x`). Other initializations of `S` allow one to exploit a known sparsity structure of `J` (see [5]). For our example, all we wish to compute is the Jacobian, so we should let

$$\texttt{g\$x} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

More details on the initialization of seed matrices are described in Section 7.4 and in [5] (this document can be found in `/usr/local/adifor/doc` – see Section 8).

3. **Call the ADIFOR-generated top-level subroutine**

   The ADIFOR-generated subroutine computes *both* the function value and the value of the derivatives. So, in our example, we can replace both the call to `func` and `fprime` by a call to `g$func$3`. In other applications, it may still be necessary to call both the original and ADIFOR-generated routine. In the call to the ADIFOR-generated top-level subroutine, the parameter `g$p$` should be set equal to the number of rows in the gradient objects, all of the `g$` variables for independent variables should be initialized to the appropriate seed, and all of the `ldg$` variables should be set equal to the leading dimension with which the corresponding gradient objects (`g$` variables) were declared. Thus, for our simple example, the call would look like:

```
call g$func$3(2, x, g$x, pmax, y, g$y, pmax)
```

For our example, the new driver is shown in Figure 4*. Note that ADIFOR computes the transpose of the Jacobian (see [5] for details). Hence, we must re-transpose `g$y` before passing it to `dgesv`. Together with the subroutine `func` and the subroutine shown in Figure 3, the new program replaces the program shown in Figure 1.

Again, the compiler on your system must be able to deal with variable names longer than six characters and must accept the `SEP` character chosen.

Step 5: Compile and Link

After a suitable driver has been developed, the ADIFOR-generated code, the driver, and any other modules necessary to form a complete program should be compiled. The compiled modules should then be linked, together with the exception handling routines for the Fortran intrinsic functions. These routines can be found in source and compiled form in the directory `/usr/local/adifor/lib` under the name `intrinsic.`*ext*, where *ext* is any of `f` (Fortran source code), `sparc.o` (object code for SPARCs and other Sun 4 compatibles), or `rs6000.o` (object code for IBM RS6000s). When possible, instruct the compiler to use as much space for internal tables as possible, because the code generated by ADIFOR may be much longer than the original code. The extra space may enable the compiler to do a much better job of optimizing. For example, the appropriate flag for the Cray compiler is `-wf "-o aggress"`.

3 Restrictions

Before ADIFOR may be applied to a composition, there are certain criteria which must be satisfied. Some restrictions arise as a direct result of incompatibilities between the various dialects of Fortran and the goal of using automatic differentiation of standard Fortran to compute the derivatives of a function. Others are more temporary in nature, and may be removed in future versions of ADIFOR.

*Some comments were removed to fit the program on one page.

```
      PROGRAM ADNEWTON
C     .. Parameters ..
      INTEGER PMAX
      PARAMETER (PMAX=2)
C     .. Local Scalars ..
      DOUBLE PRECISION DUMMY,TEMP,TOL
      INTEGER INFO
C     .. Local Arrays ..
      DOUBLE PRECISION G$X(PMAX,2),G$Y(PMAX,2),X(2),Y(2)
      INTEGER IPIV(2)
C     .. External Functions ..
      DOUBLE PRECISION DLANGE
      EXTERNAL DLANGE
C     ..
      TOL = 1.0E-12
      WRITE (*,FMT=*) 'Input 2-element starting vector '
      READ (*,FMT=*) X(1),X(2)

      CALL FUNC(X,Y)
c
   10 IF (DLANGE('1',2,1,Y,2,DUMMY).LT.TOL) GO TO 20
c
c compute function and Jacobian at current iterate
c
      G$X(1,1) = 1.0
      G$X(1,2) = 0.0
      G$X(2,1) = 0.0
      G$X(2,2) = 1.0
      CALL G$FUNC$3(2,X,G$X,PMAX,Y,G$Y,PMAX)
c
c transpose g$y
c
      TEMP = G$Y(2,1)
      G$Y(2,1) = G$Y(1,2)
      G$Y(1,2) = TEMP
c
c solve J * s = - f and update x = x + s
c
      Y(1) = -Y(1)
      Y(2) = -Y(2)
      CALL DGESV(2,1,G$Y,PMAX,IPIV,Y,2,INFO)
      X(1) = X(1) + Y(1)
      X(2) = X(2) + Y(2)
c
c compute new function value
c
      CALL FUNC(X,Y)
      WRITE (*,FMT=1000) 'Current Function Value:',Y(1),Y(2)
      GO TO 10
   20 CONTINUE
      WRITE (*,FMT=1000) 'Root is approximately:',X(1),X(2)
 1000 FORMAT (a,1x,2 (d15.8,2x))
      END
```

Figure 4. The Driver for the Newton Program Using ADIFOR-generated Code

- **Composition must conform to the Fortran 77 standard**

  ADIFOR recognizes standard Fortran 77 syntax. If a program uses non-standard extensions, ADIFOR will probably not accept them. For portability reasons, it is probably a good idea anyway to make sure that all code is standard-conforming. In particular, ADIFOR will not correctly deal with nonstandard intrinsic or type conversion functions, such as `arsin()`, `arcos()`, and `dfloat()`. These should be replaced with standard functions like `asin()`, `acos()`, and `dble()`. Also not supported are system calls such as `etime()`. In most cases, such calls do not have an effect on function evaluation, and may be removed or commented out prior to processing by ADIFOR. Another nonstandard feature which most compilers support but ADIFOR does not is the `NAMELIST` command. One nonstandard feature which ADIFOR does support is identifier names longer than 6 characters. Many compilers support variable names up to 32 characters in length.

- **A top-level subroutine must be present**

  There must exist some subroutine such that all independent and dependent variables are passed as parameters or in common blocks to and from this subroutine. Often, the computation to be differentiated is embedded in a main program, and must first be encapsulated in a subroutine to be suitable for processing with ADIFOR.

  For example, take the following example:

```
program main
  ...
read(*,*) x(1)
t= result of some computation involving x(1)
  ...
read(*,*) x(2)
  ...
y= result of some computation involving x(1) and x(2)
  ...
y= result of some other computations involving x(1) and x(2)
  ...
end
```

  To extract a subroutine suitable for using ADIFOR to generate code for $\frac{\partial \mathtt{y}}{\partial \mathtt{x}(1)}$ and $\frac{\partial \mathtt{y}}{\partial \mathtt{x}(2)}$, we must:

  - Rearrange the computation such that `x(1)` and `x(2)` are initialized before any computations involving `x(1)` and `x(2)` are performed.
  - Depending on whether we include the second assignment statement `y` to encapsulate the desired function, the ADIFOR-generated code will return the derivatives of the first or second value assigned to `y`, but never both. If we want both, one solution is to expand `y` to an array `y(1)` and `y(2)` .
  - Encapsulate the computations in a subroutine. Thus, our original program becomes:

```
program main
  ...
read(*,*) x(1)
  ...
read(*,*) x(2)
  ...
call subr(x,y)
  ...
end

subroutine subr(x,y)
  ...
t= result of some computation involving x(1)
```

```
  ...
y(1)= result of some computation involving x(1) and x(2)
  ...
y(2)= result of some other computations involving x(1) and x(2)
return
end
```

- **Functions in the active subtree**

  The active subtree consists of all the modules needed to execute the top-level subroutine. Currently, ADIFOR does not support `FUNCTION` calls. The user must change all of the functions in the program composition to `SUBROUTINE`s. For example the following code will not be accepted by ADIFOR,

```
program main
  ...
result = add(x,y)
  ...
end

function add(a,b)
  ...
r = a+b
return r
end
```

  and should be changed to,

```
program main
  ...
call add(x,y,result)
  ...
end

subroutine add(a,b,r)
  ...
r = a+b
end
```

  An alternative is to use the ADPRE preprocessor described in section 5.2.

- **I/O functions**

  Sometimes the values of independent variables are read or computed within the active subtree (that is, within the subtree of procedures below the top-level subroutine). This does not pose a problem, so long as the independent variables are declared in the top-level subroutine and in the procedure that calls it, and I/O functions are handled properly.

  `READ` and `WRITE` statements in the active subtree are echoed into the ADIFOR-generated code *with no changes*. If an active subtree contains I/O statements, the user should think carefully. `READ`ing the value of an independent variable inside the active subtree or `READ`ing a "constant" whose value has been computed elsewhere depending on one or more independent variables may produce incorrect derivative results. However, this situation often occurs in codes which write out states for possible restarts of computations. Workarounds are discussed in Section 5.1.

- **Variables not visible at the top-level**

  Consider the program:

```
program main
  ...
call foo(x,y)
  ...
end

subroutine foo(x,y)
a = x+1
y = x*x
b = x/2
  ...
end
```

If we want the derivative of `y` with respect to variable to `x`, the code is appropriate as is. But, if we want the derivatives of

- `y` with respect to variable `a`,
- `b` with respect to variable `x`, or
- `b` with respect to variable `a`,

we run into a problem. Specifically, we cannot nominate a local variable of subroutine `foo` as dependent or independent, since it is not visible outside of `foo`. In order to avoid this problem, we make all 'interesting' variables in subroutine `foo` visible through parameter passing or common blocks. For example, program `main` could be rearranged to:

```
program main
  ...
foo(x,y,a,b)
  ...
end

subroutine foo(x,y,a,b)
a = x+1
y = x*x
b = x/2
  ...
end
```

An alternative to this workaround is the buddy system discussed in Section 5.1.

- **SAVEing data**

  For the purpose of automatic differentiation, it is somewhat difficult to define the proper semantics and handling of SAVE statements or data initialized by DATA statements. ADIFOR's handling of these constructs may be reasonable for your problem, but it is probably wise at this stage to avoid these constructs. Be vary wary of routines that contain SAVE statements that ADIFOR decides to "clone" into multiple copies for use in different derivative calling contexts.

- **Multiple entry points**

  ADIFOR does not support multiple entry points. For example, the following program will not work in ADIFOR,

```
program nogood
...
call foo1(a,b)
```

```
...
call foo2(x,y)
...
end
```

where the subroutine `foo1` is defined as follows:

```
subroutine foo1(a,b)
...
entry foo2(k,l)
...
end
```

- **Complex variables**

  Currently, ADIFOR does not support complex variables or functions.

- **Procedure parameters**

  ADIFOR does not support procedure parameters. This restriction can often be circumvented by replacing the procedure parameter with an integer parameter, and modifying the code to call the appropriate procedure based on the value of this integer.

- **Reserved variable names**

  ADIFOR uses the variables `g$p$`, `g$pmax$`, and `g$i$` in the generated code. While it is acceptable to use these variable names in a subroutine calling the ADIFOR-generated code, they should not be used in a program to be processed by ADIFOR.

- **Externals that are not referenced**

  External declarations present in the original code, which are no longer referenced in the code generated by ADIFOR, might create an error of 'unsatisfied external references' in some compilers. For example, the Silicon Graphics compiler produces an error message about an undefined entry "_second_" on the following code.

```
program foo
real f,g$f(5)
integer i
external func

call g$func(5,f,g$f,5)
write(*,*) f, (g$f(i), i=1,5)
end

subroutine g$func(g$p$,f,g$f,ldg$f)
real f,g$f(ldg$f)
integer g$i$,g$p$

f = 5.0
do g$i$ = 1, g$p$
   g$f(g$i$) = 0.0d0
enddo
return
end
```

  To fix this problem, the user should take out external declarations for variables no longer used. This may be done automatically using the NAG declaration tool `nag_decs` described in Section 8.3.

- **Dprod()**

  The current version of ADIFOR does not support the Fortran intrinsic function, `dprod()`. This is the result of an oversight, and it is anticipated that future versions of ADIFOR will support `dprod()`.

- **Hollerith constants**

  ADIFOR does not currently understand Hollerith constants as part of format declarations. The NAG Fortran polisher can be used to convert Hollerith constants to normal strings and move all `FORMAT` statements to the end of the file. To invoke the polisher, use the command:

  ```
  nag_polish -po /usr/local/adifor/misc/polish.opt <fortran file>
  ```

  Information on the NAG polisher can be found using `man nag_polish` and `man nag_polopt`.

- **Columns beyond 72**

  ADIFOR does not ignore columns beyond number 72. If a file contains line numbers or other unnecessary information in columns 73 and above, this text should be removed using the command

  ```
  cat <filename> | colrm 73.
  ```

- **Statement functions**

  Currently, ADIFOR does not support this Fortran 77 feature. For example, the following code is not accepted by ADIFOR:

  ```
          REAL A,B,C
  c the following statement function computes the discriminant of a quadratic equation
          DISCR(A,B,C) = B ** 2 - 4 * A * C
          ...
  ```

  The user should use a subroutine call instead, as follows,

  ```
          REAL A,B,C,V
  ...
          CALL DISCR(A,B,C,V)
          ...
  ```

  where the subroutine `DISC(A,B,C,V)` is defined as follows:

  ```
  c the following subroutine computes the descriminant of a quadratic equation
          SUBROUTINE DISC(A,B,C,V)
             REAL A,B,C,V

             V = B ** 2 - 4 * A * C
          end
  ```

- **Multiple returns feature**

  The current version of ADIFOR does not support the multiple returns feature of Fortran. For example, the following program will not work in ADIFOR.

  ```
          program nogood
          ...
   30     ... code ...
          ...
          CALL foo(X,Y,*30,*50)
   50     ... code ...
          end

          subroutine foo(A,B,*,*)
          ...
          if( This ) return 1
          if( That ) return 2
          ...
          return
          end
  ```

- **No expressions in argument list**

  Currently, ADIFOR does not support the use of expressions as arguments of a procedure call. As a workaround, these expressions should be assigned to a temporary variable, which is used as the argument. Thus,

  ```
  call foo(active+5.0)
  ```

  should be rewritten as

  ```
  temp = active+5.0
  call foo(temp)
  ```

  which is processed correctly.

  Constants or expressions passed as arguments, which correspond to parameters that may be modified, are unsafe references (see [8]) and flagged by `nag_pfort` as such. They are not reliably portable and hence, there may be some compilers where their use leads to unexpected results. So replacing the constants by temporaries as shown above (hence making the code suitable for ADIFOR), also increases the portability of the original code.

- **Include Files**

  Currently, ADIFOR does not support include files. The user is advised to first use the C preprocessor `/usr/lib/cpp` to generate Fortran-77 files from files containing include statements. See the man pages for `cpp` for more details.

4 Mathematical Pitfalls

There are some operations which do not have any (or, at least not the expected) mathematical meaning with respect to differentiation. Among these are:

- **Derivatives of integers and characters**

  The derivative of an integer or character is meaningless. As a consequence, if an integer is assigned a value from an *active variable* (that is, a variable that either is a dependent or independent variable, or a variable whose value depends on that of an independent variable *and* whose value is used to compute a dependent variable), the integer does not become active. Thus, the gradient objects of any variables that depend on these integers may not have the expected values. The same holds true for characters.

- **Reals equivalenced to double precisions**

  The process of converting an array of `real` variables into an array of `double precision` variables using the `equivalence` statement has no real mathematical meaning. Thus, if a program performs this operation, the `double precision` gradient objects, and any gradient objects which depend on them, will be meaningless. The same holds true if an array of `double precision` variables is `equivalenced` to an array of `real` variables. Note that this is a *very unportable* programming practice anyway, since its results depend heavily on the floating-point representation.

- **Introducing points of nondifferentiability**

  It is sometimes the case that, for the sake of improving efficiency, a program tests the value of a variable to see if a function is being evaluated at a special point in space, then computes the value of the function based on that knowledge. For example, the following piece of code computes $y = x^4$.

```
if ((x .eq. 0.0d0) .or. (x .eq. 1.0d0)) then
   y = x
else
   t = x*x
   y = t*t
endif
```

If automatic differentiation is used to compute $\frac{dy}{dx}$, then the value of $\frac{dy}{dx}|_{x=0}$ will be 1 (because the statement $y = x$ implies that $\frac{dy}{dx} = \frac{dx}{dx} = 1$) rather than the expected 0. Similarly, the value of $\frac{dy}{dx}|_{x=1}$ will be 1 rather than 4. This "anomaly" stems from the fact that automatic differentiation differentiates the statements executed in the course of program execution. This issue, as well as other subtle pitfalls, are discussed in [6], which can be found in `/usr/local/adifor/doc` (see Section 8).

- **Nondifferentiable functions**

  Some Fortran intrinsic functions, such as `sqrt()`, `abs()`, and `max()`, are not differentiable at all points. Special subroutines have been implemented to handle these exceptions. The subroutines are located in the file `intrinsic.f` (which can be found in `/usr/local/adifor/lib` — see Section 8) and this file should be included in the compilation of ADIFOR-generated code. For more details on exception handling and the supporting subroutines, see [4], which can be found in `/usr/local/adifor/doc` (see Section 8).

5 Workarounds and Shortcuts

5.1 Input and Output

ADIFOR currently just echoes I/O statements like `READ` and `WRITE`. This creates problems, in particular with `READ`s, if the variable read is a so-called *active* variable, that is, a variable that either is a dependent or independent variable, or a variable whose value depends on that of an independent variable *and* whose value is used to compute a dependent variable.

Normally, ADIFOR assumes that

- independent variables are passed into the top-level routine and dependent variables are passed out, and
- independent variables are initialized outside of the top-level routine and their values used after the top-level routine has completed.

"Passing" is either via subroutine parameters or common blocks. So, the normal ADIFOR interface cannot compute derivatives

- with respect to variables that are initialized by a `READ` statement,
- for the various values of a variable whose intermediate values are written out, and
- for a variable that is declared locally in the top-level routine, i.e. is not visible outside the top-level routine.

This section describes some workarounds for these situations.

- **READ :**

  Suppose we would like to compute the derivatives with respect to a variable `x` that is read in as in the following subroutine

```
      subroutine readlcl(y)
      real y, x(10)
      integer i
      y = 1.0
      read(*,*) (x(i),i=1,10)
      do 10 i = 1,10
         y = y * x(i)
10    continue
      return
      end
```

The normal ADIFOR interface does not allow this, and processing this subroutine with ADIFOR produces the error message

```
ERROR: INDEPENDENT variable x is not in COMMON and
       is not a formal parameter of readlcl.
```

A workaround is to allocate a variable of the same dimensions as `x`, the so-called "buddy" that is initialized to 0, is passed into `readlcl`, and whose corresponding derivative object is seeded properly for the computation of the derivatives of `x`. So we could modify `readlcl` to `breadlcl` (`xbuddy` is `x`'s buddy):

```
      subroutine breadlcl(xbuddy,y)
      real y, x(10), xbuddy(10)
      integer i
      y = 1.0
c
c     zero out x to achieve g$x = 0
c
      do 30 i = 1,10
         x(i) = 0.0
30    continue

      read(*,*) (x(I),I=1,10)
c
c     adding of buddy here will result in correct seeding
c     of g$x
c
      do 20 i = 1,10
         x(i) = x(i) + xbuddy(i)
20    continue
      do 10 i = 1,10
         y = y * x(i)
10    continue
      return
      end
```

Having specified `xbuddy` as an independent variable, ADIFOR then produces[†]

```
          subroutine g$breadlcl$3(g$p$, xbuddy, g$xbuddy, ldg$xbuddy, y, g$y
     *, ldg$y)
C
C       Formal y is active.
C       Formal xbuddy is active.
C
        integer g$p$
```

[†]Some declaration lines were merged and some unused `continue` statements removed to reduce the length of the listing.

```
        integer g$pmax$
        parameter (g$pmax$ = 10)
        integer g$i$
        real r$1
        integer ldg$y
C
        integer i
        real y, g$y(ldg$y)
        real x(10), xbuddy(10),g$x(g$pmax$, 10), g$xbuddy(ldg$xbuddy, 10)
        integer ldg$xbuddy
        if (g$p$ .gt. g$pmax$) then
          print *, 'Parameter g$p is greater than g$pmax.'
          stop
        endif
        y = 1.0
        do 99992 g$i$ = 1, g$p$
          g$y(g$i$) = 0.0
99992   continue
C
C       zero out x to achieve g$x = 0
C
        do 99999, i = 1, 10
          x(i) = 0.0
          do 99991 g$i$ = 1, g$p$
            g$x(g$i$, i) = 0.0
99991     continue
99999   continue

        read (*, *) (x(i), i = 1, 10)
C
C       adding of buddy here will result in correct seeding
C       of g$x
C
        do 99998, i = 1, 10
C         x(i) = x(i) + xbuddy(i)
          do 99990 g$i$ = 1, g$p$
            g$x(g$i$, i) = g$x(g$i$, i) + g$xbuddy(g$i$, i)
99990     continue
          x(i) = x(i) + xbuddy(i)
99998   continue
        do 99997, i = 1, 10
C         y = y * x(i)
          r$1 = x(i)
          do 99989 g$i$ = 1, g$p$
            g$y(g$i$) = r$1 * g$y(g$i$) + y * g$x(g$i$, i)
99989     continue
          y = y * r$1
99997   continue
        return
      end
```

Assuming that we wanted to compute the gradient of `y` with respect to `x`, we would initialize `xbuddy` to all zeros in the main program, and `g$xbuddy` to the identity matrix. Zeroing out `x` and adding `xbuddy` to `x` has the effect that `x` gets the desired value, and `g$x` is assigned the correct seed matrix value. We note that, instead of passing `xbuddy` as an argument, we could have allocated it in a common block.

When the desired independent variable `x` is in a common block instead of a local variable, we have an-

other possibility in addition to the buddy system. Suppose we have the subroutine `readcmn` defined as

```
      subroutine readcmn(y)
      real y
      integer i

      real x(10)
      common /cx/ x

      y = 1.0
      read(*,*) (x(I),I=1,10)
      do 10 i = 1,10
         y = y * x(i)
10    continue
      return
      end
```

and we would like to compute the derivatives of `y` with respect to `x` at the value of `x` that is read in. When we specify `x` as an independent and `y` as a dependent variable, ADIFOR produces

```
          subroutine g$readcmn$3(g$p$, y, g$y, ldg$y)
C
C       Common block /cx/ contains active variables.
C       Variable x in Common block /cx/ is active.
C       Formal y is active.
C
        integer g$p$
        integer g$pmax$
        parameter (g$pmax$ = 10)
        integer g$i$
        real r$1
        integer ldg$y
C
        integer i
        real y
        real g$y(ldg$y)
        real x(10)
        real g$x(g$pmax$, 10)
        common /cx/ x
        common /g$cx/ g$x
        if (g$p$ .gt. g$pmax$) then
          print *, 'Parameter g$p is greater than g$pmax.'
          stop
        endif
        y = 1.0
        do 99996 g$i$ = 1, g$p$
          g$y(g$i$) = 0.0
99996   continue
        read (*, *) (x(i), i = 1, 10)
        do 99999, i = 1, 10
C         y = y * x(i)
          r$1 = x(i)
          do 99995 g$i$ = 1, g$p$
            g$y(g$i$) = r$1 * g$y(g$i$) + y * g$x(g$i$, i)
99995     continue
          y = y * r$1
10        continue
99999   continue
        return
```

```
      end
```

As the `READ` statement is simply echoed, the seed matrix `g$x` is not initialized when `x` is read in. To achieve this, we can either use the buddy system described above, or perform the initialization

- in the program calling `g$readcmn$3` (assuming that `x` was not used before the `READ` statement as is the case in our example), or
- by adding the initialization of the seed matrix after the `READ` statement in the ADIFOR-generated code.

The latter approach of course requires performing this modification whenever ADIFOR is rerun.

- **WRITE :**

Suppose we would like to compute derivative with respect to several values that are assigned to the same variable. The normal ADIFOR interface only retrieves the derivatives with respect to the last value. The solution is to nominate some buddies as dependent variables, and have them preserve the derivatives of the values of interest. For example, if we have

```
      subroutine writelcl(x)
      real y, x(10)
      integer i

      y = 1.0
      do 10 i = 1,10
         y = y * x(i)
10    continue
      write(*,*) y

      y = 0.0
      do 20 i = 1,10
         y = y + x(i)
20    continue
      write(*,*) y

      return
      end
```

we allocate two buddies `ybuddy1` and `ybuddy2` to which we assign the two values of `y` that are computed in the course of execution of the subroutine. The resulting subroutine is (assuming we allocate the buddies in a common block)

```
      subroutine bwritelcl(x)
      real y, x(10)
      integer i

      real ybuddy1, ybuddy2
      common /YB/ ybuddy1, ybuddy2

      y = 1.0
      do 10 i = 1,10
         y = y * x(i)
10    continue
      write(*,*) y
c
c      save value of y and its derivs
c
      ybuddy1 = y

      y = 0.0
      do 20 i = 1,10
         y = y + x(i)
20    continue
c
c     save value of y and its derivs
c
      ybuddy2 = y

      write(*,*) y

      return
      end
```

and, having specified `x` as an independent variable and `ybuddy1` and `ybuddy2` as dependent variables, ADIFOR produces

```
          subroutine g$bwritelcl$3(g$p$, x, g$x, ldg$x)
C
C       Common block /yb/ contains active variables.
C       Variable ybuddy2 in Common block /yb/ is active.
C       Variable ybuddy1 in Common block /yb/ is active.
C       Formal x is active.
C
        integer g$p$
        integer g$pmax$
        parameter (g$pmax$ = 10)
        integer g$i$
        real r$1
        real g$ybuddy2(g$pmax$)
        real g$ybuddy1(g$pmax$)
C
        integer i
        real y
        real g$y(g$pmax$)
        real x(10)
        real g$x(ldg$x, 10)
        integer ldg$x

        real ybuddy1, ybuddy2
        common /yb/ ybuddy1, ybuddy2
```

```
        common /g$yb/ g$ybuddy1, g$ybuddy2
        if (g$p$ .gt. g$pmax$) then
          print *, 'Parameter g$p is greater than g$pmax.'
          stop
        endif
        y = 1.0
        do 99991 g$i$ = 1, g$p$
          g$y(g$i$) = 0.0
99991   continue
        do 99999, i = 1, 10
C         y = y * x(i)
          r$1 = x(i)
          do 99990 g$i$ = 1, g$p$
            g$y(g$i$) = r$1 * g$y(g$i$) + y * g$x(g$i$, i)
99990     continue
          y = y * r$1
10        continue
99999   continue
        write (*, *) y
C
C       save value of y and its derivs
C
        ybuddy1 = y
        do 99989 g$i$ = 1, g$p$
          g$ybuddy1(g$i$) = g$y(g$i$)
99989   continue

        y = 0.0
        do 99988 g$i$ = 1, g$p$
          g$y(g$i$) = 0.0
99988   continue
        do 99998, i = 1, 10
C         y = y + x(i)
          do 99987 g$i$ = 1, g$p$
            g$y(g$i$) = g$y(g$i$) + g$x(g$i$, i)
99987     continue
          y = y + x(i)
20        continue
99998   continue
C
C       save value of y and its derivs
C
        ybuddy2 = y
        do 99986 g$i$ = 1, g$p$
          g$ybuddy2(g$i$) = g$y(g$i$)
99986   continue

        write (*, *) y

        return
      end
```

On exit from `g$bwritelcl$3`, `g$ybuddy1` contains the derivative of the value of `y` at the assignment `ybuddy1 = y`, and `g$buddy2` contains the derivative of the value of `y` at the assignment `ybuddy2 = y`.

- **Variables not visible:**

  Occasionally, we would like to treat variables local to a particular subroutine as independent or dependent

variables. As was discussed in Section 3, one solution is to change the variables so that they are parameters passed into the subroutine, rather than local to the subroutine. However, this is not always an ideal solution. In such cases, using the "buddy system" may be more appropriate. Suppose, for example, that we have some local variable `h` with respect to which we would like to determine sensitivities in the following section of code:

```
subroutine planck(e,lambda)
real e,lambda
real h

h=6.625e-27

e = h * lambda
return
end
```

Then, we can use the buddy system to find sensitivities with respect to `h`. At the level of the call to the top-level subroutine, we add:

```
common /BUDDES/ HBUDDY
common /g$BUDDES/ g$HBUDDY(pmax)

HBUDDY = 0.0
g$h = all zeroes
g$HBUDDY = whatever seed you need
```

In the top-level subroutine `planck`, we add an allocation and assignment to yield:

```
subroutine planck(e,lambda)
real e,lambda
real h
common /BUDDES/ HBUDDY
real hbuddy

h=6.625e-27
h = h + hbuddy

e = h * lambda
return
end
```

Therefore, in the augmented code we will have `g$h=g$h+g$hbuddy`. The local variable `h` becomes active, and the initial value of its gradient object `g$h` is equal to the seed with which the buddy gradient object `g$hbuddy` is initialized. Note that the value of `h` is not affected, since `hbuddy` has a value of zero.

5.2 ADPRE

A preprocessor, called ADPRE, is available for rewriting functions and statement functions in a form that ADIFOR recognizes and extracting expressions used as arguments to subroutine calls, saving the results in temporary variables which are passed instead. ADPRE can be used by setting the `RN_HOME` environment variable using the command

```
setenv RN_HOME /anything
```

and running the preprocessor by issuing the command

```
adpre -P <composition file>,
```

where `<composition file>` is the composition file used for invoking ADIFOR. *ADPRE is an experimental piece of software and thus is not robust nor can it be expected to always behave as desired.* One flaw in the preprocessor is that the variable names generated may not be unique. Duplicated variable names can be detected automatically using the NAG declaration tool `nag_decs` described in Section 8.3.

5.3 A Generic Makefile

A generic makefile, called `make.adifor`, is available to make the process of generating code with ADIFOR and using that code simpler. This makefile can invoke the NAG portability verifier, create the composition file, run ADIFOR, and more. It is included in Appendix A for reference and can be found in the `/usr/local/adifor/utils` directory (see Section 8). The file may be copied to the same directory as the code to be processed and modified to fit that code. Only three variables in the file need to be changed:

AD_TOPLEVEL: The name of the top-level subroutine. There should be a script file with the name $(AD_TOPLEVEL).adf, as described in Step 2 of Section 2.

ADIFOR_INPUT: The names of all of the files that make up the program composition, except the main program.

ADIFOR_MAIN: The name of the file containing the main program.

Once the makefile has been modified appropriately, the user can invoke the make facility using `make -f make.adifor <argument>`, where `<argument>` is one of:

help: To get a list of available options.

portability: To run the NAG portability verifier.

calltree: To generate a calltree using the NAG utility.

declare: To run the NAG declarations tool.

adifor: To process the code using ADIFOR.

compile: To compile each of the files created by ADIFOR.

clean: To remove all of the temporary files introduced by ADIFOR.

6 Advanced Topics

6.1 Efficient Common Block Organization

Fortran allows common blocks to be accessed differently in different modules. For example, a common block declared as

```
common /block/ A(10,10),B(100)
```

in one subroutine might be declared as

```
common /block/ X(4,5,10)
```

in another. We call this feature "common block reshaping." In order to have reshaping work for common blocks containing active variables, a derivative object is created for every entry in a common block, even if only one member

of the common block really needs derivative objects. Using the above example, if only B was active, we would still declare

```
common /g$block/ g$A(g$pmax$,10,10),g$B(g$pmax$,100)
```

and g$A would not be used. In this fashion, derivative objects of B would be correctly identified with derivative objects for X, as implied by the original reshaping. Note that reshaping breaks down when common blocks contain entries of mixed type (in particular integer and character), but in these circumstances reshaping is a dangerously nonportable practice anyway, and ADIFOR issues a warning.

To eliminate the unnecessary storage allocation, the user is advised to organize the program's global variables so that active and non-active variables are put in separate common blocks. It is beneficial to separate variables of type integer and character because these variables cannot be active. ADIFOR-generated code lists which variables are truly active. Hence, after the first ADIFOR run, detailed activity information is available.

As an example, the following code:

```
...
double Xactive,Wactive,Zactive,Ynotactive
integer I,J
common /blockAll/ Xnotactive,Wactive,Zactive,Ynotactive,I,J
...
```

could be rearranged to:

```
...
double Xactive,Wactive,Zactive,Ynotactive
integer I,J
common /blockActive/ Wactive,Zactive
common /blocknotActive/ Xnotactive,Ynotactive
common /blockNeverActive/ I,J
...
```

so that storage for gradient objects is allocated only for active variables.

6.2 ADMakefile

The makefile generated by ADIFOR, ADMakefile, contains two variables whose values may be set to control the operation of the translator, ADMISC and ADFLAVORS. ADFLAVORS may be set to OPT, UNOPT, or both. The UNOPT flag indicates that ADTRANS should create unoptimized versions of the ADIFOR-generated code, which may be useful for debugging purposes. This code may contain multiplications by 1.0 and other superfluous operations, and will be stored in files with the extension .unopt.f. The OPT flag indicates that ADTRANS should create optimized versions of the code, where arithmetic operations involving the identity have been removed. At present, the only available options for ADMISC is SAXPY, an experimental device for supporting sparse derivative objects. For now, this variable should be left blank.

7 An Advanced Example

As a more complex example of the process of generating derivative code using ADIFOR, consider again the subroutine in Figure 5. We may use the makefile in Appendix A to simplify our task.

```
      SUBROUTINE SSINC(NELEM,NNOD,NTOTAL,NCLASS,NKIND,X,NPART,CPART,
     +                 THICK,TNSDTA,MATL,ASTRPY,FLIMIT,IRATE,XMUP,XMUD,
     +                 BF,NP,TMAX,YMAX,ISYM,IBEAD,BEADUF,NSTOP,EPSDEV,
     +                 ZPCHDP,FAILDP,IFAIL,ZSTRAN,ZSHAPE,T,KIND,S,ECUM,
     +                 ZSTRSS,KNDNOD,ISUCC,IELPEK,ZPKSTR,ZDRAW,NREJCT,
     +                 HOLRAD,ZCRCST,XMOVE,IFLSTP)


C     .. Scalar Arguments ..
      DOUBLE PRECISION BEADUF,EPSDEV,FAILDP,HOLRAD,T,THICK,TMAX,XMOVE,
     +                 XMUD,XMUP,YMAX
      INTEGER IBEAD,IFAIL,IFLSTP,IRATE,ISYM,MATL,NCLASS,NELEM,NNOD,NP,
     +        NPART,NREJCT,NSTOP,NTOTAL
C     ..
C     .. Array Arguments ..
      DOUBLE PRECISION ASTRPY(5),BF(2),CPART(8,1000),ECUM(1000,2),
     +                 FLIMIT(5),S(1000,2,2),TNSDTA(10),X(1000,2)
      REAL ZCRCST(0:30),ZDRAW(0:30,2),ZPCHDP(0:30),ZPKSTR(0:30),
     +     ZSHAPE(0:30,1000,2),ZSTRAN(0:30,1000,4),ZSTRSS(0:30,1000,2)
      INTEGER IELPEK(0:30),ISUCC(0:30),KIND(1000),KNDNOD(0:30,1000),
     +        NKIND(2)
C     ..
C     .. Arrays in Common ..
      DOUBLE PRECISION ENGST(1000,4)
C     ..
C     .. Common blocks ..
      COMMON /CENGST/ENGST
C     ..

C     ... code for the function evaluation

      RETURN
      END
```

Figure 5. Sample Top-level Subroutine

7.1 Preliminaries

The use of the makefile in Appendix A still requires that we create a dummy main program, make sure the active subtree conforms to the Fortran standard and other restrictions, and create a script file, called `ssinc.adf`.

The dummy main program

```
program dummy
end
```

explained in Section 2, Step 2 will be used. We shall assume that the dummy main program is in the file `dummy.f`. Next, a thorough analysis of the active subtree should be performed to insure that it does not violate any of the restrictions described in Section 3.

7.2 Creating the ADIFOR script file

To proceed, we create an ADIFOR script file. Suppose we wish to have ADIFOR generate code for the derivatives of the variable `engst(i,1), i=1, nelem` with respect to the variables `tnsdta(3), astrpy(1), xmup, xmud,` and `bf(1:2)`. Then, we generate the script file:

```
TOP ssinc
PMAX 19
IVARS tnsdta,astrpy,xmud,xmup,bf
OVARS engst
SEP $
```

Recall that the value `PMAX` corresponds to the maximum length of gradient objects, which may be as much as $10+5+1+1+2 = 19$. Since we are only concerned with certain elements of the variables nominated as independent, we could use a value as small as $1+1+1+1+2 = 6$ for `PMAX`. If we are certain that we will never consider the derivatives with respect to the other elements, then this value (6) should be used for `PMAX`. Otherwise, the conservative value of 19 should be used. We have chosen the conservative approach for this example.

7.3 Running ADIFOR

To proceed, we run ADIFOR using the command `make -f make.adifor adifor`. If the script file has been created properly, all restrictions have been met, and the variables in the makefile have been properly defined, ADIFOR will generate several new subroutines. Only the top-level subroutine, `g$ssinc$4001ca00`, needs to be called from the user's code. All other subroutines generated by ADIFOR are called, directly or indirectly, by this subroutine.

7.4 Seed Matrix Initialization

The subject of seed matrix initialization was already addressed in Step 4 of Section 1. However, it may not be clear what should be done in the case where we have multiple independent variables. As an example of this situation, refer to the top-level subroutine in Figure 5. We have run ADIFOR, declaring `tnsdata, astrpy, xmup, xmud,` and `bf` as independent variables and `engst` as the dependent variable, so the generated code is capable of computing

$$J = J_{engst} = \left(\frac{\partial\, engst}{\partial\, tnsdata(1:10)}, \frac{\partial\, engst}{\partial\, astrpy(1:5)}, \frac{\partial\, engst}{\partial\, xmup}, \frac{\partial\, engst}{\partial\, xmud}, \frac{\partial\, engst}{\partial\, bf(1:2)} \right),$$

which is a 4000×19 matrix.

However, we are only concerned with the derivatives of `engst(i,1), i = 1,1000` with respect to `tnsdata(3), astrpy(1), xmup, xmud,` and `bf(1:2)`. The total number of independent variables is therefore $1+1+1+1+2=6$. Each gradient object corresponds to the derivatives of some variable with respect to the independent variables under consideration. As a consequence, each gradient object should have 6 rows. So, since `tnsdata` is of dimension 10, `g$tnsdata` is of dimension 6×10. Similarly, `g$astrpy` is of dimension 6×5, `g$xmup` is of dimension 6×1, `g$xmud` is of dimension 6×1, and `g$bf` is of dimension 6×2. Based on the independent variables with which we are concerned, these gradient objects should be initialized as follows:

$$\texttt{g\$tnsdata} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad \texttt{g\$astrpy} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$\texttt{g\$xmup} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad \texttt{g\$xmud} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad \text{and } \texttt{g\$bf} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The rows of `g$<IndVar>` represent the first, second, ..., and sixth independent variables considered in the problem. The columns of `g$<IndVar>` entries represent the elements in the original data object. For example, in `g$tnsdata`, we initialized to 1.0 the derivative of the third entry (3rd col.) of `tnsdata` with respect to the first independent variable (1st row), that is with respect to itself. Together, the five matrices constitute the seed matrix. Thus,

$$\texttt{g\$engst} = \left(J \times \begin{pmatrix} \texttt{g\$tnsdata}^T \\ \texttt{g\$astrpy}^T \\ \texttt{g\$xmup}^T \\ \texttt{g\$xmud}^T \\ \texttt{g\$bf}^T \end{pmatrix} \right)^T,$$

a 6×1000 element matrix. In particular,

- `g$engst(1,*, 1)` contains the derivatives of `engst(*,1)` with respect to `tnsdata(3)`,
- `g$engst(2,*, 1)` contains the derivatives of `engst(*,1)` with respect to `astrpy(1)`,
- `g$engst(3,*, 1)` contains the derivatives of `engst(*,1)` with respect to `xmup`,
- `g$engst(4,*, 1)` contains the derivatives of `engst(*,1)` with respect to `xmud`, and
- `g$engst(5:6,*, 1)` contains the derivatives of `engst(*,1)` with respect to `bf(1:2)` .

Other examples of seed matrix initialization can be found in [5].

7.5 Incorporating the ADIFOR-generated subroutine

Following the procedure outlined in Section 2, the gradient objects g$engst, g$tnsdta, g$astrpy, g$xmup, g$xmud, and g$bf should be declared and initialized, and a call to g$ssinc$4001ca00 added at the point where the derivative values are required. So, the calling program should look something like:

```
      PROGRAM MAIN
      ...
c
c      subroutine dzero2(m,n,a,lda) initializes a matrix to 0.0 .
c      subroutine dzero1(m,a) intializes a matrix to 0.0 .
c
c      allocate storage for derivative objects
c      ***************************************
c
c      number of independent variables is 6
c
      integer g$p$par, g$p$common
      parameter (g$p$par = 6, g$p$common = 19)

c allocate derivative objects

        double precision g$t(g$p$par), g$xmud(g$p$par), g$xmup(g$p$par),
     +     g$astrpy(g$p$par,5), g$bf(g$p$par,2),
     +     g$ecum(g$p$par,1000,2), g$s(g$p$par,1000,2,2),
     +     g$tnsdta(g$p$par,10)

        double precision g$engst(g$p$common, 1000, 4)
        common /g$cengst/ g$engst
      ....
c
c     initialize seed matrix
c     **********************
c**********************************************************************
c*** Each of the independent variables corresponds to a different ****
c*** column of the seed matrix.                                   ****
c**********************************************************************
c
c     for tnsdata(3) for column 1
c
      call dzero2(g$p$,10,g$tnsdta,g$p$)
      g$tnsdta(1,3) = 1.0
c
c     for astrpy(1) for column 2
c
      call dzero2(g$p$,5,g$astrpy,g$p$)
      g$astrpy(2,1) = 1.0
c
c     for xmup for column 3
c
      call dzero1(g$p$,g$xmup)
      g$xmup(3) = 1.0
c
c     for xmud for column 4
c
      call dzero1(g$p$,g$xmud)
      g$xmud(4) = 1.0
```

```
c
c     for bf for column 5 and 6
c
      call dzero2(g$p$,2,g$bf,g$p$)
      g$bf(5,1) = 1.0
      g$bf(6,2) = 1.0

c     call ADIFORed version of ssinc
c     ****************************
c
      CALL g$ssinc$4001ca00(g$p$,
     +     NELEM,NNOD,NTOTAL,NCLASS,NKIND,X,NPART,CPART,THICK,
     +     TNSDTA,g$tnsdta,g$p$,
     +     MATL,
     +     ASTRPY,g$astrpy,g$p$,
     +     FLIMIT,IRATE,
     +     XMUP,g$xmup,g$p$,XMUD,g$xmud,g$p$,BF,g$bf,g$p$,
     +     NOUT,TMAX,YMAX,ISYM,IBEAD,BEADUF,NSTOP,EPSDEV,ZPCHDP,FAILDP,
     +     IFAIL,ZSTRAN,ZSHAPE,
     +     T,g$t,g$p$,
     +     KIND,
     +     S,g$s,g$p$,ECUM,g$ecum,g$p$,
     +     ZSTRSS,KNDNOD,ISUCC,
     +     IELPEK,ZPKSTR,ZDRAW,NREJCT,HOLRAD,ZCRCST,XMOVE,IFLSTP)
c
c      Rows 1,6 of g$engst(:.:,1) correspond to the derivative of the variable
c      engst(i,1), i=1, nelem with respect to the variables tnsdata(3), astrpy(1),
c      xmup, xmud, bf(1:2), respectively.
c
       ...
      STOP
      END
```

Note that the allocations differ for derivative objects in common blocks and those that get passed into `g$ssinc$4001ca00` as subroutine parameters. **For variables in comon blocks, it is necessary to use the same leading dimension as was specified as `PMAX` to ADIFOR** (19, in this case), since otherwise the calling program and the ADIFOR-generated routine use inconsistent declarations. For variables that are passed in as parameters, we may allocate more space than is required, since the leading dimensions of these gradient objects are passed to the ADIFOR-generated derivative code.

8 Using ADIFOR at Argonne

At present, we do not encourage ADIFOR distribution, since the system is likely to undergo many changes. Instead, we suggest the following procedure:

1. Assemble the program composition on your system and ftp the Fortran source to Argonne.
2. Run ADIFOR at Argonne to generate derivative code.
3. Ftp ADIFOR-generated code back to your system, and incorporate it into your application.

8.1 Organization

To use ADIFOR, you need to get an account on our Suns (if you do not have one, send mail to `adifor-request@mcs.anl.gov`).

When you log into Argonne to use ADIFOR, you should log into

cosmo.mcs.anl.gov (Internet no. 140.221.10.10).

Cosmo is a multi-processor Sparc-compatible Solbourne workstation with 256 MBytes of real memory and plenty of swapspace. It is well suited for running memory-intensive jobs such as ADIFOR. As an alternative, you can use

canopus.mcs.anl.gov (Internet no. 140.221.3.131)
dude.mcs.anl.gov (Internet no. 140.221.1.12)

Please use canopus and dude only when cosmo is unavailable, as those are personal workstations, not general computing resources. Since disk space is tight, we ask you to *keep your account at Argonne as cleaned up as possible.*

The ADIFOR binaries as well as library files, documentation, and other useful information can be found in subdirectories of the `/usr/local/adifor` directory, according to the following organization:

- **bin:** Contains the ADIFOR executables. Make sure that `/usr/local/adifor/bin` is in your Unix search path.
- **doc:** Contains a README file plus postscript versions of ADIFOR working notes and other relevant papers. In particular, this directory will contain the up-to-date version of this manual.
- **examples:** Contains some examples of programs processed with ADIFOR. The examples may require files in the **lib** and **utils** subdirectories in order to compile properly.
- **lib:** Contains files that in general are necessary for the compilation of ADIFOR-generated code. In particular, the file `intrinsic.f` used for exception handling can be found here. Any programs which uses any of the Fortran intrinsic functions should have `intrinsic.f` compiled with the ADIFOR version.
- **man:** Contains the man page for ADIFOR.
- **utils:** Contains subroutines that may prove useful in incorporating ADIFOR-generated code in a program.

8.2 Support

Limited support is available via email. Any potential bugs should be reported to `adifor-bugs@mcs.anl.gov`. Any questions or comments on the functionality of ADIFOR should be directed to `adifor@mcs.anl.gov`. Requests for accounts or other administrative issues should be addressed to `adifor-request@mcs.anl.gov`.

8.3 NAG Tools

The NagTools Fortran utilities are available at Argonne for help in satisfying the requirement that compositions processed by ADIFOR adhere to the Fortran standard. These tools can check whether a program conforms to the ANSI standard, make sure that every variable is declared explicitly, and generate a calling tree. To run these utilities, the user needs to add `/usr/local/NAGWare_f77_tools/scripts` in the Unix search path. Here is a short description of some of the NAGWare utilities:

- **nag_pfort :** is the Fortran 77 NAGWare portability verifier. Nag_pfort checks for
  - conformance with the ANSI Fortran 77 standard,
  - conformance with a portable subset of the ANSI Fortran 77 standard, and
  - correct inter-program-unit communication.
- **nag_fcalls :** prints the call graph of a Fortran 77 program.

- **nag_decs :** is the NAGWare Fortran 77 declaration standardiser. This tool rebuilds the declarative part of a Fortran 77 program unit declaring all names and also pretty prints.

More details can be found in [8] and in the man pages for the individual tools (the manual pages are located in `/usr/local/NAGWare_f77_tools/manl`).

Your site may provide similar tools, such as `forchek`. Check with your system administrator.

9 Common Problems

There are several problems which users have encountered while trying to process programs with ADIFOR. We provide a brief explanation of each and possible solutions.

- **ADIFOR reports that a file can not be found**

  Assuming the file in question is in fact present, the most probable reason for this error message is trailing blanks in the composition file. ADIFOR interprets trailing blanks as part of the filename, and therefore is unable to find the file. See Section 2, Step 1 for more details on the composition file.

- **Unneeded labels and `CONTINUE` statements appear in the ADIFOR-generated subroutines**

  In addition to creating new labels and `CONTINUE` statements, ADIFOR preserves those present in the original programs. There are two reasons for this functionality. The first reason is to insure that any references to these labels (by a computed `GOTO`, for example) in the original program remain properly defined. Labels are also preserved to facilitate cross-referencing between the original and ADIFOR-generated code. If a certain algorithm is present near a particular label in the original program, it will be at the same location in the ADIFOR-generated code.

- **ADIFOR runs out of string space**

  This may be due to source files being too large. ADIFOR typically cannot handle individual files that are more than a few thousand lines long. If feasible, try breaking the file into several smaller files, by hand or using the `fsplit` utility.

- **Compiler complains about presence of $ character**

  If a program contains functions which have points of nondifferentiability, ADIFOR will insert calls to the exception handling library (see Section 4 and [4] for more details). At present, the subroutines in this library have names containing the $ character. This will change in future versions of ADIFOR. Presently, the simplest solution is to globally replace the $ character with some symbol which the compiler will accept.

Acknowledgments

We thank Moe El-Khadiri for his contributions to a preliminary version of this report. We also thank Jane Dudley, Larry Green, Laura Hall, Kara Haigler, Joe Manke, Janet Rogers and Karen Williamson for their perseverance in using early versions of ADIFOR and for sharing their insights with us. Lastly, we thank Mike Wenner for making the "advanced example" available to us.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 1992.

[2] Christian Bischof, Alan Carle, George Corliss, and Andreas Griewank. ADIFOR: Automatic differentiation in a source translator environment. In Paul Wang, editor, *International Symposium on Symbolic and Algebraic Computing 92*, pages 294–302, Washington, D.C., 1992. ACM.

[3] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[4] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Report ANL/MCS–TM–159 (also ADIFOR Working Note #3), Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[5] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Report ANL/MCS–TM–158 (also ADIFOR Working Note #2), Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[6] Herbert Fischer. Special problems in automatic differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 43 – 50. SIAM, Philadelphia, Penn., 1991.

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

[8] NAG. *NAGWare f77 Tools (Unix)*. The Numerical Algorithms Group Limited, London, 1991.

[9] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

Appendix A: A Generic Makefile

```
SHELL = /bin/csh

#
#      name of this makefile
#
ADIFOR_MAKE = make.adifor

#
#      name of top-level subroutine
#
AD_TOPLEVEL = ssinc

#
#      files containing top-level subroutine as
#      well as all files that are called below top-level
#      subroutine
#
ADIFOR_INPUT = wkhd.f toolcn.f timstp.f stif.f ssinc.f \
       smrgin.f rates.f outpt2.f noncon.f nodpsr.f \
       mgb1.f loadsm.f crvtur.f crvevl.f contpr.f \
       bndcnd.f bend.f
#
#      main program for top-level subroutine
#
ADIFOR_MAIN = admain.f

help:
       @echo "available targets:"
       @echo "portability -- run original program through NAG verifier"
       @echo "calltree    -- generate calling tree"
       @echo "declare     -- declare all variables explicitly"
       @echo "adifor      -- run adifor"
       @echo "compile     -- compile the adifor-generated subroutines"
       @echo "cleanup     -- delete adifor temporary files"

portability: $(ADIFOR_INPUT) $(ADIFOR_MAIN)
       nag_pfort -keepatr $(ADIFOR_INPUT) $(ADIFOR_MAIN)

calltree: $(ADIFOR_INPUT) $(ADIFOR_MAIN)
       nag_fcalls $(ADIFOR_INPUT) $(ADIFOR_MAIN)

declare: $(ADIFOR_INPUT) $(ADIFOR_MAIN)
       nag_decs  $(ADIFOR_INPUT) $(ADIFOR_MAIN)

adifor: $(AD_TOPLEVEL).adf $(AD_TOPLEVEL).comp
       adifor $(AD_TOPLEVEL).adf $(AD_TOPLEVEL).comp
       make -f ADMakefile

$(AD_TOPLEVEL).comp:  $(ADIFOR_INPUT) $(ADIFOR_MAIN)
       ls -1 $(ADIFOR_INPUT) $(ADIFOR_MAIN) > $@

# Concerning the ``compile'' target:
# The mechanism using 'ls' to determine the files generated by
# adifor is not perfect.  It might detect some files not generated
# by adifor.  However, since all it does with these files is
# recompile them, this should not pose a threat to the user's code.
```

```
compile:
	set src = `/bin/ls *\.[0-9a-f]*\.f | egrep -v 'unopt\.f$$'`; \
	set obj = `echo $$src | sed -e 's/\.f/\.o/g'`; \
	$(MAKE) -f $(ADIFOR_MAKE) compile2 ADIFOR_OBJECTS="$$obj" \
	ADIFOR_SOURCES="$$src"

compile2: $(ADIFOR_OBJECTS)
	$(F77) $(FFLAGS) $(ADIFOR_SOURCES)

#
#	The following entry removes all the temporary files created by ADIFOR.
#

clean :
	rm -rf $(AD_TOPLEVEL).dir adtmp ADDIR
	rm -f *.unopt.f $(AD_TOPLEVEL).comp
```