ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-192

ADIFOR 2.0 User's Guide

(Revision B)

by

Christian Bischof,[†] Alan Carle,[*] Peyvand Khademi,[†]
Andrew Mauer,[†] and Paul Hovland [†]

Mathematics and Computer Science Division
Technical Memorandum No. 192
and
Center for Research on Parallel Computation
Technical Report CRPC-95516-S

April 1995

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38, by the National Aerospace Agency under Purchase Order L25935D and Cooperative Agreement No. NCCW-0027, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

[†] Address: Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, {bischof, khademi, mauer, hovland}@mcs.anl.gov.

[*]Address: Center for Research on Parallel Computation, 6100 S. Main Str., Rice University, Houston, TX 77005-1892, carle@cs.rice.edu.

How to Get ADIFOR 2.0

To retrieve the ADIFOR 2.0 automatic differentiation software, visit the ADIFOR group World Wide Web home pages

`http://www.cs.rice.edu/fortran-tools/ADIFOR/adifor.html`

or

`http://www.mcs.anl.gov/Projects/autodiff/adifor.html`.

The home page contains a link to a request form for ADIFOR 2.0. Follow the instructions in the request form to acquire the ADIFOR 2.0 software. The ADIFOR group home page also contains links to a Unix `tar` format file containing the FORTRAN 77 source for the examples in this manual, and links to publications related to ADIFOR.

Legal Notices

Contents

ADIFOR 2.0 User's Guide
(Revision B)

Argonne Technical Memorandum ANL/MCS-TM-192
CRPC Technical Report CRPC-95516-S

Christian Bischof
bischof@mcs.anl.gov

Peyvand Khademi
khademi@mcs.anl.gov

Andrew Mauer
mauer@mcs.anl.gov

Paul Hovland
hovland@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue., Argonne, IL 60439

Alan Carle
carle@cs.rice.edu

Center for Research on Parallel Computation, MS 41
Rice University
Houston, TX 77005-1892

Abstract

Automatic differentiation is a technique for computing the derivatives of functions described by computer programs. ADIFOR implements automatic differentiation by transforming a collection of FORTRAN 77 subroutines that compute a function f into new FORTRAN 77 suborutines that compute the derivaties of the outputs of f with respect to a specified set of inputs of f. This guide describes step by step how to use version 2.0 of ADIFOR to generate derivative code. Familiarity with UNIX and FORTRAN 77 is assumed.

Chapter 1

For ADIFOR 1.0 Users

Users of ADIFOR 1.0 wishing to migrate to ADIFOR 2.0 should read at least the following sections of this manual:

Section 2.1 describes the `csh` environment variables that you must set to invoke ADIFOR 2.0.

Section 2.4 succinctly presents all of the steps required to process a code with ADIFOR 2.0 and compile and link a derivative-computing executable.

Chapter 3 describes the new options processing mechanism used by ADIFOR 2.0, and describes the new format of compositions.

Appendix D describes important changes in naming conventions between ADIFOR 1.0 and ADIFOR 2.0.

After reading these sections, and trying out ADIFOR 2.0 for the first time, we strongly suggest that you additionally read at least the following sections of the manual:

Chapter 5 documents the known deficiencies in our support for FORTRAN 77. The number of deficiencies is greatly reduced from that in ADIFOR 1.0.

Chapter 8 provides a list of problems that users of ADIFOR 2.0 may encounter.

Chapter 9 defines all of the options to ADIFOR 2.0 and presents their default values.

Appendix B describes the new exception handling mechanism of ADIFOR 2.0.

Appendix C describes the SparsLinC library, which provides support for sparse derivative computations within ADIFOR 2.0.

Chapter 2

Some Preliminaries

Automatic differentiation is a technique for computing the derivatives of functions described by computer programs. See [13, 18] for an introduction to automatic differentiation. ADIFOR implements automatic differentiation by transforming a collection of FORTRAN 77 subroutines that compute a function f into new FORTRAN 77 subroutines that compute the derivatives of the outputs of f with respect to a specified set of inputs of f. This paper describes step by step how to use version 2.0 of ADIFOR to generate derivative code. Familiarity with UNIX[1] and FORTRAN 77 is assumed.

We strongly suggest that you, before reading this manual, have a look at the overview paper of ADIFOR 2.0 [4]. It provides an overview of the philosophy of ADIFOR, references of successful applications of ADIFOR, and a perspective of how automatic differentiation relates to other approaches for computing derivatives.

The ADIFOR 2.0 system consists of the ADIFOR 2.0 preprocessor, the ADIntrinsics template expander and library, and the SparsLinC library. The `Adifor2.0` command invokes both the ADIFOR 2.0 preprocessor and the ADIntrinsics template expander. Figure 2.1 presents a block diagram of the ADIFOR 2.0 process. The process consists of three key steps:

1. Apply ADIFOR 2.0 to your FORTRAN 77 program to produce augmented code for the computation of derivatives. ADIFOR 2.0 invokes the ADIntrinsics template expander directly.
2. Construct a derivative driver code that invokes the derivative code generated by ADIFOR 2.0 and makes use of the computed derivatives.
3. Compile the FORTRAN 77 code generated by ADIFOR 2.0 and your derivative driver code, and link these with the derivative support packages, i.e., the ADIntrinsics exception handling package (see Appendix B), and (optionally) the SparsLinC sparse derivative package (see Appendix C).

The first step of this process can be performed on Sparc's running SunOS 4.1.3, SunOS 5.3 (Solaris), and IBM RS 6000's running version 3.2.5 of AIX, though the FORTRAN 77 code generated by the ADIFOR 2.0 preprocessor should be able to be executed on any machine on which you have a FORTRAN 77 compiler. We currently provide the necessary libraries for the second step precompiled for SunOS 4.1.3, SunOS 5.3, and IBM RS 6000's running version 3.2.5 of AIX, as well. Source code for the libraries is also provided in case you need to compile them to execute on other architectures. A "C" compiler is required to compile the SparsLinC library.

2.1 Configuration

To execute ADIFOR 2.0, set the environment variable `AD_HOME` to be the directory in which ADIFOR 2.0 is installed on your machine, the variable `AD_ARCH` to "sun4" (for a Sparc running SunOS 4.1.3 or SunOS 5.3) or "rs6000" (for an RS 6000 running AIX 3.2.5), and the variable `AD_OS` to "SunOS-4.x" (for a Sparc

[1] UNIX is a trademark of AT&T.

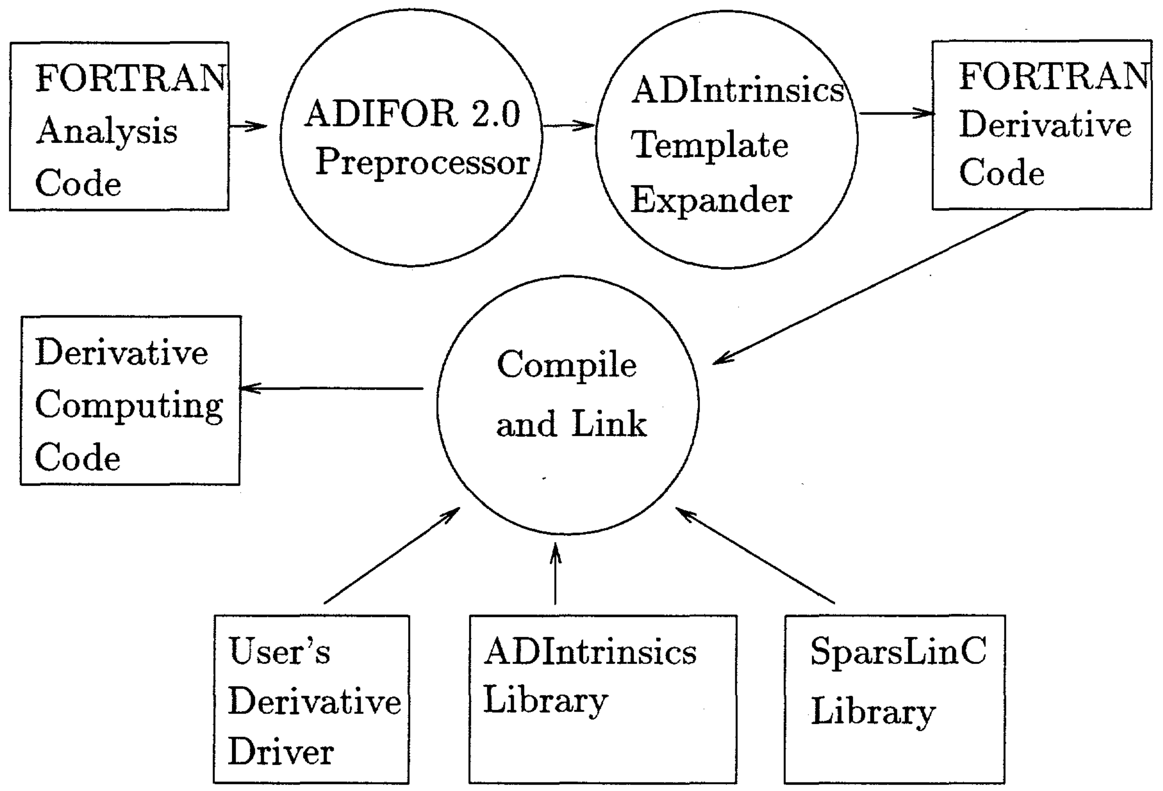

Figure 2.1. Block Diagram of the ADIFOR Process

```
setenv AD_HOME /usr/local/ADIFOR2.0
setenv PATH    $AD_HOME/bin:$PATH
setenv MANPATH $AD_HOME/man:$MANPATH
setenv AD_ARCH sun4
setenv AD_OS   SunOS-4.x
```

Figure 2.2. Portion of .cshrc File

running SunOS 4.1.3) or "SunOS-5.x" (for a Sparc running SunOS 5.3) or "AIX" (for an RS 6000 running AIX 3.2.5).

The directories "$AD_HOME/bin" and "$AD_HOME/man" should be added to your execution and manual paths, respectively. (The notation $X represents the value of the environment variable X.) We suggest modifying your ".cshrc" file to define AD_HOME and to modify your execution and manual paths. Figure 2.2 shows a fragment of a ".cshrc" file that has been modified assuming that the ADIFOR 2.0 executables and libraries have been installed into the directory /usr/local/ADIFOR2.0 on a Sparc running SunOS 4.1.3. The rest of this manual assumes that you have set AD_HOME and modified your execution path and manual path as just described.

The directory $AD_HOME contains the following subdirectories:

- **bin:** Contains Adifor2.0, the ADIFOR 2.0 preprocessor, and purse, the ADIntrinsics template expander. The template expander is responsible for expanding generic exception-handling macros introduced by ADIFOR 2.0 into the appropriate FORTRAN 77 code. The purse executable is a perl script, and we have provided perl in case it is unavailable on your system.
- **templates:** Contains the definition of the exception handling macros used by purse.
- **docs:** Contains postscript versions of relevant working notes and papers, including this manual.
- **examples:** Contains examples of programs processed with ADIFOR 2.0.
- **man:** Contains the man page for ADIFOR 2.0, purse, perl, ADIntrinsics, and SparsLinC.
- **src:** Contains the source for the ADIntrinsics and SparsLinC libraries.
- **lib:** Contains the precompiled versions of the ADIntrinsics and SparsLinC libraries.

You may want to consider compressing the files in the **src** and **docs** directories to save disk space.

2.2 How ADIFOR 2.0 Transforms a Program

In this section, we describe the mechanism used by ADIFOR 2.0 to transform your FORTRAN 77 code into code that computes derivatives of dependent variables with respect to independent variables. The mechanism has three key subtasks: code canonicalization, variable nomination, and code generation. Understanding these three tasks will help you better understand the derivative code that ADIFOR 2.0 generates. We briefly describe these subtasks in the next sections.

2.2.1 Code Canonicalization

In the code canonicalization phase, the FORTRAN 77 code is rewritten into a standard form. For example, expressions appearing as arguments to function or subroutine calls and function calls appearing within conditional tests are hoisted into assignments to new temporary variables. Statement functions are expanded into in-line code. This phase also breaks up long right-hand sides of assignment statements into smaller pieces, and rewrites them such that all variables appearing on the right-hand side of an assignment statement are of the same type. The latter transformation is needed for the code to be able to link in the SparsLinC library (see Appendix C).

2.2.2 Variable Nomination

ADIFOR 2.0 must decide which variables need to have "directional gradient objects" or "gradient objects" associated with them. ADIFOR 2.0 associates a gradient object with every variable whose value may depend on the value of a variable considered "independent" with respect to differentiation, and whose value impacts a variable considered "dependent" with respect to differentiation. Such a variable is called *active*. Variables that do not require derivative information are called *passive*.

ADIFOR 2.0 employs interprocedural analysis techniques to determine which variables in your code are active. First, ADIFOR 2.0 derives a "local interaction graph" for each subroutine. This is a bipartite graph where nodes representing input parameters or variables in common blocks are connected with nodes representing output parameters or variables in common blocks whose values they influence.

Next, an interprocedural analysis is performed, which determines, in essence, all possible program paths through which an independent variable can affect a dependent one and identifies intermediate variables that are involved along such a path. This analysis involves computing a transitive closure of the whole program graph composed from the local interaction graphs. In the presence of common blocks, equivalences, and arbitrary control structures, this is a nontrivial and computationally intensive process.

2.2.3 Code Generation

After active variables have been nominated, derivative code is generated for each assignment statement containing an active variable, and gradient objects are allocated. For assignment statements containing a FORTRAN 77 intrinsic, a template is generated that will later be instantiated by the ADIntrinsics system.

2.3 Functionality of ADIFOR 2.0-generated Code

Consider a function func with an n-vector x as independent and an m-vector y as dependent variables. That is, we have

```
subroutine func(n,x,m,y)
integer n, m
real x(n), y(m)
...
end
```

ADIFOR 2.0 inserts a gradient object g_x for x and g_y for y (as well as gradient objects for all other active variables in func) and replaces each assignment statement involving an active variable with a few assignment statements and a vector loop from 1 to g_p_. The interface of the code it generates is then

```
subroutine g_func(g_p_,n,x,g_x,ldg_x,m,y,g_y,ldg_y)
integer n, m, g_p_
real x(n), y(m), g_x(ldg_x,n), g_y(ldg_y,m)

...
end
```

So, for example, g_x(:,i) is the gradient object corresponding to x(i). If g_func is invoked with an integer g_p_ and values for x and g_x, then it computes y and

$$\text{g\_y} = (\frac{\partial y}{\partial x} \times \text{g\_x}^T)^T.$$

While somewhat inconvenient, the fact that the gradient dimension is the first dimension in the gradient objects cannot be avoided if we want to be able to deal with assumed-size arrays (e.g., declared as `real x(*)`).

```
program main
real x, y
read *, x
call squareroot(x, y)
print *, y
end
```

Figure 2.3. A Very Simple Program (main.f)

We now illustrate the flexibility inherent in the ADIFOR 2.0-generated code. First, recall the definition of the Jacobian of func,

$$J = \frac{d\,y}{d\,x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \in \mathbf{R}^{m \times n}.$$

Second, let $S = \mathbf{g\_x}^T$. We refer to S as the "seed matrix." The equation defining **g_y** above states that the ADIFOR 2.0-generated code computes $(J * S)^T$, where the number of columns p of S corresponds to the FORTRAN 77 variable **g_p_** in the generated code. Since most of the work of the derivative code is performed in the gradient loops (which range from 1 to **g_p_**), the size of p has a direct impact on the runtime and storage requirements for running the derivative code.

Properly initializing S, we can then obtain:

Full Jacobian: Choosing S as the $n \times n$ identity matrix, we compute the transpose of the full Jacobian J. The complexity of the resulting derivative code is $O(n)$ times that of the original function.

Jacobian-Vector Product: Choosing $S = d \in \mathbf{R}^n$, we compute the transpose of the Jacobian-vector product Jd in a time that is a small multiple of the function evaluation time. Since

$$Jd = \lim_{h \to 0} \frac{func(x + hd) - func(x)}{h},$$

this interface allows us to compute directional derivatives along arbitrary directions.

Selecting Derivatives: Choosing $S = [e_5, \ldots, e_{10}, e_{13}]$, where e_i is the ith canonical unit vector, i.e., an n-vector of all zeros except for an entry of 1 in the ith position, we compute the transpose of the 5th through 10th, and 13th columns of J.

See Appendix A for extensive information on seed matrix initialization.

2.4 A Quick Example

We demonstrate the use of ADIFOR 2.0, using its default configurations, with the very simple program shown in Figures 2.3 and 2.4. Procedure `squareroot` assigns the square root of the value of variable `x` to variable `y`. We now show, with only limited explanation, the sequence of steps required to construct a procedure that computes the derivative of `squareroot` at a user-specified value of `x`. A more detailed description of the ADIFOR 2.0 process and of the various options available in ADIFOR 2.0 is presented in Chapter 4.

1. Construct a composition `simple.cmp` that lists the names of all of the FORTRAN 77 source files that constitute the example program. Figure 2.5 shows the composition we construct.
2. Construct a script file `simple.adf` that tells ADIFOR 2.0 to differentiate the procedure named `squareroot` with the independent variable `x` and the dependent variable `y`, i.e., to generate code to compute the derivative $\frac{dy}{dx}$, where `y` is computed from `x` by procedure `squareroot`. The script file is shown in Figure 2.6.

```
      subroutine squareroot(x, y)
      real x, y
      y = sqrt(x)
      end
```

Figure 2.4. A Very Simple Program (squareroot.f)

```
main.f
squareroot.f
```

Figure 2.5. Script File (script.cmp) for Simple Example

```
AD_TOP   = squareroot
AD_PMAX  = 1
AD_IVARS = x
AD_DVARS = y
AD_PROG  = simple.cmp
```

Figure 2.6. Script File (script.adf) for Simple Example

```
      subroutine g_squareroot(g_p_, x, g_x, ldg_x, y, g_y, ldg_y)
        real x, y
        integer g_pmax_
        parameter (g_pmax_ = 1)
        integer g_i_, g_p_, ldg_y, ldg_x
        real r1_p, r2_v, g_y(ldg_y), g_x(ldg_x)
        if (g_p_ .gt. g_pmax_) then
          print *, 'Parameter g_p_ is greater than g_pmax_'
          stop
        endif
        r2_v = sqrt(x)
        if ( x .gt. 0.0e0 ) then
           r1_p = 1.0e0 / (2.0e0 *  r2_v)
        else
           call ehufSV (9, x, r2_v, r1_p, 'g_squareroot.f', 32)
        endif
        do g_i_ = 1, g_p_
          g_y(g_i_) = r1_p * g_x(g_i_)
        enddo
        y = r2_v
C--------
      end
```

Figure 2.7. Derivative Code (g_squareroot.f)

```
      program driver
      real x,y
      real g_x(1), g_y(1)

      real *, x
      g_x = 1.0
      call g_squareroot(1, x, g_x, 1, y, g_y, 1)

      print *, y
      print *, g_y
      end
```

Figure 2.8. Derivative Code Driver (`driver.f`) for the Very Simple Example

```
f77 -c driver.f
f77 -c g_squareroot.f
f77 -o driver driver.o g_squareroot.o          \
       $AD_HOME/lib/ReqADIntrinsics-$AD_ARCH.o \
       $AD_HOME/lib/libADIntrinsics-$AD_ARCH.a
```

Figure 2.9. Commands to Compile and Link Derivative Code Executable

3. Create, in the `output_files` subdirectory, the procedure `g_squareroot`, as shown in Figure 2.7, by executing the command

   `Adifor2.0 AD_SCRIPT=simple.adf`.

   Note that an exception handler is invoked when `sqrt` is invoked with a zero argument, as the derivative of $\sqrt{}$ is undefined.
4. Create the derivative code driver `driver.f` as shown in Figure 2.8. The driver invokes `g_squareroot` with a user-specified value of `x` to compute the value of `y` and $\frac{dy}{dx}$.
5. Compile and link `driver.f`, `g_squareroot.f` and the ADIntrinsics exception handling archive using the commands shown in Figure 2.9 to build the desired derivative computing executable.

2.5 A Roadmap

The rest of this manual is organized as follows:

Chapter 3 describes how to set up the inputs to the ADIFOR 2.0 preprocessor to enable it to generate derivative code. The input to ADIFOR 2.0 takes the form of *option bindings* that are specified on the command line or in startup files, and *compositions*, lists of FORTRAN 77 files that constitute the program that contains the function to be differentiated.

Chapter 4 is devoted to a step-by-step description of how to process a code by using ADIFOR 2.0 and an explanation of how ADIFOR 2.0-generated code should be incorporated into a program.

Chapter 5 documents the known deficiencies in ADIFOR 2.0's support for FORTRAN 77. For each deficiency, a workaround is presented.

Chapter 6 explains how to apply ADIFOR 2.0 in cases where the "function to be differentiated" does not have the form expected by ADIFOR 2.0.

Chapter 7 covers some of the pitfalls associated with automatic differentiation of FORTRAN 77 programs.

Chapter 8 provides a list of problems that users of ADIFOR 2.0 may encounter.

Chapter 9 defines all of the options to ADIFOR 2.0 and presents their default values.

Appendix A describes seed matrix initialization, a powerful concept that provides users of ADIFOR 2.0 significant control over the computation performed by the generated derivative code, and allows one to compute arbitrary directional derivatives.

Appendix B describes the ADIntrinsics template expander and library. ADIntrinsics provides user-customizable handling of exceptions within ADIFOR 2.0.

Appendix C describes the SparsLinC library, which provides support for sparse derivative computations within ADIFOR 2.0.

Appendix D describes changes in naming conventions between ADIFOR 2.0 and ADIFOR 1.0 and is intended to be read only by users of ADIFOR 1.0.

Chapter 3

Specifying Input for ADIFOR 2.0

In order to apply ADIFOR 2.0 to a set of FORTRAN 77 procedures to generate derivative code, it is necessary to tell ADIFOR 2.0 several key pieces of information:

1. The names of the files containing the FORTRAN 77 source code to be processed. The names of the procedures are provided to ADIFOR 2.0 in a file referred to as a *composition*. ADIFOR 2.0 must be told the name of the file containing the composition.
2. The name of the "top routine," that routine whose invocation causes the function to be evaluated. ADIFOR 2.0 determines the names of all of the routines that may be transitively invoked by the top routine by examining the source code.
3. The names of the independent and dependent variables. The ADIFOR 2.0-generated code computes the derivatives of the dependent variables with respect to the independent ones.
4. Values of numerous other options to ADIFOR 2.0 that control how vector operations in the ADIFOR-generated code are implemented, what level of exception reporting for nondifferentiable FORTRAN 77 intrinsics is performed, and to what extent the code should be customized for particular execution environments.

The names of the composition file, the top routine, and the independent and dependent variables, and values for the various options, are provided to ADIFOR 2.0 in the form of bindings, as described in the next section. Section 3.2 describes the format of compositions. Section 3.3 describes source files that are acceptable for processing with ADIFOR 2.0 and describes some common deviations from the FORTRAN 77 standard that cause problems.

3.1 Option Processing in ADIFOR 2.0

This section describes ADIFOR 2.0's option-processing mechanism. Information is provided to ADIFOR 2.0 as bindings. Bindings have the form

```
OPTION = VALUE,
```

or

```
OPTION = VALUE1,..., VALUEN.
```

The second form is used in defining list-valued options. Bindings may be provided as command line arguments and, additionally, as lines in a "script" file. Bindings specified as command line arguments to ADIFOR 2.0 may contain whitespace, consisting of a sequence of spaces and tabs, if they are quoted.

A script file is a sequence of lines. Blank lines are ignored. Each nonblank line contains a binding having either of the two forms shown above. All characters on a line after the comment character '#' are ignored. There is no formal requirement for the name of the script file, but our informal convention is to use the `.adf` extension. Bindings defined in script files may always use whitespace liberally.

```
AD_PROG  = rosenbrock.cmp
AD_TOP   = func
AD_IVARS = x
AD_DVARS = y
AD_PMAX  = 2    # x has 2 elements
AD_OUTPUT_DIR = .
```

Figure 3.1. Example Script File (rosenbrock.adf)

All ADIFOR 2.0 options begin with an "AD_" prefix. Values of options are typically the names of files (**AD_PROG**), the names of procedures in the program (**AD_TOP**), lists of names of variables in the program (**AD_IVARS** and **AD_DVARS**), integers (**AD_PMAX**), Boolean values (**AD_DUMP_CALLGRAPH**), and switches (**AD_FLAVOR** and **AD_EXCEPTION_FLAVOR**). For Boolean-valued options, `false` and `0` are considered to be equivalent, as are `true` and `1`. Section 9 documents all of the ADIFOR 2.0 options.

ADIFOR 2.0 processes bindings on its command line in the order that they are listed. As bindings are processed, new bindings always override values defined by a previous binding for the same option. The option **AD_SCRIPT** is used to specify the name of a script file. Whenever a binding for **AD_SCRIPT** is encountered, the file identified as the value of **AD_SCRIPT** is opened, and the bindings in the file processed in order.

Relative path names specified as command line arguments to ADIFOR 2.0 are taken as relative to the directory in which ADIFOR 2.0 was executed. Relative path names specified in bindings specified in a script file are taken as relative to the directory containing the script file.

Now consider a sequence of examples using the script file `rosenbrock.adf` shown in Figure 3.1.

- **Example 1**

  ```
  Adifor2.0 AD_PROG=rosenbrock.cmp AD_TOP=func \
          AD_DVARS=y AD_IVARS=x AD_PMAX=2 AD_OUTPUT_DIR=.
  ```

  This command defines **AD_PROG** to be the filename "rosenbrock.cmp", **AD_TOP** to be name of the procedure "func", **AD_IVARS** to be the (single item) list "x", **AD_DVARS** to be the (single item) list "y", and **AD_PMAX** to be the integer value 2. ADIFOR will place derivative files in the current directory (which in UNIX is usually denoted by a dot).

- **Example 2**

  ```
  Adifor2.0 AD_SCRIPT=rosenbrock.adf
  ```

  This command defines exactly the same values for the same set of options.

- **Example 3**

  ```
  Adifor2.0 AD_SCRIPT=rosenbrock.adf AD_PMAX=5
  ```

  This command defines the exactly the same set of values for the same set of options, except for option **AD_PMAX** whose value is overridden with the integer value 5.

3.2 Compositions

Compositions list the names of all of the source files to be processed by ADIFOR 2.0. A composition is a list of pathnames to source files with zero, one, or more pathnames per line. All characters on a line after the comment character '#' are ignored. Multiple pathnames on the same line are delimited by commas and whitespace, where whitespace is any sequence of spaces or tabs. Relative pathnames are taken to be relative to the directory containing the composition.

The name of the composition must end with a ".cmp" extension. The name of each source file must end with a ".f" suffix. Each source file listed in a composition may contain the source for one or more FORTRAN 77 routines.

Since ADIFOR 2.0 uses interprocedural analysis to reduce the cost of computing derivatives, a composition must describe a *complete* and *consistent* program. To be *complete*, a program must not have missing entry points, i.e., the program must link without undefined external references. To be *consistent*, all procedure interfaces in the program must agree as to the number of arguments and the types of the arguments being passed. Many programs in use today have inconsistent interfaces. Fixing the inconsistencies may take significant effort, but is usually an enlightening process.

In addition to being complete and consistent, your program must not be recursive. ADIFOR 2.0 will complain if it encounters a recursive program and will print out the names of each of the routines that are recursive. Recursion in FORTRAN 77 programs is usually, but not always, an indication of some underlying error.

When ADIFOR 2.0 generates derivative code for a file `somedir/foo.f`, it places the generated source code into a file `g_foo.f` in the subdirectory identified by the option `AD_OUTPUT_DIR` of the directory in which ADIFOR 2.0 was executed. Therefore, no two pathnames listed in a composition may have the same basename, where the basename of `somedir/foo.f` is taken to be `foo.f`. ADIFOR 2.0 will complain if multiple files in your program have the same basename.

In reality, it is necessary to submit to ADIFOR 2.0 only the portion of the program that defines the function to be differentiated. This usually requires that you create a *dummy main* procedure that invokes the routines to be differentiated (see Chapter 4).

3.3 Acceptable FORTRAN 77 Source Files

ADIFOR 2.0 recognizes standard FORTRAN 77 syntax extended with `DO-ENDDO`, `IMPLICIT NONE`, `DOUBLE COMPLEX`, and `INCLUDE`. Variable names need not be limited to six characters. If a program uses nonstandard extensions, ADIFOR 2.0 will probably not accept them. In particular, ADIFOR 2.0 will not accept nonstandard intrinsic or type conversion functions, such as `arsin()`, `arcos()`, and `dfloat()`. These should be replaced with standard functions like `asin()`, `acos()`, and `dble()`. In any case, for portability reasons, it is probably a good idea anyway to make sure that all code is standard-conforming. Also not accepted are system calls such as `etime()`. In most cases, such calls do not affect function evaluation and may be removed, commented out, or replaced with a syntactically correct but nonfunctional subroutine, prior to processing with ADIFOR 2.0.

We strongly urge you to make sure that all of the files in your composition compile correctly and adhere to the FORTRAN 77 standard before submitting them to ADIFOR 2.0 for processing. ADIFOR will complain about syntax errors, but its error messages are likely to be more cryptic. ADIFOR 2.0 will also complain about problems in your source code that the typical FORTRAN 77 compiler will fail to identify, specifically, inconsistencies between callsites and the procedures they invoke, and inconsistencies between common block declarations across procedures.

For example, in the following program fragment an `integer*4` array of length 3 is passed to a subroutine whose arguments were declared to be of type `character*12`.

```
      program main
      integer*4 x(3)
      ...
      call func(x)
      ...
      end

      subroutine func(c)
      character*12 c
      ...
      end
```

The following program fragment declares common blocks to be of different length in different program units.

```
      program main

      call func1
      call func2
      ...
      end

      subroutine func1
      common /cmn/ x(10)
      ...
      end

      subroutine func2
      common /cmn/ x(20)
      ...
      end
```

The FORTRAN 77 language definition requires that each common block, other than the blank common block //, must have the same size in each procedure in which it is declared. Another violation of the FORTRAN 77 standard is the fact that the common block is not declared in the main program (from which both subroutines are called. While this is usually not an issue, because of the nature in which global variables are implemented, unexpected things could happen if a compiler exploited the liberty of the standard.

Chapter 4

A Tutorial Example

We demonstrate the use of ADIFOR 2.0 using the simple program shown in Figures 4.1 and 4.2. 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 $\frac{dy}{dx}s = -y$. Our goal will be to replace the subroutine `fprime`, which approximates $\frac{dy}{dx}$ by using central divided differences, with an ADIFOR-generated derivative code. This complete example is provided in `$AD_HOME/examples/newton`.

Rosenbrock's function is used only for illustrative purposes. It is not indicative of the power of ADIFOR, which has processed programs up to 60,000 lines in length, albeit using 280 Mb of virtual memory in the process.

Step 1: Create a Composition File

Figure 4.3 presents composition `rosenbrock.cmp` for the example, assuming that `newton`, `func`, and `fprime` have been stored into the files `newton.f`, `func.f`, and `fprime.f`, and that code for `dlange` and `dgesv` and all of the routines that they invoke has been located.

As mentioned in Section 3.2, it is only really necessary to provide ADIFOR 2.0 with the source code that defines the function to be differentiated. Figure 4.4 presents a dummy main program that directly invokes procedure `func`. Note, in particular, that no variables have to be initialized in the dummy main program, since we have no plans to execute this code. Figure 4.5 presents the composition `rosenbrock-func-only.cmp` that includes this dummy main program.

Step 2: Determine Values for ADIFOR 2.0 Options

To compute a Jacobian for the Newton example, you must provide ADIFOR 2.0 with values for the following options:

`AD_PROG`: The value of `AD_PROG` is the name of the "composition" to be processed. The name of the composition is communicated to ADIFOR 2.0 by specifying `AD_PROG=composition-name` on the command line.

In this example, `AD_PROG` will be set to `rosenbrock.cmp` or `rosenbrock-func-only.cmp`.

`AD_TOP`: The value of `AD_TOP` is the name of the procedure that contains the function to be differentiated. That procedure may then, transitively, invoke a set of other procedures. We refer to the procedure that is invoked to evaluate the function as the *top-level routine* or **TOP**. The name of the procedure **TOP** is communicated to ADIFOR 2.0 by using the command line option `AD_TOP=procedure-name`.

In Section 6 we will describe how to handle codes where the function to be differentiated does not conveniently correspond to a procedure invocation.

```
      PROGRAM NEWTON
      DOUBLE PRECISION DUMMY,TOL, DLANGE
      INTEGER INFO, N, IPIV(2)
      DOUBLE PRECISION X(2),Y(2),YPRIME(2,2)
      EXTERNAL DGESV, FPRIME, FUNC, DLANGE
      TOL = 1.0E-12
      WRITE (*,FMT=*) 'Input 2-element starting vector '
      READ (*,FMT=*) X(1),X(2)
      CALL FUNC(X,Y)
   10 IF (DLANGE('1',2,1,Y,2,DUMMY).LT.TOL) GO TO 20
      CALL FPRIME(X,Y,YPRIME)
      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)
      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 4.1. A Simple Implementation of Newton's Method

In this example, the function to be differentiated corresponds to the subroutine func, so we will set AD_TOP to be func.

AD_IVARS and AD_DVARS: The values of AD_IVARS and AD_DVARS are comma-separated lists of independent (input) and dependent (output) variables of **TOP**, respectively. AD_OVARS is a synonym for AD_DVARS. A variable may be designated as independent, dependent, or both (if it is overwritten during the execution of AD_TOP).

There is no way to nominate individual elements of a FORTRAN 77 array as being independent and dependent, although it is possible to specify at run time that only derivatives with respect to a particular set of elements should be computed (see Appendix A). Variables in the AD_IVARS and AD_DVARS lists must have type real, double precision, complex or double complex. The independent and dependent variables must be formal parameters of **TOP**, or global variables declared within **TOP**. Again, in Section 6 we will describe how to handle codes in which the variables that logically correspond to the independent and dependent variables are neither formal parameters nor global variables in **TOP**.

In this example, in order to compute the derivatives of y with respect to x, we will set AD_DVARS to y and AD_IVARS to x.

AD_PMAX: The value of AD_PMAX is the upper bound on the number of independent variables for which derivatives can be computed simultaneously. It is necessary to specify this upper bound because FORTRAN 77 does not provide a standard mechanism for dynamic memory allocation. It is introduced as the first dimension of each of the gradient objects declared by ADIFOR 2.0. The value of AD_PMAX is communicated by using the option AD_PMAX=integer-value.

In the Newton example, we choose to set AD_PMAX to 2, since x is an array with 2 elements and we would like to compute derivatives with respect to x(1) and x(2). In general, in the invocation of the routines generated by ADIFOR 2.0, we can use any value of g_p_ that is not larger than AD_PMAX. This issue is explained in more depth in Appendix A. We also note that if subroutines using the same

```
      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
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 4.2. Rosenbrock's Function and Divided-Difference Approximations of the Jacobian

```
newton.f
func.f
fprime.f

# LAPACK routines
dlange.f dgesv.f lsame.f dlassq.f
xerbla.f dgetrf.f dgetrs.f ilaenv.f
dgetf2.f dlaswp.f

# BLAS routines
dtrsm.f dgemm.f idamax.f dswap.f dscal.f dger.f
```

Figure 4.3. Composition for Newton's Method Example (rosenbrock.cmp)

```
      PROGRAM MAIN
      DOUBLE PRECISION X(2),Y(2)
      CALL FUNC(X,Y)
      END
```

Figure 4.4. Dummy Main for Newton's Method Example (dummy.f)

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

Figure 4.5. Composition for Newton's Method Example (rosenbrock-func-only.cmp)

common blocks are processed separately with ADIFOR 2.0, it is essential to use the same value of AD_PMAX in both cases, as otherwise the gradient object common blocks are declared inconsistently.

AD_OUTPUT_DIR: The value of AD_OUTPUT_DIR specifies the name of the directory in which ADIFOR 2.0 places the generated derivative code.

In the Newton example, we have chosen to set AD_OUTPUT_DIR to be "." so that the generated code will be placed back into the directory in which ADIFOR 2.0 is executed.

Step 3: Invoke ADIFOR 2.0

When executed with the command:

```
Adifor2.0 AD_PROG=rosenbrock.cmp AD_TOP=func \
      AD_DVARS=y AD_IVARS=x AD_PMAX=2 AD_OUTPUT_DIR=.
```

ADIFOR 2.0 creates the subdirectory AD_cache, which contains internal information created by ADIFOR 2.0. Source files generated by ADIFOR 2.0 are placed in the working directory. If AD_OUTPUT_DIR had been unspecified, then the default value of output_files would have caused the generated files to be placed into the subdirectory output_files. ADIFOR 2.0 emits the augmented code for procedure func into the file g_func.f, whose source is shown in Figure 4.6. Note that usually an assignment statement in the original code has been replaced by a few assignment statements and a vector loop of length g_p_. When g_p_ is moderate, or the gradient objects always dense vectors, this is an efficient representation of this vector operation. The SparsLinC library (see Appendix C) provides an alternative approach for expressing this vector operation when the gradient objects are mostly sparse vectors.

Exactly the same processing process will be performed by executing the command

```
Adifor2.0 AD_SCRIPT=rosenbrock.adf
```

by using the script file that was shown in Figure 3.1.

Step 4: Incorporate ADIFOR-generated Subroutine

Incorporating the ADIFOR-generated subroutine into a program to compute derivatives requires the following three steps:

1. **Allocate the gradient objects in the calling module.** The user should carefully check the ADIFOR-generated code to determine which variables in common blocks and which arguments to the top-level routine have been found to be active. For our small example, the declarations are

   ```
   double precision g_x(PMAX,2), g_y(PMAX,2)
   ```

   where PMAX is an integer constant (FORTRAN 77 PARAMETER) whose value is greater than or equal to the value of AD_PMAX. In this case, we choose to set PMAX to 2.

2. **Initialize the seed matrix.** In order to compute the Jacobian of the function defined by func, the gradient object for the independent variable x should be initialized to a 2×2 identity matrix. This initialization amounts to saying that the derivative of each independent variable with respect to itself is 1.0.

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 the calls to func and fprime by a single call to g_func.

   In the call to the ADIFOR-generated top-level subroutine, the parameter g_p_ should be set equal to the length of the gradient objects, and all of the ldg_ variables should be set equal to the leading dimension with which the corresponding gradient objects (g_ variables) were actually declared. Thus, for our simple example, the call would look like

```
      subroutine g_func(g_p_, x, g_x, ldg_x, y, g_y, ldg_y)
        double precision x(2), y(2)
C
        integer g_pmax_
C
        parameter (g_pmax_ = 2)
        integer g_i_, g_p_, ldg_y, ldg_x
        double precision d5_b, d2_b, g_y(ldg_y, 2), g_x(ldg_x, 2)
        intrinsic dble
C
C
        if (g_pmax_ .gt. g_p_) then
          print *, 'Parameter g_pmax_ is greater than g_p_'
          stop
        endif
        d2_b = dble(10.0)
        d5_b = -d2_b * x(1) + (-d2_b) * x(1)
        do g_i_ = 1, g_p_
          g_y(g_i_, 1) = d5_b * g_x(g_i_, 1) + d2_b * g_x(g_i_, 2)
        enddo
        y(1) = dble(10.0) * (x(2) - x(1) * x(1))
C--------
        do g_i_ = 1, g_p_
          g_y(g_i_, 2) = -g_x(g_i_, 1)
        enddo
        y(2) = 1.0d0 - x(1)
C--------
        return
      end
```

Figure 4.6. The ADIFOR-generated Code for Subroutine `func`

```
      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(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 4.7. The Driver for the Newton Program Using ADIFOR-generated Code

```
          call g_func(2, x, g_x, PMAX, y, g_y, PMAX)
```

For our example, the new driver is shown in Figure 4.7.[1] As mentioned above, since ADIFOR-generated derivative code computes the transpose of the Jacobian, we must retranspose g_y before passing it to dgesv. Together with the subroutine func and the subroutine shown in Figure 4.6, the new program replaces the program shown in Figure 4.1.

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.

```
f77 -c adnewton.f
f77 -c g_func.f
f77 -c dlange.f
f77 -c dgesv.f
f77 -c ...
```

The compiled modules should then be linked together with the ADIntrinsics package to generate a working executable.

```
f77 -o adnewton adnewton.o g_func.o dlange.o dgesv.o ... \
       $AD_HOME/lib/ReqADIntrinsics-$AD_ARCH.o \
       $AD_HOME/lib/libADIntrinsics-$AD_ARCH.a
```

The module ReqADIntrinsics-$AD_ARCH.o and archive libADIntrinsics-$AD_ARCH.a implement the ADIFOR 2.0 exception handling mechanism. See Appendix B for more information on the ADIntrinsics template expander and library.

[1] Some comments were removed to fit the program on one page.

Chapter 5

Known Deficiencies

In this section we describe several deficiencies in ADIFOR 2.0's support of full FORTRAN 77. In each case, it is relatively easy to "work around" each of these deficiencies. ADIFOR 2.0 flags each of these as being "not supported" any time that they are encountered.

5.1 Intrinsics Passed as Procedure Parameters

ADIFOR 2.0 prohibits intrinsics, such as DSIN and DCOS, from being passed as procedure parameters as shown in the standard-conforming FORTRAN 77 code:

```
      subroutine bad(x0, x1)
      double precision x0, x1
      external integrate
      intrinsic dsin

      call integrate(dsin,x0, x1)
      end
```

This deficiency can easily be circumvented by introducing a wrapper function for each intrinsic, which is to be passed as a procedure parameter, and by then passing that wrapper routine as the procedure parameter instead of the intrinsic. For example, the following code performs the same computation as the code shown above by using a wrapper function MYDSIN for intrinsic DSIN:

```
      subroutine good(x0, x1)
      double precision x0, x1
      external integrate, mydsin
      call integrate(mydsin, x0, x1)
      end

      function mydsin(x)
      double precision x
      intrinsic dsin
      mydsin = dsin(x)
      end
```

5.2 Intrinsics Overridden by External Functions

ADIFOR 2.0 prohibits external routines from overriding intrinsic functions as shown in the standard-conforming FORTRAN 77 code:

```
      subroutine bad(x,y)
      external cos
      double precision x, y, cos
c     call user defined function with name "cos"
      y = cos(x0)
      end

      function cos(x)
      ...
      end
```

Again, this deficiency can easily be circumvented by renaming the external function so that it does not collide with the name of any intrinsic function, as follows:

```
      subroutine good(x,y)
      external mycos
      double precision x, y, mycos
      y = mycos(x0)
      end

      function mycos(x)
      ...
      end
```

5.3 I/O Statements That Contain Function Invocations

ADIFOR 2.0 prohibits I/O statements, **READ**, **WRITE**, and **PRINT**, from invoking functions and statement functions as shown in the standard-conforming FORTRAN 77 code:

```
      subroutine bad(y)
      double precision y(10)
      integer  f
      external f
      read  (3, 50) x, y(f(x))
 50   format (...)
      end
```

Modifying code that invokes functions from within I/O statements is very easy, but may change the meaning of the I/O statements in ways that require other I/O statements in the program to be changed as well. For example, the function call in the **READ** statement above can be removed from an I/O statement by rewriting the code as follows:

```
      subroutine okay(y)
      double precision y(10)
      integer  f, i
      external f
      read  (3, 50) x
      i = f(x)
      read  (3, 51) y(i)
50    format (...)
51    format (...)
      end
```

Notice, however, that in the original code, the two elements that are read may come from the same line of the input file, while in the new code, the two elements must come from different lines of the input file.

Chapter 6

Advanced Topics

Normally, ADIFOR 2.0 assumes that independent variables are passed into the top-level routine **TOP**, and dependent variables are passed back out to the procedure that invoked **TOP**. Furthermore, it is assumed that the values of the independent variables will be assigned before **TOP** is invoked. "Passing" is either via procedure parameters or via global variables in common blocks. So, the normal ADIFOR 2.0 interface cannot compute derivatives of the following:

- variables that are declared and computed in the main program,
- variables that are declared locally in the top-level routine or variables declared in a routine transitively invoked by the top-level routine,
- variables that are assigned values during evaluation of `AD_TOP` and then overwritten, and
- variables that are initialized by a `READ` statement.

This section describes some workarounds for these situations.

6.1 Computation Is Not Encapsulated in Procedure

Consider 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)
      end
```

To extract a procedure suitable for using ADIFOR 2.0 to generate code for $\frac{\partial \mathtt{y}}{\partial \mathtt{x(1)}}$ and $\frac{\partial \mathtt{y}}{\partial \mathtt{x(2)}}$, you should rearrange the computation so that both `x(1)` and `x(2)` are initialized first, then invoke a new procedure that computes `y` from `x(1)` and `x(2)` and then returns the value of `y` as follows:

```
program main
read(*,*) x(1)
read(*,*) x(2)
y = compute(x(1), x(2))
end

function compute(x1, x2)
y = result of some computation involving x1 and x2
end
```

6.2 Variables Other Than Parameters and Globals in AD_TOP

Consider the following 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 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. 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
call foo(x,y,a,b)
end

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

or, alternatively,

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

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

An alternative to this workaround is the buddy system discussed below.

6.3 Variables That Are Overwritten

Consider the following program:

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

      subroutine foo(x,y)
10    y = x*x
20    y = y * x
      end
```

Say we want to compute the derivatives with respect to x of variable y at both the statement with label 10 and the statement with label 20. Nominating variable y as the dependent variable, will generate code that computes only the derivative of y at the statement with label 20.

In order to avoid this problem, we can expand y into an array and modify the code to the code that follows:

```
      program main
      real y(2)
      call foo(x,y)
      end

      subroutine foo(x,y)
      real y(2)
10    y(1) = x * x
20    y(2) = y(1) * x
      end
```

6.4 Variables Involved in I/O Statements

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 procedure does not pose a problem, as long as the independent variables are parameters or global variables in AD_TOP, and I/O functions are handled properly. Unfortunately, we cannot automate the proper handling of I/O functions involving active variables because, in general, we have no way to trace the flow of data values that are read or written to files.

Without this information, we have no way of knowing whether the gradient object for a variable that is involved in a READ statement should be set to 0.0 or initialized by reading in derivative values from the file system. Similarly, we have no way of knowing whether we should write the values of the gradient objects for variables involved in a WRITE statement to the file system. Therefore, ADIFOR 2.0 currently just echoes I/O statements like READ and WRITE without introducing code to initialize or propagate the derivatives of variables involved in the I/O statement. Because of the problems that this approach may cause, ADIFOR 2.0 generates a warning message whenever it processes a source file that contains an I/O statement involving an active variable. The warning message is printed out to stderr as the code is processed, and embedded as a comment just before the suspect I/O statement.

Fortunately, in most of the cases that we have encountered, it is possible to use a scheme based on "buddy variables" to modify the original function code in a manner that makes it possible for ADIFOR 2.0 to generate correct derivative code in the presence of I/O of active variables. As an example, consider trying to process the following code to compute the derivative of e at the statement with label 20 with respect to h at the statement with label 10:

```
      program main
      real lambda
      read *, lambda
      call foo(lambda)
      end

      subroutine foo(lambda)
      real lambda, e, h
10    read *, h
      e = h * lambda
20    write *, e
      end
```

One approach to modifying this code would be to extract the READ statements in foo into main, and to convert variables e and h into parameters to foo. As an alternative, consider modifying the original code into the following code:

```
      program main
      real lambda, hbuddy, ebuddy
      common /buddyvar/ hbuddy, ebuddy

      read *, lambda
      call foo(lambda)
      end

      subroutine foo(lambda)
      real lambda, e, h, hbuddy, ebuddy
      common /buddyvar/ hbuddy, ebuddy

      h = 0
10    read *, h
      h = h + hbuddy
      e = h * lambda

      ebuddy = e
20    write *, e
      end
```

and then nominating hbuddy as the independent variable, and ebuddy as the dependent variable. Initialization of hbuddy to 0.0, and g_hbuddy to 1.0, in the derivative driver for g_foo then results in g_ebuddy being assigned the derivative of e with respect to h. Notice that nominating hbuddy and ebuddy as the independent and dependent variables forces variables h and e to be active. Since h is assigned the value 0.0 prior to the read statement, g_h will be assigned the value 0.0. Therefore, since g_hbuddy is initialized to 1.0, g_h will be assigned the value 1.0 just after the READ, as required to compute the derivative of e with respect to h. Finally, the value of the computed derivative can be returned via the global variable g_ebuddy.

The scheme that we just described has three key components. The first component forces variables in I/O statements that depend on the independent variables and that are used to compute dependent variables to be identified as active variables. The second component forces the derivatives of variables appearing in READ statements to be initialized properly. Finally, the third component makes it possible to retrieve the values of the derivatives for variables that appear in WRITE statements.

Chapter 7

Pitfalls of Differentiating FORTRAN 77

Some operations that are allowed in FORTRAN 77 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 the integer variable 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.

- **Equivalencing of variables of different types**

  The process of equivalencing variables that have different types such as in the following code fragment

  ```
        real r(10)
        double precision d(5)
        complex z(5)
        equivalence(r,d)
        equivalence(r,z)
  ```

  has no real mathematical meaning. Thus, if a program performs this operation, ADIFOR 2.0 will generate the corresponding equivalences for the gradient objects of the equivalenced variables, but they (and any gradient objects which depend on them) may have meaningless values. Note that this form of equivalencing is nonportable anyway, since its results depend heavily on the floating-point representation.

- **Introducing points of nondifferentiability**

  Sometimes, for the sake of improving efficiency, a program tests the value of a variable to see whether a function is being evaluated at a special point in space, and 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 1.0 (because the statement $y = x$ implies that $\frac{dy}{dx} = \frac{dx}{dx} = 1$) rather than the expected 0.0. Similarly, the value of $\frac{dy}{dx}|_{x=1}$ will be 1.0 rather than 4.0. 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, is discussed in [11].

Chapter 8

Potential Problems

Users may encounter several problems while trying to process programs with ADIFOR 2.0. We provide a brief explanation of each and possible solutions.

- **ADIFOR 2.0 may complain about errors in the original FORTRAN 77 source code**
  As discussed in Section 3.3, ADIFOR 2.0 may report that errors are present in your FORTRAN 77 program that typical FORTRAN 77 compilers will not detect. Inconsistencies in subroutine interfaces and common blocks are the most frequently reported errors (see Section 3.3).
- **ADIFOR 2.0-generated code fails to link on a Sparc**
  Sun changed the interface to the internal I/O routines provided in `libF77.a` between versions SC1.0 and SC2.0 of the `f77` compilation system. The version of the ADIntrinsics library that we provide has been compiled using version SC2.0. Unresolved references for entries beginning with three underscores, such as `___do_l_in`, `___do_l_out`, `___e_rsle`, `___s_rsle`, and `___flushio`, will be reported if you attempt to compile your source files with version SC1.0 and link against the libraries we provide. If you do not have access to version SC2.0, it may be necessary for you to recompile the ADIntrinsics library with version SC1.0.
- **ADIFOR 2.0 may generate subscripted variables with more than 7 dimensions**
  If the source code being differentiated contains active variables that are declared as arrays with 7 dimensions, then ADIFOR 2.0, when generating dense derivative code, will insert gradient objects with 8 dimensions. FORTRAN 77 limits the number of dimensions for arrays to 7. It is unlikely that you will run into this problem, but if you do, then check your compiler to see whether it has an option that will extend its limits.
- **ADIFOR 2.0 may generate variable names longer than 6 characters**
  ADIFOR 2.0 generates names for new variables that may be more than 6 characters long. FORTRAN 77 limits the number of characters in a name to 6, but all compilers we have worked with extend this limit. It is unlikely that you will run into this problem. If you do, then check your compiler to see whether it has an option that will extend its limits.
- **ADIFOR 2.0 generates `DO-ENDDO` loop statements instead of introducing a labeled `CONTINUE` statement to end each loop**
  The `DO-ENDDO` statement is not standard FORTRAN 77, but is accepted by all compilers that we have encountered.
- **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 ensure 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.

- **By default, ADIFOR 2.0 inserts variables whose names contain '_' characters**
  Some compilers may not permit '_' characters to appear in variable names. This problem can be avoided by setting the option `AD_SEP` to a character other than '_'.

Chapter 9

ADIFOR 2.0 Options

This section provides short descriptions of each of the ADIFOR 2.0 options. Default values for options are presented within square brackets. Options that can be defined with a list of values are identified with a "*" superscript.

9.1 Mandatory Options

- **AD_DVARS***
  List of names of the FORTRAN 77 variables that contain the dependent variables of the function to be differentiated. Synonym for **AD_DVARS**.
- **AD_IVARS***
  List of names of the FORTRAN 77 variables that contain the independent variables of the function to be differentiated.
- **AD_OVARS***
  AD_OVARS is a synonym for **AD_DVARS**. At least one of **AD_OVARS** and **AD_DVARS** must be defined.
- **AD_PMAX** (MANDATORY if **AD_FLAVOR** is dense (default))
  Maximum number of independent variables of the function to be differentiated. The value of this option is compiled into each of the ADIFOR 2.0-generated dense derivative code files and is used as the first dimension of gradient objects for local and global variables.
- **AD_PROG**
  Name of composition file.
- **AD_TOP**
  Name of the top-level routine, the routine whose invocation is responsible for evaluating the function that is to be differentiated.

9.2 Other Options

- **AD_ALLSAVED** [0]
  If set to **true**, then ADIFOR 2.0 will treat all local and global variables in your program as being static variables, i.e., as if they had been listed in **SAVE** statements. Many FORTRAN 77 compilers treat all local and global storage as being static, which means that variables always retain their value between invocations of procedures. If your code assumes that all storage will be treated as static storage by your compiler, then you must set **AD_ALLSAVED** to true to generate correct derivative code.

- AD_CACHE [AD_cache]
  Name of directory in which ADIFOR 2.0 stores information about your program as analysis is performed. Permits incremental reanalysis of your code after changes to the source code or changes in options.
- AD_DUMP_CALLGRAPH [false]
  If set to true, causes ADIFOR 2.0 to print out a callgraph for the program.
- AD_EXCEPTION_FLAVOR [verbose]
  May be set to terse, verbose, counting, or performance to control level of exception handler error reporting. See Appendix B for more information.
- AD_FLAVOR [dense]
  ADIFOR 2.0 generates dense derivative code (i.e., expressing gradient objects loops as normal FORTRAN 77 loops) if AD_FLAVOR is set to dense, and sparse derivative code (i.e., calls to the SparsLinC library) if it is set to sparse.
- AD_OUTPUT_DIR [output_files/]
  Directory into which ADIFOR 2.0 places the augmented source code files.
- AD_PREFIX [g]
  Character that serves as initial character of gradient object names and derivative computing procedure names. For example, by default, the gradient object for foo is g_foo.
- AD_SCALAR_GRADIENTS [false]
  If set to true and AD_FLAVOR is "dense", then ADIFOR 2.0 ADIFOR 2.0 will generate code that assumes that g_pmax_ is 1. Executing this code provides an efficient means of generating $J * v$, where J is the Jacobian of the function being differentiated, and v is a vector.
- AD_SCRIPT
  Name of file containing additional definitions of bindings.
- AD_SEP [_]
  Character that it used to separate components of ADIFOR 2.0 generated names. If AD_SEP is changed to '$', then the gradient object for foo will be named g$foo.
- AD_SUPPRESS_LDG [false]
  If set to true and AD_FLAVOR is "dense", then ADIFOR 2.0 will generate code that assumes that all gradient objects are allocated with first dimensions set to g_pmax_. Leading dimension arguments will not be passed as parameters throughout derivative code. Use of this option may allow the generated code to be vectorized efficiently.
- AD_SUPPRESS_NUM_COLS [false]
  If set to true and AD_FLAVORis "dense", then ADIFOR 2.0 will generate code that assumes that g_p_ is g_pmax_, and hence does not pass g_p_ as a parameter throughout derivative code. Use of this option may allow the generated code to be vectorized efficiently.
- AD_TEMPLATE_DIR
  Specifies a directory in which to search for ADIntrinsic template files. Only a single additional directory may be specified. See Appendix B for more information.

Appendix A

Seed Matrix Initialization

A.1 Introduction

This appendix focuses on the proper and efficient use of ADIFOR-generated codes through detailed examination of seed matrix initialization for the following cases:

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

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

Note: The examples presented in Appendix A correspond to seed matrix initialization for the default or "nonsparse" flavor of ADIFOR 2.0 (see **AD_FLAVOR** in Chapter 9). The differences between the sparse and nonsparse ADIFOR 2.0-generated codes, which are discussed in Appendix C, impose differences in the mechanics of seed matrix initialization in each case (see Section C.4.4 for details). Nonetheless, the general seeding ideas presented here for the nonsparse case apply equally as well to the sparse case.

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

Our first example is adapted from Problem C2 in the STDTST set of test problems for stiff ODE solvers [10] and was brought to our attention by George Corliss of Marquette University. The routine **FCN2** computes the right-hand side of a system of ordinary differential equations $y' = yp = f(x, y)$ by calling a subordinate routine **FCN**:

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

      CALL FCN(X,Y,YP)
      RETURN
      END


C  File:  FCN.f

      SUBROUTINE FCN(X,Y,YP)

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

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

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

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

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

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

```
      subroutine g_fcn2(g_p_, m, x, y, g_y, ldg_y, yp, g_yp, ldg_yp)
C
C       ADIFOR: runtime gradient index
        integer g_p_
```

```
C       ADIFOR: translation time gradient index
        integer g_pmax_
        parameter (g_pmax_ = 20)
C       ADIFOR: gradient iteration index
        integer g_i_
C
        integer ldg_y
        integer ldg_yp
        integer n
        double precision x, y(m), yp(m)
        integer id, iwt
        double precision w(20)
        common /stcom5/ w, iwt, n, id
C
C       ADIFOR: gradient declarations
        double precision g_y(ldg_y, m), g_yp(ldg_yp, m)
        if (g_p_ .gt. g_pmax_) then
          print *, "Parameter g_p is greater than g_pmax."
          stop
        endif
        call g_fcn(g_p_, x, y, g_y, ldg_y, yp, g_yp, ldg_yp)
        return
      end


      subroutine g_fcn(g_p_, x, y, g_y, ldg_y, yp, g_yp, ldg_yp)
C
C       ADIFOR: runtime gradient index
        integer g_p_
C       ADIFOR: translation time gradient index
        integer g_pmax_
        parameter (g_pmax_ = 20)
C       ADIFOR: gradient iteration index
        integer g_i_
C
        integer ldg_y
        integer ldg_yp
C       ROUTINE TO EVALUATE THE DERIVATIVE F(X,Y) CORRESPONDING TO THE
C       DIFFERENTIAL EQUATION:
C       DY/DX = F(X,Y) .
C       THE ROUTINE STORES THE VECTOR OF DERIVATIVES IN YP(*). THE
C       DIFFERENTIAL EQUATION IS SCALED BY THE WEIGHT VECTOR W(*)
C       IF THIS OPTION HAS BEEN SELECTED (IF SO IT IS SIGNALLED
C       BY THE FLAG IWT).
        double precision x, y(20), yp(20)
        integer id, iwt, n
        double precision w(20)
        common /stcom5/ w, iwt, n, id
        double precision sum, cparm(4), ytemp(20)
        integer i, iid
        data cparm /1.d-1, 1.d0, 1.d1, 2.d1/
C
C       ADIFOR: gradient declarations
        double precision g_y(ldg_y, 20), g_yp(ldg_yp, 20)
        double precision g_sum(g_pmax_), g_ytemp(g_pmax_, 20)
        if (g_p_ .gt. g_pmax_) then
          print *, "Parameter g_p is greater than g_pmax."
          stop
        endif
        if (iwt .lt. 0) then
          goto 40
        endif
        do 99999, i = 1, n
C             ytemp(i) = y(i)
          do g_i_ = 1, g_p_
            g_ytemp(g_i_, i) = g_y(g_i_, i)
          enddo
          ytemp(i) = y(i)
C             y(i) = y(i) * w(i)
          do g_i_ = 1, g_p_
            g_y(g_i_, i) = w(i) * g_y(g_i_, i)
          enddo
          y(i) = y(i) * w(i)
20        continue
99999   continue
40      iid = mod(id, 10)
C       ADAPTED FROM PROBLEM C2
C       yp(1) = -y(1) + 2.d0
        do g_i_ = 1, g_p_
          g_yp(g_i_, 1) = -g_y(g_i_, 1)
        enddo
        yp(1) = -y(1) + 2.d0
C       sum = y(1) * y(1)
        do g_i_ = 1, g_p_
          g_sum(g_i_) = y(1) * g_y(g_i_, 1) + y(1) * g_y(g_i_, 1)
        enddo
        sum = y(1) * y(1)
        do 99998, i = 2, n
C             yp(i) = -10.0d0 * i * y(i) + cparm(iid - 1) * (2 ** i) * sum
          do g_i_ = 1, g_p_
            g_yp(g_i_, i) = cparm(iid - 1) * (2 ** i) * g_sum(g_i_) + -1
     *0.0d0 * i * g_y(g_i_, i)
          enddo
          yp(i) = -10.0d0 * i * y(i) + cparm(iid - 1) * (2 ** i) * sum
C             sum = sum + y(i) * y(i)
          do g_i_ = 1, g_p_
            g_sum(g_i_) = g_sum(g_i_) + y(i) * g_y(g_i_, i) + y(i) * g_y
     *(g_i_, i)
          enddo
          sum = sum + y(i) * y(i)
50        continue
99998   continue
        if (iwt .lt. 0) then
          goto 680
        endif
        do 99997, i = 1, n
C             yp(i) = yp(i) / w(i)
          do g_i_ = 1, g_p_
            g_yp(g_i_, i) = (1 / w(i)) * g_yp(g_i_, i)
          enddo
          yp(i) = yp(i) / w(i)
C           y(i) = ytemp(i)
          do g_i_ = 1, g_p_
            g_y(g_i_, i) = g_ytemp(g_i_, i)
          enddo
          y(i) = ytemp(i)
660       continue
99997   continue
680     continue
        return
      end
```

The derivative objects g_y and g_yp are declared as matrices with 20 columns (since both y and yp were declared as vectors of length 20) and leading dimension ldg_y and ldg_yp, respectively. The parameter g_p denotes the actual length of the gradient objects in a call to g_fcn2. Since Fortran 77 does not allow dynamic memory allocation, derivative objects for local variables are statically allocated with leading dimension pmax, whose value was selected by the user during the invocation of ADIFOR. A variable and its associated derivative object are treated in the same fashion; that is, if x is a function parameter, so is g_x. Derivative objects corresponding to locally declared variables or variables in common blocks are declared locally or in common blocks as well.

Subroutine g_fcn2 relates to the Jacobian

$$J_{yp} = \begin{pmatrix} \frac{\partial yp_1}{\partial Y_1} & \cdots & \frac{\partial yp_1}{\partial Y_m} \\ \vdots & & \vdots \\ \frac{\partial yp_m}{\partial Y_1} & \cdots & \frac{\partial yp_m}{\partial Y_m} \end{pmatrix}$$

as follows: Given input values for g_p_, m, x, y, g_y, ldg_y, and ldg_yp, the routine g_fcn2 computes both yp and g_yp, where

$$\texttt{g\_yp(1:g\_p\_,1:m)} = (J_{yp}(\texttt{g\_y(1:g\_p\_,1:m)}^T))^T.$$

The superscript T denotes matrix transposition. The user must allocate g_yp and g_y with leading dimensions ldg_yp and ldg_y that are at least g_p_. While the implicit transposition may seem awkward at first, this is the only way to handle assumed-size arrays (like real a(*)) in subroutine calls.

Assume that m and g_p are 20 and that ldg_yp and ldg_y are at least 20. Then we can compute the derivative matrix J_{yp} simply by initializing g_y to the identity:

```
**************
* Approach 1 *
**************
      DO 10 I = 1, M
         DO 5 J = 1, M
            G_Y(I,J) = 0.0D
    5    CONTINUE
         G_Y(I,I) = 1.0D0
   10 CONTINUE
      call g_fcn2(20, m, x, y, g_y, ldg_y, yp, g_yp, ldg_yp)
```

On exit from g_fcn2, the variable g_yp contains the transpose of the Jacobian J_{yp}. Note that for this program to work, g_fcn2 must have been generated with AD_PMAX at least 20.

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

```
**************
* Approach 2 *
**************
      DO 10 I = 1, M
      *
      *       initialize first row of G_Y to i-th unit vector
      *
              DO 5 J = 1, M
                 G_Y(1,J) = 0.0D
            5 CONTINUE
              G_Y(1,I) = 1.0D0
      *
      *       call ADIFOR-generated derivative code
      *
              call g_fcn2(1, m, x, y, g_y, ldg_y, yp, g_yp, ldg_yp)
      *
      *       store ith column of the Jacobian in ith row of Jactrans array
      *
              DO 15 J = 1,M
                 JACTRANS(I,J) = G_YP(1,J)
           15 CONTINUE
           10 CONTINUE
```

Even though g_yp(i,j) as computed in Approach 1 equals jactrans(i,j) computed in Approach 2, the second method is significantly less efficient. This inefficiency arises from the fact that the value of yp itself is computed once in the first approach, but m times in the second approach. Thus, it is usually best to compute as large a slice of the Jacobian as memory restrictions will allow. However, in this case, AD_PMAX = 1 is sufficient, and, as a result, the memory requirements of the ADIFOR-generated code can be expected to be more modest, roughly 1/20th of the memory requirements of the previous code. In this fashion, the ADIFOR interface provides a mechanism for accomodating memory/runtime tradeoffs. An example of a parallel "derivative stripmining" technique based on this approach is presented in [6].

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

The second example involves a code that models adiabatic flow [19], a commonly used module in chemical engineering. This code models the separation of a pressurized mixture of hydrocarbons into liquid and vapor components in a distillation column, where pressure (and, as a result, temperature) decrease. This example was communicated to us by Larry Biegler of Carnegie-Mellon University.

In its original version, the top-level subroutine

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

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

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

copying the values passed in those arguments into the proper common blocks in aifl. As our first example, assume that we are interested in $\frac{\partial liquid}{\partial feed}$ and $\frac{\partial vapor}{\partial feed}$ [1]. In this case, ADIFOR generates

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

```
      subroutine g_aifl(g_p_, kf, feed, g_feed, ldg_feed, pressure,
     $                  liquid, g_liquid, ldg_liquid,
     $                  vapor, g_vapor, ldg_vapor)
      integer g_p_, kf, ldg_feed, ldg_liquid, ldg_vapor
      real feed(*), g_feed(ldg_feed,*), pressure(*),
     $     liquid(*), g_liquid(ldg_liquid,*),
     $     vapor(*), g_vapor(ldg_vapor,*)
```

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

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

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

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

```
      subroutine g_aifl(g_p_, kf, feed, g_feed, ldg_feed,
     $                  pressure, g_pressure, ldg_pressure,
     $                  liquid, g_liquid, ldg_liquid, vapor)
```

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

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

is a 6×9 matrix. ADIFOR computes

$$\texttt{g\_liquid} = \left(J_{liquid} \begin{pmatrix} \texttt{g\_feed}^T \\ \texttt{g\_pressure}^T \end{pmatrix} \right)^T .$$

If we wish to compute the whole Jacobian J, then

$$\begin{pmatrix} \texttt{g\_feed}^T \\ \texttt{g\_pressure}^T \end{pmatrix}$$

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

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

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

From the previous discussion, ADIFOR may seem to be well suited for computing dense Jacobian matrices, but rather expensive for sparse Jacobians. A primary reason is that the forward mode of automatic differentiation upon which ADIFOR is mainly based (see [4]) requires roughly g_p_ operations for every assignment statement in the original function. Thus, if we compute a Jacobian J with n columns by setting g_p_ $= n$, its computation will require roughly n times as many operations as the original function evaluation, independent of whether J is dense or sparse. However, it is well known [8, 12] that the number of function evaluations that are required to compute an approximation to the Jacobian by finite differences can be much less than n if J is sparse. Fortunately, the same idea can be applied to greatly reduce the running time of ADIFOR-generated derivative code as well. This section suggests a technique for exploiting sparsity in derivative computions *if the sparsity pattern is known a priori.* Appendix C describes the the SparsLinC library, which, in conjunction with ADIFOR 2.0, allows exploitation of sparsity without a priori knowledge, and even computes the sparsity pattern of the Jacobian as a byproduct of the derivative computation.

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

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

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

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

That is, the function f_1 depends only on x_1, f_2 depends only on x_1 and x_4, and so on. The key idea in sparse finite difference approximations is to identify *structurally orthogonal* columns j_i of J– that is, columns whose inner product is zero, independent of the value of x. In our example, columns 1 and 2 are structurally orthogonal, and so are columns 3 and 4. This means that the set of functions that depend nontrivially on x_1, and the set of functions that depend nontrivially on x_2 are disjoint.

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

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

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

As a more realistic example, we consider the swirling flow problem, part of the MINPACK-2 test problem collection [2], which was made available to us by Jorge Moré of Argonne National Laboratory. Here we solve a nonlinear system of equations $F(x) = 0$ for $F : \mathbf{R}^n \rightarrow \mathbf{R}^n$. The swirling flow code has the form

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

Like all codes in the MINPACK-2 test collection, it is set up to compute the function values (in `fvec`) and, if desired, the analytic first-order derivatives (in `fjac`) as well. The vectors `x` and `fvec` are of size `nxmax` =

14*nint. For example, for nint = 4, the Jacobian of F is of size nxmax = 56 and has the structure shown in Figure A.1.

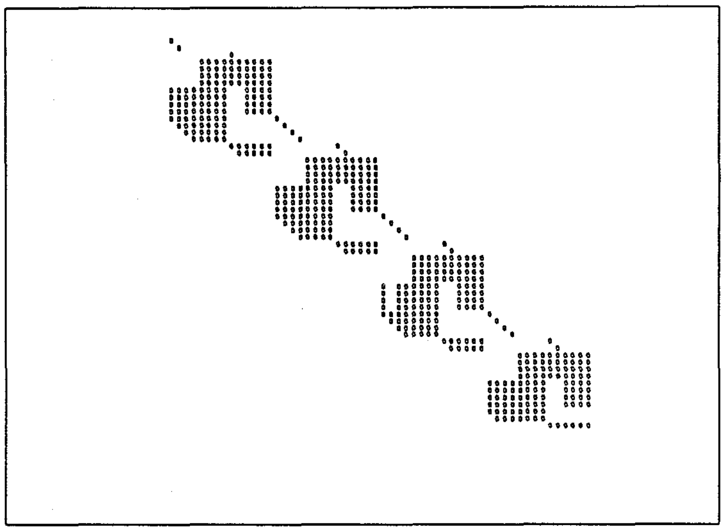

Figure A.1. Structure of the swirling flow Jacobian, n = 56

The derivative subroutine produced by ADIFOR is

```
      subroutine g_dswrl3 (g_p_, nxmax, x, g_x, ldg_x,
     $                     fvec, g_fvec, ldg_x,
     $                     fjac, ldfjac, 1, eps, nint)
```

If we initialize g_x to a 56×56 identity matrix, and let g_p_=56, and if ldg_x is at least 56, then on exit from g_dswrl3, g_fvec will contain the transpose of $\frac{\partial F}{\partial x}$, stored as a dense matrix. As it turns out, less than 7 % of the total operations performed with gradient objects in the ADIFOR code involve nonzeros. On the other hand, by using a graph-coloring algorithm designed to identify structurally orthogonal columns (we used the one described in [7]), we can determine that this Jacobian can be grouped into 14 sets of structurally orthogonal columns, independent of the size of the problem. In our example, columns 1, 16, 31, and 51 were in the first group; columns 2, 17, 37, and 43 were in the second group; and so on. We can take advantage of this fact by initializing the first column of g_x^T such that it has 1.0 in rows 1, 16, 31, and 51; by initializing the second column of g_x^T such that it has 1.0 in rows 2, 17, 37, and 43; and so on. The structure of g_x^T thus initialized is shown in Figure A.2 together with the resulting compressed Jacobian g_fvec^T. Note that instead of g_p_= 56 we now can get by with g_p_= 14, a sizeable reduction in cost.

Assuming that color(i) is the "color" of column i of the Jacobian and that nocolors is the number of colors (in our example we had 14 colors), the following code fragment properly initializes g_x, calls g_dswrl3 to compute the compressed Jacobian, and then extracts the Jacobian.

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

      call g_dswrl3 (nocolors, nxmax, x, g_x, pmax,
     +                     fvec, g_fvec, pmax,
```

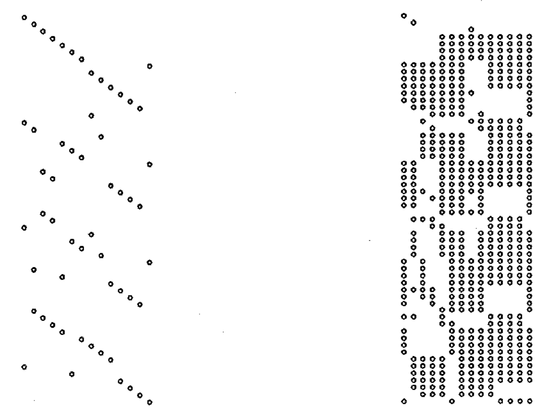

Figure A.2. Left: Structure of $(\mathtt{g\_x})^T$ Right: Structure of $(\mathtt{g\_fvec})^T$

```
     +                fjac, ldfjac, 1, eps, nint)
c     job = 1 indicates that only the function value is to be computed in
c             dswrl3.


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

      do i = 1, n
        do j = 1, n
          if (nonzero(j,i)) then
            jac(j,i) = g_fvec(color(i),j)
          else
            jac(j,i) = 0.0
          endif
        enddo
      enddo
```

Experimental results using this approach on a suite of problems from the MINPACK test set collection are presented in [3].

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

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

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

c  Subroutine Arguments
```

```
c      ==> n       number of observations
c      ==> m       number of columns in independent variable
c      ==> np      number of parameters
c      ==> nq      number of responses per observation
c      ==> beta    current values of parameters
c      ==> xplusd  current value of independent variable, i.e., x + delta
c      ==> ldxpd   leading dimension of xplusd
c     <==  f       predicted function values
c      ==> ldf     leading dimension of f

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

      double precision par(20),fn(2)

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

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

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

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

  100 continue
      return
      end


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

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

c  compute first of multi-response observations

   fn(1) =   beta(1)
    +                    + beta(2)*xplusd(1)
```

```
     +                    + beta(3)*xplusd(2)
     +                    + beta(4)*xplusd(1)*xplusd(2)

c  compute second of multi-response observations

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

      return
      end
```

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

A.5.1 Approach 1 – Generate derivatives only for fnc

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

```
subroutine g_fnc(g_p_, x, g_x, ldg_x, fn, g_fn, ldg_fn)
integer m, np, nq
parameter( np = 8, m = 2, nq = 2)
double precision x(np+m), fn(nq), g_x(ldg_x,np+m), g_fn(ldg_fn,nq)
```

If inside fun we replace the call to fnc with a call to g_fnc, always initializing g_x to a 10×10 identity matrix before the call, then

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

and

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

Closer inspection reveals that the 10×2 array g_fn always has the following structure (numbers are used to uniquely identify nonzero elements):

$$\begin{pmatrix} 1 & 0 \\ 2 & 0 \\ 3 & 0 \\ 4 & 0 \\ 0 & 5 \\ 0 & 6 \\ 0 & 7 \\ 0 & 8 \\ 9 & 10 \\ 11 & 12 \end{pmatrix}.$$

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

$$\texttt{g\_x} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

which results in

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

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

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

A.5.2 Approach 2 – Generate derivatives for fun

An alternative method of applying ADIFOR is to process subroutine fun. ADIFOR detects the interprocedural data dependence between fun and fnc and therefore generates g_fun as well as g_fnc, with g_fnc called properly within g_fun. We obtain

```
      subroutine g_fun(g_p_,n,m,np,nq,beta,g_beta,ldg_beta,
     $                 xplusd,g_xplusd,ldg_xplusd,ldxpd,f,g_f,ldg_f,ldf)
      integer g_p_, n, m, np, nq, ldg_beta,ldg_xplusd,ldxpd,ldg_f,ldf
      double precision beta(np), g_beta(ldg_beta,np),
     $                 xplusd(ldxpd,m), g_xplusd(ldg_xplusd,ldxpd,m),
     $                 f(ldf,nq), g_f(ldg_f,ldf,nq)
```

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

$$\begin{pmatrix} \texttt{g\_beta}^T \\ \texttt{g\_xplusd}^T \end{pmatrix}$$

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

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

```
      g_p_ = np + m*n
      do 44 i = 1, np
        do 144 j = 1, g_p_
          g_beta(j,i) = 0.0
 144    continue
        g_beta(i,i) = 1.0
  44  continue
      do 45 i = 1, m
        do 145 j = 1, n
          do 245 k = 1, g_p_
            g_xplusd(k,j,i) = 0.0
 245      continue
          g_xplusd(np+((i-1)*n)+j,j,i) = 1.0
 145    continue
  45  continue
```

When initialized in this manner, ADIFOR computes

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

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

```
      g_p = np + m*n
      . . .
      g_xplusd(np+((i-1)*n)+j,j,i) = 1.0
```

to the much simpler

```
      g_p = np + m
      . . .
      g_xplusd(np+i,j,i) = 1.0
```

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

$$\text{g\_beta} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and g\_xplusd[n]} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

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

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

$$\text{g\_beta} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \text{ and } \text{g\_xplusd}[n] = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

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

```
c n=63, m=2, np=8
      halfnp = 4
      g_p_ = 4 + m
      do 44 i = 1, halfnp
        do 144 j = 1, g_p_
          g_beta(j,i) = 0.0
          g_beta(j,i+halfnp) = 0.0
 144    continue
        g_beta(i,i) = 1.0
        g_beta(i,i+halfnp) = 1.0
  44  continue
      do 45 i = 1, m
        do 145 j = 1, n
          do 245 k = 1, g_p_
            g_xplusd(k,j,i) = 0.0
 245      continue
          g_xplusd(halfnp+i,j,i) = 1.0
 145    continue
  45  continue
```

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

A.6 Computing Gradients of Partially Separable Functions

A particular class of functions that arises often in optimization contexts is that of the so-called partially separable functions [9, 14, 15, 16, 17]. That is, we have a function $f : \mathbf{R}^n \rightarrow \mathbf{R}$ that can be expressed as

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

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

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

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

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

      fval = 0

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

      ......

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

      return
      end
```

If we submit `f` to ADIFOR, it generates

```
      subroutine g_fn(g_p_,n,x,g_x,ldg_x,fval,g_fval,ldg_fval).
```

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

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

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

we have the identities

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

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

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

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

      call g(n,x,gval)

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

      return
      end
```

It may not appear that we have gained anything, since J_g is $nf \times n$. If we initialize g_x in

```
      subroutine g_g(g_p_,n,x,g_x,ldg_x,gval,g_gval,ldg_gval)
```

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

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

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

and each of the f_i's depends only on n_i of the x's. By using the graph coloring techniques described in Section A.4, we can compute J_g at a cost that is proportional to the number of columns in the compressed J_g, and then add up its (sparse) rows. As a result, we can compute ∇f at a cost that is potentially much less than n times the evaluation of f. Alternatively, we can employ the SparsLinC library (see Appendix C), which will exploit sparsity even if the Jacobian contains a few dense rows (in this case, its chromatic number is n, and nothing has been gained).

Appendix B

ADIntrinsics 1.0: Exception Handling Support for ADIFOR 2.0

B.1 Introduction

B.1.1 What Is an Exception?

In ADIFOR parlance, an "exception" is an event that occurs when an elementary function is evaluated at a point where the function result is defined, but the derivative is not. For instance, the square root of zero is zero, but the derivative of the square root function at zero is not defined.

For most functions, there are several reasonable interpretations of what should be done when an exception occurs. ADIFOR 2.0 is programmed to choose one of those, but only a human can decide whether this choice is the correct one for any given instance. When an exception occurs, you should examine your code to make sure that ADIFOR 2.0 has done something reasonable in your case.

A discussion for different approaches for handling exceptions as well as the choice of default values for exceptional situations is discussed in [5].

B.1.2 What Code Is Needed?

If you are not interested in customizing the exception handler in your program, you need only make sure that you call the routine **ehrpt** (for **e**xception **h**andler **r**e**p**or**t**) to report any exceptions that may have occurred. This call should occur after all of the ADIFOR 2.0-generated code has run. A good place for it is right after your top-level subroutine.

```
program main
[...]
call ADIFOR_GENERATED_CODE ()
[...]
call EHRPT
return
end
```

In addition, one file and one library must be added to your link step. These are

1. the file `$AD_HOME/lib/ReqADIntrinsics-$AD_ARCH.o`, and

2. the library `$AD_HOME/lib/libADIntrinsics-$AD_ARCH.a`.

A sample link line may appear as follows:

```
f77 -o executable sub.o main.o \
       $AD_HOME/lib/ReqADIntrinsics-$AD_ARCH.o \
       -L$AD_HOME/lib -lADIntrinsics-$AD_ARCH
```

B.2 Redirecting Exception Handler Output

To direct the exception handler output to a different unit, open the unit in your driver program, and then call **ehsup** with two parameters: `-1`, and then the unit number. The driver is also responsible for closing this unit before the program terminates. Failure to do so may result in a loss of output that has been buffered but not written to the file.

```
call ehsup (-1, UNIT-NUMBER)
```

A segment of the user code might resemble this fragment.

```
open (UNIT=13, FILE='adifor-errors.out')
call ehsup (-1, 13)
[... Useful Work ...]
close(13)
```

B.3 Purse and Exception Handler Templates

To provide a flexible means of customization of the exception handling mechanism in ADIFOR 2.0, we have split derivative code generation into two components – ADIFOR 2.0 generates code containing invocations of "templates," which are then expanded into explicit Fortran code by the purse postprocessor. To users, these two components look like a single step, since ADIFOR 2.0 invokes `purse` directly.

By default, `purse` expands templates based on the set of template definitions stored in files in the directory `$AD_HOME/templates`. If necessary, users can create new templates or modify default ones as necessary. In order to modify default templates, all that is necessary is to place the modified template into a directory (say '`new-templates`', or the current directory, '`.`'). ADIFOR 2.0 must then be instructed where to find the new template file by setting the **AD_TEMPLATE_DIR** option to be the name of the directory containing the new template file.

The user can customize the exception handling in several different ways:

1. By directing ADIFOR 2.0 to use the verbose, counting, terse, or performance exception handling mode for error reporting.
2. By setting the exception handling mode on a fine-grained level.
3. By ignoring exceptions in a particular region of code.
4. By changing the values returned by the exception handler.
5. By overriding the template expansion for a particular intrinsic with a user's own template, locally.
6. By overriding the template expansion for a particular intrinsic with the user's own template, globally.

Each of the mechanisms is now described.

B.3.1 Exception Handler Modes

The exception handler operates in four modes: verbose, counting, terse, and performance. In verbose mode, every time an exceptional condition occurs, a message is written to the program's error unit (unit number zero, which usually outputs to the screen) indicating the function, the arguments to the function, and the file name and line number containing this function evaluation. This information allows one to track down exactly where the exception is occurring and decide whether it is generating appropriate results. Counting, terse, and performance modes provide a decreasing amount of information about exceptions that occur.

Counting mode maintains a running total of each *type* of exception that occurs, as shown in Figure B.1.

```
Exception(s) occurred evaluating ABS              : 100 times.
Exception(s) occurred evaluating POWER: df/dx     : 5 times.
Exception(s) occurred evaluating ACOS first deriv : 17 times.
```

Figure B.1. Counting Mode Error Report

Terse mode indicates whether any exceptions of a given type occurred. This mode may be useful for vectorizing compilers, where the recurrence required for counting may inhibit vectorization. A sample terse mode output is given in Figure B.2.

```
Exception(s) occurred evaluating ABS
Exception(s) occurred evaluating POWER: df/dx
Exception(s) occurred evaluating ACOS first deriv
```

Figure B.2. Terse Mode Error Report

Performance mode contains no exception checking at all. One should only use performance mode after running the code with another mode and being sure that no exceptions occur. Performance mode assumes that no exceptions occur, and may not give correct derivative information if they do occur. No report is made, since no exceptions are tracked.

The exception handling mode may be chosen at the time ADIFOR 2.0 is run by setting the `AD_EXCEPTION_FLAVOR` variable to one of: `performance`, `terse`, `counting`, or `verbose`.

B.3.2 Fine-Grained Control of Exception Handler Modes

Fine-grained control over exception handler modes is achieved by embedding directives in the user's code. These directives are reasonably simple to use, but a brief explanation of their syntax and some warnings about their use is appropriate.

Note: These directives have a reasonably intuitive syntax:

- any comment character (`C`, `c`, or `*`) may be used to begin the comment line;
- spaces cannot appear in the middle of a keyword, but may appear around parentheses and commas;
- the directives can appear in upper or lower case, as can the keywords (arguments) given; and
- zero or more whitespace characters may appear between the comment character and the beginning of the directive, but no spurious "garbage" should appear in the line, even after column 72.

Warning: Directives affect only the parts of the program that are *literally* after them. In particular, a directive cannot change the mode in which an invoked procedure runs. The example below shows *incorrect* usage of the AD_EXCEPTION_LEVEL directive.

```
C       This is an incorrect use of the AD_EXCEPTION_LEVEL directive.
C       it has no effect on the subroutine "slow_func".
C
C       AD_EXCEPTION_LEVEL(PERFORMANCE)
        call slow_func
C       AD_EXCEPTION_LEVEL(DEFAULT)
```

The verbosity level can be dynamically set with the AD_EXCEPTION_LEVEL() directive. Valid levels are verbose, counting, terse, performance, and "default," which restores the exception level to the one with which ADIFOR 2.0 was run. For example, to guarantee verbose exception reporting around a certain region, the user might use the following code:

```
C       AD_EXCEPTION_LEVEL(VERBOSE)
        [... Interesting Code Here ...]
C       AD_EXCEPTION_LEVEL(DEFAULT)
```

Warning: Terse mode is incompatible with both counting mode and verbose mode. Do not switch from verbose or counting mode to terse mode anywhere in your program. Doing so will cause the summary information reported to be incorrect.

B.3.3 Ignoring Exceptions in a Region

To ignore exceptions in a region, bracket the region with the directives AD_EXCEPTION_BEGIN_IGNORE and AD_EXCEPTION_END_IGNORE. "Ignoring" exceptions simply means that no exceptional information is printed out; it does not mean that the exception handler is disabled. Disabling the exception handler (that is, using performance mode) is not an option, because at exceptional points the performance mode may return a value different than that returned by the exception handler.

```
C       AD_EXCEPTION_BEGIN_IGNORE()
        [... Exceptions to be Ignored Here ...]
C       AD_EXCEPTION_END_IGNORE()
```

Warning: These directives *do not nest.* This means that *any* AD_EXCEPTION_END_IGNORE cancels *all* previous AD_EXCEPTION_BEGIN_IGNORE commands, regardless of how many preceded the end ignore.

Here is an example showing how the ignore directives do not nest.

```
C       AD_EXCEPTION_BEGIN_IGNORE()
        [... Exceptions are Ignored Here ...]
C       AD_EXCEPTION_BEGIN_IGNORE()
        [... Exceptions are Ignored Here ...]
C       AD_EXCEPTION_END_IGNORE()
        [... Exceptions are REPORTED Here ...]
C       AD_EXCEPTION_END_IGNORE()
        [... Exceptions Continue to be Reported Here ...]
```

Note: Currently, the "ignore" mode is implemented by placing the exception handler in counting mode for the given region.

B.3.4 Setting Exceptional Values

It is possible to override the default values for the exceptions. This overriding is precision-specific, so it is done through a routine **ehsup***, where * is one of s, d, c, or z, for single, double precision, complex, or double complex, respectively.

To override an exceptional value, one needs to know two facts: the integer that represents the intrinsic for which the exception is occurring, and the integer "offset" of the exceptional condition whose return value is to be altered. The integer representing the intrinsic can be found in Table B.1. Almost all intrinsics have only a single exceptional condition, and therefore have an offset of one. Those that do not follow this rule are discussed below.

Before discussing multiple exceptional conditions, let us examine a brief example of setting an exceptional value. Suppose one wishes to change the exceptional value of ABS at zero (for both real and double precision) so that the partial derivative of ABS(x) with respect to x at zero is one. First, one would look in Table B.1 to find that the integer representing ABS is 3. ABS has only one exceptional value, so the offset used to set $\partial(\texttt{ABS}(x))/\partial x$ is one. Having figured out all of this information, one would use the following two calls to set the desired partials of ABS.

```
C       Set single precision partial of abs
        call ehsups (3,1,1.0e0)
C       Set double precision partial of abs
        call ehsupd (3,1,1.0d0)
```

The SQRT4CABS "function" is a dummy intrinsic generated by ADIFOR 2.0 to handle the complex ABS function. Let $z = x + iy$. The complex ABS(z) function is rewritten as

$$\text{abs}(z) = \text{SQRT4CABS}(x^2 + y^2)$$

By default, SQRT4CABS has the same exceptional behaviour as SQRT.

Multiple Exceptional Values

As mentioned above, not all functions have only one exceptional value. The following functions have more than one possible exception: SQRT, ASIN, ACOS, and POWER (POWERBASE, POWEREXP, POWERBOTH). For the first three, SQRT, ASIN, and ACOS, the user can set both the first and second derivatives at the exceptional point. An offset of one is used for the first derivative and two for the second derivative. The POWER operator is more complicated, so it is explained in the next section.

Multiple Exceptional Values: The POWER Operator

The POWER operator for real arithmetic has two exceptional conditions, as listed below.

1. f'_x for $x = 0$, $0 < y < 1$, and
2. f'_y for $x < 0$, or $x = 0$ and $y = 0$.

The first condition may be set using either POWERBOTH with an offset of one, or POWERBASE with an offset of one. The second may be set by using POWERBOTH with an offset of two, or POWEREXP with an offset of one.

The POWER operator for complex arithmetic has two different exceptional conditions. These are

1. an exception occurred at 0^0, or
2. an exception occurred at 0^y with $\text{Re}(y) < 0$.

These values are returned for all derivatives (both first and second) at the point where the exception occurred. The values can be set using any one of POWERBOTH, POWERBASE, or POWEREXP; they are offset one and two, respectively.

Intrinsic	Numerical Value
AINT	1
ANINT	2
DNINT	2
ABS	3
MOD	4
SIGN	5
DIM	6
MAX	7
MIN	8
SQRT	9
POWERBASE	10
POWEREXP	11
POWERBOTH	12
ASIN	13
ACOS	14
SQRT4CABS	15

Table B.1. Intrinsic Functions

B.3.5 Overriding a Specific Instance of a Template

The experienced user may wish to replace a given instance of an intrinsic's exception handling template with their own. To do so, provide the name of the intrinsic to be overriden and the name of the new exception handling template to be used. For example:

```
C       AD_EXCEPTION_OVERRIDE_INTRINSIC_BEGIN(SIN,MYSIN)
[...AD Code...]
C       AD_EXCEPTION_OVERRIDE_INTRINSIC_END(SIN)
```

You can either bracket the code to be overriden with **BEGIN** and **END** directives as above, or use a special "**ONCE**" directive to indicate that the overriding should only happen to the next matching intrinsic encountered.

```
C       AD_EXCEPTION_OVERRIDE_INTRINSIC_ONCE(SIN,MYSIN)
```

In addition to using this directive, you must create an exception handler template file for the new intrinsic, and use the **AD_TEMPLATE_DIR** option to ADIFOR 2.0 to tell it to search the named directory before searching the default directory. See the next section for more information about writing templates.

B.3.6 Replacing All Instances of a Template

To replace all instances of a given template with another, create a template file with the same name as the one to be overridden and use the **AD_TEMPLATE_DIR** option to ADIFOR 2.0 to tell it to search the named directory before searching the default directory. See B.4 for more information writing templates.

B.4 Writing Templates

Writing a template is a relatively simple process; it may well be worth your while to write a template in order to change the behavior of the exception handler at some point.

B.4.1 Filenames of the Template Files

Each intrinsic has one or two template files: `<intrinsic>.T` and (if the intrinsic supports complex arguments) `c<INTRINSIC>.T`. The `*.T` files contain the template for real and double precision, and the `c*.T` files contain the template for complex and double complex code. Note that many routines are meaningful only in real arithmetic and do not have complex counterparts.

B.4.2 Format of the Template Files

Each of these files will be written in a very specific way so that the code will be readable to a human but simple for `purse` to modify. The code should be legal Fortran 77: it must be indented to column 7, and any directives should be expressed in comments. The function inputs will be denoted by `x` and `y` in the code, and the function result will be denoted by `z`. First derivatives with respect to `x` and `y` will be `fx` and `fy`, respectively. Second derivatives will be `fxx`, `fxy`, and `fyy`. No other variables can be used in the code, since it will be embedded in an unknown context.

As an example, this is the template derivative code file for the square root function for real numbers. Note that it is correctly indented, legal (looking) Fortran 77.

```
C     PERFORMANCE_FVAL
      z = sqrt(x)
      if ( x .gt. 0.0 ) then
C     PERFORMANCE_FIRST
         fx = 1.0 / (2.0 * z)
C     PERFORMANCE_SECOND
         fxx = fx / (2.0 * x)
      else
         call EXCEPTION_HANDLER
      endif
```

Most of the details should be obvious from this example. To generate performance mode code (with no error checking), we merely insert the statements that are preceded by a comment with the string "PERFORMANCE" in it. The phony call to "EXCEPTION_HANDLER" is transformed into the appropriate exception handler call by `purse`.

In a few rare cases, we actually want to generate different code for first- and second-order derivatives, to avoid recomputation of some quantities. In this case, we use the C preprocessor-style `#ifdef FIRST`, `#ifdef SECOND`, and `#ifdef PERFORMANCE` directives. (The `#else` directive will switch between the cases, as one would expect.) Since the file is not really processed with the C preprocessor, most of the niceties have been omitted as the following warning makes clear. An example of a complicated template file using these features is included in Appendix B.5.

> *Warning*: These C preprocessor style directive *cannot be nested*, and the only things that are permissible to check are `FVAL`, `FIRST`, `SECOND`, and `PERFORMANCE`.
>
> *Note*: Everything within an `#ifdef PERFORMANCE` branch is emitted in performance mode. The `#else` branch is emitted in an ordinary mode. If one tests `#ifndef PERFORMANCE`, the `#else` branch (if any) is emitted in performance mode.

It is clear that we can generate second-order derivative code merely by copying the given code and making a few substitutions. To generate first-order derivative code, we omit all lines containing the strings `fxx`, `fxy`, or `fyy`. To generate function values, we put ourselves in performance mode and omit all first- and second-order partials.

B.4.3 Typing Issues

To help compilers get the correct type for numerical constants, such as `1.0`, that appear in the template code, we allow a "macro" `TYPE` to appear in the code. It will expand to give the correct precision to the number it contains. For example, given the code below,

```
        fx = TYPE(1.0) / ( x*x + y*y )
```

for a double-precision expansion, we output

```
        fx = 1.0d0 / (x*x + y*y)
```

Whitespace is allowed after the TYPE and before and after the argument. For safety, we do *not* expand a lowercase type, but abort with an error. No parentheses are allowed in the parameter, but none should appear in a number in any case.

> *Note*: No complex TYPE() parameters are permitted at this time. When a TYPE macro is expanded in a complex context, it becomes a complex number whose real part is the value given and whose imaginary part is zero. Also note that it is not possible to "fake" a complex number not of this form by writing
>
> ```
>     (TYPE(1.0), TYPE(7.0))
> ```
>
> because if this is expanded in a complex context, it will become:
>
> ```
>     ((1.0, 0.0), (7.0, 0.0))
> ```

B.5 Examples of Complicated Template Files

Figure B.3 shows an example of a template file that produces different code for first and second derivatives.

Figure B.4 shows an example of a template file with C preprocessor style directive that control when performance-mode code should be output. Recall that if a line is not preceded by a comment containing the string PERFORMANCE, and it is not in the "true" branch of a PERFORMANCE ifdef, it is not emitted in performance mode. This means that if we were to move the #ifdef] so that it just enclosed the exception handler call, every other line would need to be preceded by a comment containing the string "PERFORMANCE". An example of this style is given in Figure B.5.

```
C     PERFORMANCE
      z = acos(x)

#ifdef FIRST
      if ( abs(x) .lt. TYPE(1.0) ) then
C     PERFORMANCE
         fx = TYPE(-1.0) / sqrt ((TYPE(1.0)-x)*(TYPE(1.0)+x))
      else
         call EXCEPTION_HANDLER
      endif
#endif /* FIRST */

#ifdef SECOND
      if ( abs(x) .lt. TYPE(1.0) ) then
C     Use fxx as scratch space!
C     PERFORMANCE
         fxx = (TYPE(1.0) - x) * (TYPE(1.0) + x)
C     PERFORMANCE
         fx = TYPE(-1.0) / sqrt (fxx)
C     PERFORMANCE
         fxx = x * fx / fxx
      else
         call EXCEPTION_HANDLER
      endif
#endif /* SECOND */
```

Figure B.3. Template File with Cases for First, Second Derivatives

```
C     PERFORMANCE
      z = abs(x)

C     PERFORMANCE
      fxx = TYPE(0.0)

#ifdef PERFORMANCE
      if (x .gt. TYPE(0.0)) then
         fx =  TYPE(1.0)
      else if (x .lt. TYPE(0.0)) then
         fx = TYPE(-1.0)
      else
C        This is the current NoLimit default
         fx = TYPE(0.0)
c         stop 'ADIFOR Exception: x = 0 in abs(x).'
      endif
#else
      if (x .gt. TYPE(0.0)) then
         fx =  TYPE(1.0)
      else if (x .lt. TYPE(0.0)) then
         fx = TYPE(-1.0)
      else
         call EXCEPTION_HANDLER
      endif
#endif
```

Figure B.4. Template File with Performance-Mode Controlling Directives

```
C     PERFORMANCE
      z = abs(x)

C     PERFORMANCE
      fxx = TYPE(0.0)

C     PERFORMANCE
      if (x .gt. TYPE(0.0)) then
C     PERFORMANCE
         fx =  TYPE(1.0)
C     PERFORMANCE
      else if (x .lt. TYPE(0.0)) then
C     PERFORMANCE
         fx = TYPE(-1.0)
C     PERFORMANCE
      else
#ifdef PERFORMANCE
C        This is the current NoLimit default
         fx = TYPE(0.0)
c         stop 'ADIFOR Exception: x = 0 in abs(x).'
#else
         call EXCEPTION_HANDLER
#endif
```

Figure B.5. Template File with Performance-Mode Controlling Directives, Alternative Style

Appendix C

Sparse Derivative Support for ADIFOR 2.0 through the SparsLinC 1.0 Library

C.1 Introduction

SparsLinC 1.0 (**Sparse Linear Combinations**) is a library of C routines that provide an implementation of the "vector linear combination":

$$w = \sum_{i=1}^{k} \alpha_i * v_i, \tag{C.1}$$

employing sparse data structures. Here w and the v_i are vectors, the α_i are scalar multipliers, and k is referred to as the arity. This operation is the fundamental computational kernel for first-order automatic differentiation.

SparsLinC utilizes dynamic data structures to represent only the nonzero information contained in each vector and performs the vector linear combinations on these sparse representations of the vectors. By doing so, it avoids storing zero values and performing computation with zeros, at the cost of introducing some overhead associated with maintaining sparse data structures.

One way of representing a sparse vector with nnz nonzeros in Fortran is by means of two arrays, each of length nnz, one an integer array containing the indices of the nonzero entries, and the other a floating-point array of appropriate precision, containing the corresponding values. So, for example, the 7-vector

(11.0, 0, 33.0, 44.0, 0, 0, 77.0)

would be represented by

Index Array:	1	3	4	7
Value Array:	11.0	33.0	44.0	77.0

We will refer to this 2-array representation of the vector as the **Fortran Sparse Format**. The corresponding nonsparse representation, which we will call the **Fortran Nonsparse Format**, would be a floating-point array of length 7, containing zeros in entries 2, 5, and 6. Lastly, there is the **SparsLinC Sparse Format**, which is the internal SparsLinC representation of the vector.

In addition to reducing the space required to store derivative values and the time required to compute derivatives, SparsLinC is also useful for uncovering the sparsity features of a problem. For example, the detection of the sparsity pattern of Jacobians is of interest in a number of computations. The computation of the Jacobian using SparsLinC yields the sparsity pattern of the Jacobian as a natural consequence of the work it does in computing the Jacobian, and thus provides all the information needed for a sparse equation solving routine, for example. We anticipate that this feature of SparsLinC will be further strengthened in future releases with the addition of diagnostic capabilities about the "sparsity behavior" of a computation.

From the user's point of view, using SparsLinC is very simple. Much of the task of interfacing ADIFOR 2.0-generated code and SparsLinC is done automatically and is transparent to the user. Section C.5.1 describes how to invoke ADIFOR 2.0 to generate derivative code that uses the SparsLinC library. Such code will be referred to as "sparse derivative code." We will refer to derivative code generated by ADIFOR 2.0 in the default case (i.e., with do-loop implementation of vector linear combinations, rather than calls to SparsLinC routines) as "nonsparse derivative code."

Section C.2 provides some background information necessary to understand the use of SparsLinC with ADIFOR 2.0. Section C.3 defines the notion of sparsity and discusses computational scenarios where sparsity exists and can be exploited by SparsLinC for faster, less memory-intensive code. In the tutorial example given in Chapter 4, Step 4 describes, for the nonsparse (default) case, how to incorporate the ADIFOR 2.0-generated derivative code in the derivative code driver. Section C.4 outlines how this is done in the sparse derivative code driver by calling the appropriate **SparsLinC Access Routines**. These routines are the subset of SparsLinC routines that allow the user to set up and configure SparsLinC, pass data to it, and extract results and performance measures from it. Section C.5 describes how to build a sparse derivative code by using ADIFOR 2.0 and SparsLinC. Section C.6 contains detailed description of the SparsLinC access routines.

C.2 Background

In ADIFOR 2.0, an **active variable** is one that lies on a dependency path from the independent to the dependent variables (the independents and dependents themselves are also considered to be active). Active variables are the ones for which we compute **directional derivatives** with respect to a set of (not necessarily normalized) directions specified via the seed matrix. In the simplest case, each unit direction is defined by one of the independent variables, which is equivalent to setting the seed matrix to be the identity.

We define the term **directional gradient vector** to be the set of directional derivatives of any scalar active variable with respect to all directions specified in the seed matrix. The term **scalar active variable** here refers both to active variables declared as scalars in the user's Fortran source code and to the individual elements of active variables that are declared as arrays. The directional gradient vectors appear as vector operands in the vector linear combinations equation (C.1).

C.3 Where Is SparsLinC Useful?

The main rationale for the development of SparsLinC is to make derivative computation **run** faster and use less memory. But not every problem will result in faster code if SparsLinC is used. The potential gain depends, to a large extent, on the inherent sparsity present in any particular derivative computation.

C.3.1 Definition of Sparsity

In a nonsparse representation, a directional gradient vector V would be declared as an array of length p, where p is the number of directions (i.e., the number of columns in the seed matrix).[1] We denote the number of nonzeros in V at a given point t during the execution by $V_{t.nnz}$. The percentage of zero entries or **sparsity** of V_t is defined as

$$V_{t.sparsity} := (1 - \frac{V_{t.nnz}}{p}) * 100\%. \tag{C.2}$$

[1] For the sake of clarification, we note that p denotes the same quantity as the Fortran variable g_p_, used elsewhere in this document.

A good measure for the **overall sparsity** present in a derivative computation is the median of the sparsities of all directional gradient vectors *during the entire execution of the derivative code.*

A necessary (but not sufficient) condition for SparsLinC to improve the runtime performance of derivative computation is that the number of directions with respect to which we wish to compute derivatives be "large". This is perhaps an obvious, but nonetheless significant, point, since if the number of directions is small, directional gradient vectors will be short and any strategy to exploit sparsity will be defeated by the overhead associated with implementing that strategy. The determination of what is considered a large sparse problem is to a great extent dependent upon the nature of the problem; however, in our experience, the threshold at which our strategy becomes effective is 20–30 directions.

Another important issue concerning sparsity in derivative computations is that the sparsity of the final result (the nonzero structure of the final directional gradient vectors of the dependents) is only a lower bound on the sparsity of the intermediate directional gradient vectors; that is, the overall sparsity of the problem may be (and often is very) much higher than that of the final derivative result. In general, sparsity diminishes as the computation proceeds, because for all vector linear combinations, the nonzero index set of the resulting left-hand-side vector is the union of index sets of the right-hand-side vectors.[2] As a consequence, in many problems, there may be a lot of "hidden" sparsity that can be exploited by using SparsLinC.

C.3.2 Sparse Derivative Problem Types

The numerical computation of gradients and Jacobians is an important step in the solution of many nonlinear problems, such as constrained optimization, mesh computations, and the solution of systems of stiff differential and algebraic equations. In many instances, these problems require derivative computations that have inherent sparsity. Two examples are gradients of partially separable functions and sparse Jacobians.

A function is partially separable if it can be represented as

$$f(x) = \sum_{i=1}^{m} f_i(x), \tag{C.3}$$

where m is the number of partitions, and where each component function, $f_i(x)$, is typically a function of just a few of the elements of x, implying that each of the corresponding directional gradient vectors, $\nabla f_i(x)$, will be sparse, even though the aggregate f depends on all of x, leading to a dense final gradient $\nabla f(x)$. Any f with a sparse Hessian belongs to this class of problem [14], regardless of whether the partially separable structure is expressed explicitly in the code.

For many Jacobian computations, the final Jacobian is itself sparse, implying that there is much sparsity to be exploited in the intermediate computations. As discussed above, every intermediate directional gradient vector is at least as sparse as (and often much sparser than) the final Jacobian.

C.4 Usage of SparsLinC Access Routines

This section outlines the SparsLinC access routines and their use in the derivative code driver. These routines allow the user to set up and configure SparsLinC, pass data to it, and extract results and performance measures from it.

C.4.1 About SparsLinC 1.0 Routines and Their Names

SparsLinC provides multiprecision arithmetic support, meaning that the underlying vectors can be represented in `REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `DOUBLE COMPLEX` precision. The routines involving a vector or vectors have a prefix letter designating the "precision" of the operation. For each precision-dependent

[2]This discussion precludes the possibility of the occurrence of numerical zeros resulting from exact cancellation (e.g., $a + (-a)$) and zero multipliers. In our experience, exact cancellation rarely occurs in derivative computation, and currently, SparsLinC does not check for it (i.e., numerically zero vector entries are treated like nonzero entries). SparsLinC does, however, check for zero multipliers, and vectors with zero multipliers are not referenced.

SparsLinC routine, all instantiations of the routine have the same interface, meaning that they have the same arguments, in the same order, and with identical declarations except for the types of the vectors and multipliers (as an example, see the declaration of `VALVEC` in the definition of the `[S,D,C,Z]SPSD` routines in Section C.6).

Here is a summary of the naming conventions we have adopted for SparsLinC routines:

- The first letter will be an "`S`", "`D`", "`C`", "`Z`", or "`X`" indicating, respectively, whether the routine manipulates vectors in `REAL`, `DOUBLE PRECISION`, `COMPLEX`, or `DOUBLE COMPLEX`precision or whether it is a nonnumeric utility routine.
- The second and third letters will be "`SP`", to denote that the routine is in the **SP**arse library.
- The last two or three letters will be an abbreviation of the task performed by the routine.

We use the shorthand, "`[S,D,C,Z]`*name*" to refer to all four precision instantiations of a routine *name*.

C.4.2 Declaration of Sparse Variables

In Section C.2 we introduced the concept of directional gradient vectors. In the case of the nonsparse invocation of ADIFOR 2.0, these vectors are implemented as Fortran arrays. In the following examples in this and subsequent sections (C.4.2 - C.5.2), assume that `x` is the independent variable (i.e., all 1000 entries of `x` are independent variables), `f` is the dependent variable, and `w` is an active variable we need to access in the derivative code driver:

```
      REAL x(1000), f(5), w
```

In the nonsparse case, the derivative code generated by ADIFOR 2.0 (assuming the ADIFOR 2.0 options AD_PREFIX and AD_SEP have the default bindings of "g" and "_", respectively) will contain the following declarations:

```
      REAL g_x(g_pmax_,1000), g_f(g_pmax_,5), g_w(g_pmax_)
```

By contrast, in the sparse case, the derivative code generated by ADIFOR 2.0 will contain the following declarations:

```
      INTEGER g_x(1000), g_f(5), g_w
```

Note that the Fortran interface to SparsLinC declares each directional gradient vector to be an `INTEGER`. This is because each Fortran `INTEGER` gradient variable will be interpreted by SparsLinC to be a pointer to the sparse representation of the corresponding vector.

It is usually possible to clip-and-paste the declarations for the directional gradient vectors, and possibly the declarations of `COMMON` blocks that contain directional gradient vectors, from the code generated by ADIFOR 2.0. This is true for both nonsparse and sparse applications of ADIFOR 2.0. Just be aware that the declarations for the directional gradient vectors in the nonsparse and sparse codes are different.

Parenthetically, if you want to compare the sparse and nonsparse approaches for a particular problem, it is often good coding practice to write one driver for both, with preprocessor directives specifying the parts where the two differ. For example, for the above declaration, the following code could appear in the driver:

```
#ifdef NON_SPARSE
      REAL g_x(g_pmax_,1000), g_f(g_pmax_,5), g_w(g_pmax_)
#elif SPARSE
      INTEGER g_x(1000), g_f(5), g_w
#endif
```

We use this format, wherever applicable (i.e., wherever corresponding sparse and nonsparse codes are present), in the rest of this discussion. (On most Unix systems, filenames ending with ".F" are interpreted by makefiles as Fortran files with preprocessor statements. Users unfamiliar with preprocessor directives can consult the "man" pages for "cpp", the C preprocessor.)

C.4.3 Initializing and Customizing SparsLinC

SparsLinC data structures must be initialized before any computation can be performed. To this end, the user *must* call the routine XSPINI before all other calls to any SparsLinC (except for calls to XSPCNF, which must precede the call to XSPINI, as described below) or ADIFOR 2.0-generated routines. XSPINI takes no arguments and is called as follows:

```
CALL XSPINI
```

The routine XSPCNF provides a means of tuning SparsLinC data structures for a particular problem at hand. Most sparse vectors maintained by SparsLinC are stored in what is commonly referred to as the "single subscript" and "compressed subscript" scheme. The single subscript scheme is the one already introduced in the Fortran context in Section C.1. In the compressed subscript scheme, in contrast, we keep track of nonzero index ranges, representing the vector of Section C.1 by

Index Array:	[1,1]	[3,4]	[7,7]	
Value Array:	11.0	33.0	44.0	77.0

This representation is more efficient than the single-subscript representation when sparse vectors contain a good portion of contiguous nonzero index ranges. A contiguous nonzero index range is a range of indices wherein all the corresponding values are nonzeros. For example, for our vector above, the largest such range has size 2 and contains elements 3 and 4. This scenario commonly arises when computing Jacobians with banded structure or gradients of partially separable functions. SparsLinC automatically converts a vector from the single-subscript to the compressed-subscript representation when the number of nonzeros in the vector exceeds a certain threshold switch_threshold, say.

Either way, since the size to which vectors can grow is not known a priori, SparsLinC must provide, for the value and index arrays, a data structure capable of representing vectors of arbitrary size. The data structure currently employed in SparsLinC is a linked list of arrays each of which has a fixed number of entries. Let us denote this number of entries with SSbucket_size for the single subscript scheme and CSbucket_size for the compressed subscript scheme.

SparsLinC allows the user to adjust these values using the XSPCNF routine. For example, the sequence of calls

```
CALL XSPCNF(1,10)
CALL XSPCNF(2,500)
CALL XSPCNF(3,20)
```

sets SSbucket_size to 10, CSbucket_size to 500, and switch_threshold to 20. This would be appropriate, for example, for computing the gradient of a partially separable function (see Section C.3.2), where each ∇f_i usually contains about 20 nonzeros, and the number of independent variables is greater than 500.

While XSPINI assigns default values to these parameters and hence there is, from a functional perspective, no need to call XSPCNF, we encourage experimenting with these parameters and welcome feedback. Our experiments have shown that SparsLinC performs best if CSbucket_size is close in value to the size of the largest contiguous nonzero index range present in the problem. The tradeoff is between runtime and memory, where a larger value of CSbucket_size is likely to result in faster runtime, but also the dynamic allocation of more memory. In all cases, SSbucket_size should be set smaller (and usually much smaller) than CSbucket_size and should not exceed switch_threshold. We are working on a facility to trace and assimilate SparsLinC runtime information to aid with SparsLinC performance tuning.

The user should pay heed to the following important note: **XSPCNF may be called only before calling XSPINI to set SSbucket_size and CSbucket_size.** This is because once XSPINI is called, the array dimensions set via these options cannot be modified. Calling XSPCNF to set SSbucket_size and CSbucket_size, after a call to XSPINI, will result in a runtime error. Calls to XSPCNF to set switch_threshold can be made at any time.

C.4.4 Initializing the Seed Matrix

Each of the precision-specific SparsLinC routines [S,D,C,Z]SPSD converts a precision-specific sparse vector stored in the Fortran Sparse Format into a corresponding vector in the SparsLinC Sparse Format. In the following example, for the purpose of demonstration, we initialize columns 19 and 20 of g_x (corresponding to the derivatives of x(19) and x(20)), in both the nonsparse and sparse ways (assume that the arrays, INDVEC and VALVEC are declared appropriately):

```
#ifdef NON_SPARSE
      g_x( 7,19) = 2.0
      g_x(19,19) = 1.0
      g_x(20,20) = 1.0
#elif SPARSE
      INDVEC(1) = 7
      VALVEC(1) = 2.0
      INDVEC(2) = 19
      VALVEC(2) = 1.0
      CALL SSPSD(g_x(19),INDVEC,VALVEC,2)
      CALL SSPSD(g_x(20),20,1.0,1)
#endif
```

Note also that a vector must be initialized in a "one-shot" fashion; hence, for example, the following piece meal approach would be an *incorrect* initialization of g_x(19):

```
      INDVEC(1) = 7
      VALVEC(1) = 2.0
      CALL SSPSD(g_x(19),INDVEC,VALVEC,1)
      INDVEC(1) = 19
      VALVEC(1) = 1.0
      CALL SSPSD(g_x(19),INDVEC,VALVEC,1)
```

Because of the "destructive copy" feature of SPSD (see Section C.6), the above would be equivalent to having made only the second of the two calls.

C.4.5 Extracting Directional Gradient Vectors from SparsLinC

SparsLinC provides two sets of precision-specific interfaces for extracting vector results:

```
[S,D,C,Z]SPXDQ (XVEC, INLEN, VPTR, OUTLEN, INFO)
```

extracts sparse_object(VPTR) into the Fortran Nonsparse Format vector XVEC. INLEN is the size of XVEC. The returned value OUTLEN is the largest index in the nonzero index set in sparse_object(VPTR). The value of INFO is used to indicate whether XVEC was sufficiently large to store all of the nonzero elements in sparse_object(VPTR). If OUTLEN is less than INLEN, then XVEC(OUTLEN+1:INLEN) is set to zero.

```
[S,D,C,Z]SPXSQ (INDVEC, VALVEC, INLEN, VPTR, OUTLEN, INFO)
```

extracts sparse_object(VPTR) into the Fortran Sparse Format vector represented by the two arrays INDVEC and VALVEC. INLEN is the size of the arrays INDVEC and VALVEC. The returned value OUTLEN is the number of nonzeros in sparse_object(VPTR). The value of INFO is used to indicate whether XVEC was sufficiently large to store all of the nonzero elements in sparse_object(VPTR). If OUTLEN is less than INLEN, then VALVEC(OUTLEN+1:INLEN) and INDVEC(OUTLEN+1:INLEN) are not referenced.

In the following code segments, we show examples of the usage of these extraction routines along with the corresponding necessary declarations (there is no equivalent ADIFOR 2.0 nonsparse extraction, since in that case the output variables are already in Fortran Nonsparse Format).

SPXDQ Example

```
PARAMETER (in_len_xd = g_pmax_)
INTEGER out_len_xd(5), info_xd(5)
REAL g_f_xd(in_len_xd,5)
...
DO i = 1, 5
  CALL SSPXDQ(g_f_xd(1,i), in_len_xd, g_f(i),
              out_len_xd(i), info_xd(i))
ENDDO
```

in_len_xd is a user-defined value specifying the leading dimension of the Fortran nonsparse column vectors of g_f_xd, i.e., it is the user's estimate of what is the largest index of nonzero value in the vector to be extracted. In this case, by setting in_len_xd = g_pmax_, we have ensured ourselves that the SparsLinC Sparse Format vector will always "fit" into the Fortran Nonsparse Format vector. (In the next example we will discuss the case of underestimating memory requirements.)

Note that as specified above, g_f_xd is defined identically to the nonsparse g_f in Section C.4.2. Given Fortran's column order array storage, the above call to SSPXDQ causes g_f_xd to be aligned exactly with the nonsparse g_f.

SPXSQ Example

```
PARAMETER (in_len_xs = 40)
INTEGER g_f_ind_xs(in_len_xs,5), out_len_xs(5), info_xs(5)
REAL g_f_val_xs(in_len_xs,5)
...
DO i = 1, 5
  CALL SSPXSQ(g_f_ind_xs(1,i), g_f_val_xs(1,i), in_len_xs, g_f(i),
              out_len_xs(i), info_xs(i))
ENDDO
```

Here, our choice of in_len_xs = 40 implies that we have made the assumption that there are at most 40 nonzeros in any row of the Jacobian $\frac{\partial f}{\partial x}$ (i.e., given our declaration of x in Section C.4.2, we assume that the least sparse directional derivative vector is 96% sparse). To make sure that our memory requirement assumption holds, we add the following code:

```
max_len_xs = 0
DO i = 1, 5
  IF (info_xs(i) .NE. 0 .AND. out_len_xs(i) .GT. max_len_xs) THEN
    max_len_xs = out_len_xs(i)
  END IF
ENDDO
```

Now max_len_xs is encoded with the information we need. That is, if zero, our assumption was true, else, max_len_xs is equal to the true number of nonzeros in the least sparse row of the Jacobian and we know how much memory is really needed to extract all nonzero derivative values.

C.4.6 Adding the Contents of a Sparse Vector to a Dense Vector

Two SparsLinC routines are provided for adding a SparsLinC Sparse Format vector to a Fortran Nonsparse Format vector.

```
[S,D,C,Z]SPXMQ (XVEC, INLEN, MULT, VPTR, OUTLEN, INFO)
```

adds to XVEC the contents of sparse_object(VPTR) multiplied by MULT (i.e., XVEC = XVEC + MULT * sparse_object(VPTR)).

```
[S,D,C,Z]SPXAQ (XVEC, INLEN, VPTR, OUTLEN, INFO)
```

is identical to SPXMQ, except that the multiplier is assumed to be one (i.e., XVEC = XVEC + sparse_object(VPTR)). Note that SPXMQ and SPXAQ are functionally very similar to the SPXDQ routine, the only difference being that SPXDQ "assigns to" XVEC while SPXMQ and SPXAQ "add to" XVEC the contents of the sparse vector. Note also, that the interfaces of SPXAQ and SPXDQ are identical.

C.4.7 Dumping the Contents of a Sparse Vector

SparsLinC provides a set of precision-specific interfaces for dumping a sparse vector to a file.

```
[S,D,C,Z]SPPRQ (VPTR, EXT)
```

writes the number of nonzeros as well as index/value pairs of sparse_object(VPTR) to stdout or a file. EXT is an INTEGER in the range [0,999] and specifies the destination of the output: if zero, output is written to stdout; otherwise, output is written to the file SPPRQ.EXT.

SPPRQ can be a useful routine during debugging, to quickly check the values of a derivative vector somewhere in the code. It also has the advantage of not requiring that the user provide memory in which to extract the nonzero values in the sparse vector.

Admittedly, the interface of SPPRQ is rather crude. This is because we have avoided passing string arguments, because of the inconsistency of the Fortran-to-C string-passing protocols on different platforms.

SPPRQ Example

```
DO i = 1, 2
  CALL SPPRQ(g_f(i), 6)
ENDDO
```

The above code prints the nonzero derivative information in g_f(1) and g_f(2) into the file "SPPRQ.6" in the current directory. Assume that g_f(1) g_f(2) contain 4 and 2 nonzero values, respectively. Then the following is an example of what might be the contents of "SPPRQ.6' subsequent to the execution of the above code:

```
Number of nonzeros = 4
    Index    Value
    -----    -----
        4    -4.892400e-01
        5     6.523200e+00
        6    -1.630800e+00
      188    -2.030000e+01

Number of nonzeros = 2
    Index    Value
    -----    -----
       37     3.812800e+00
      256     1.000000e+00
```

Note that the vectors are printed out in the order in which the corresponding SPPRQ was called, and there is no identification in the file denoting which set of numbers belong to which vector. This task is left to the user.

C.4.8 Extracting Performance Information

In addition to providing derivative information, SparsLinC can also provide information about its own performance. Because of the system-specific nature of timing routines, runtime measures are best arrived at by enveloping the appropriate system calls around the call to the top level subroutine. For example:

```
        CALL timer(t1)
        CALL g_top_foo(x, g_x, ...)
        CALL timer(t2)
        t_elapsed = t2 - t1
```

An implementation of `timer` that returns elapsed user time as a real value is provided in the `$(AD_HOME)/lib/libtimer-$(AD_ARCH).a` archive, which can be linked into your executable by adding one of the following to your link line:

```
    ... -L$AD_HOME/lib -ltimer-$AD_ARCH
```

or

```
    ... $AD_HOME/lib/libtimer-$AD_ARCH.a
```

The SparsLinC routine `XSPMEM` returns how many kilobytes of memory have been dynamically allocated in the process of computing derivatives:

```
        REAL USEDKB
        ...
        CALL XSPMEM(USEDKB)
```

C.4.9 Freeing Dynamically Allocated Memory

The routine `XSPFRA` frees all dynamically allocated memory in SparsLinC. Freeing memory might be useful if after finishing the derivative computation, the user wishes to perform some further memory-intensive computation. There are no arguments, and the call is simply

```
        CALL XSPFRA
```

`XSPFRA` has the effect of leaving "dangling pointers", meaning that the Fortran `INTEGER` gradient variables, which are interpreted by SparsLinC as pointers, will retain the values (addresses) they contained before `XSPFRA` was called. However, after the call to `XSPFRA`, the memory pointed to by these pointers will no longer be under SparsLinC control. Any attempt to use these variables as pointers (e.g., by using them as pointer arguments to some SparsLinC routine) will likely cause a segmentation fault. For this reason, *no calls to any SparsLinC routine should be made after* `XSPFRA`.

C.5 A Brief Tutorial Example

SparsLinC is designed to be easy to use. There are three basic steps to be followed:

1. Applying ADIFOR 2.0 to generate sparse derivative code. (Section C.5.1),
2. Writing the "Sparse" derivative code driver. (Section C.5.2), and
3. Linking all the Fortran code with SparsLinC 1.0. (Section C.5.3).

C.5.1 Invoking ADIFOR 2.0 to Generate Sparse Code

In order for ADIFOR 2.0 to create derivative code in which vector linear combinations are performed by means of calls to SparsLinC 1.0 routines, the following ADIFOR 2.0 option must be specified in either the ADIFOR 2.0 command line or the script file:

AD_FLAVOR=sparse

C.5.2 An Example Derivative Code Driver

The derivative code driver is a user-generated Fortran program that invokes the derivative code generated by ADIFOR 2.0 (see Step 4 in the tutorial in Chapter 4). In general, the sparse derivative code driver is analogous to the nonsparse derivative code driver and differs from the latter in only a few places. The following is an example derivative code driver, based on the code fragments shown throughout Section C.4:

```
      PROGRAM DRIVER

      REAL x(1000), f(5), w

#ifdef NON_SPARSE
      REAL g_x(g_pmax_,1000), g_f(g_pmax_,5), g_w(g_pmax_)
#elif SPARSE
      INTEGER g_x(1000), g_f(5), g_w
      PARAMETER (in_len_xs = 40)
      INTEGER g_f_ind_xs(in_len_xs,5), out_len_xs(5), info_xs(5)
      REAL g_f_val_xs(in_len_xs,5)
      REAL USEDKB
#endif

CCC   We assume some statements at this point initialize the independent
CCC   variables.

#ifdef SPARSE
CCC   Tuning of SparsLinC parameters (optional) and mandatory initialization
      CALL XSPCNF ( 1, 20 )
      CALL XSPCNF ( 2, 500)
      CALL XSPCNF ( 3, 10 )
      CALL XSPINI
#endif

CCC   Initializing the seed matrix as identity.

#ifdef NON_SPARSE
      DO i=1,1000
         DO j=1,1000
            g_x(i,j) = 0.0d0
         ENDDO
         g_x(i,i) = 1.0d0
      ENDDO
#elif SPARSE
      DO i=1,1000
         CALL SSPSD(g_x(i),i,1.d0,1)
      ENDDO
#endif

#ifdef NON_SPARSE
      CALL g_top_foo(g_p_, x, g_x, ldg_x, f, g_f, ldg_f,
     +               w, g_w, ldg_w, non_active_var)
#elif SPARSE
      CALL g_top_foo(x, g_x, f, g_f, w, g_w, non_active_var)
#endif

#ifdef SPARSE
      DO i = 1, 5
```

```
        CALL SSPXSQ(g_f_ind_xs(1,i), g_f_val_xs(1,i), in_len_xs, g_f(i),
                    out_len_xs(i), info_xs(i))
      ENDDO
      max_len_xs = 0
      DO i = 1, 5
        IF (info_xs(i) .NE. 0 .AND. out_len_xs(i) .GT. max_len_xs) THEN
          max_len_xs = out_len_xs(i)
        ENDIF
      ENDDO

      CALL XSPMEM(USEDKB)
#endif
```

Taking a close look at the calls to the top level routine, **g_top_foo**, in the nonsparse and sparse derivative code driver, we realize that the only differences between the sparse and nonsparse calls are that there is never a need to pass a leading dimension argument along with each gradient variable argument, and also there is no need to pass a value for **g_p_**, the runtime nonsparse directional gradient vector size. Note that, regardless of whether ADIFOR 2.0 is invoked in the sparse or nonsparse mode, it generates the same subroutine name.

C.5.3 Linking with SparsLinC 1.0

To use SparsLinC, you must link a machine-specific version of the SparsLinC library into your executable by adding one of the following to your link line:

```
... -L$AD_HOME/lib -lSparsLinC-$AD_ARCH
```

or

```
... $AD_HOME/lib/libSparsLinC-$AD_ARCH.a
```

C.6 Detailed Specification of Access Routines

This section contains the detailed description of the SparsLinC 1.0 access routines discussed in Section C.4.

We adopt the convention that for a Fortran INTEGER variable VPTR, acting as a pointer to a SparsLinC Sparse Format vector, the sparse derivative object pointed to by VPTR is called sparse_object(VPTR). Also, to save space, only the calling sequence for one particular floating-point precision is provided.

SSPSD, DSPSD, CSPSD, ZSPSD

```
SUBROUTINE SSPSD (VPTR, INDVEC, VALVEC, LEN)
```

Purpose

Conversion of a vector in Fortran Sparse Format into a vector in SparsLinC Sparse Format. The Fortran Sparse Format vector is given by the two arrays, `INDVEC(1:LEN)` and `VALVEC(1:LEN)`, representing the indices and values of a sparse vector x (say), respectively. x is copied into sparse_object(`VPTR`), which is the vector in SparsLinC Sparse Format. The indices in `INDVEC` need not be in any particular order (internally, `SPSD` performs an ascending order sort). However, `INDVEC` and `VALVEC` must be identically aligned. That is, if in the Fortran Nonsparse Format x has a nonzero entry at index i with value v, then for some J, `INDVEC(J)` $= i$ and `VALVEC(J)` $= v$. `SPSD` performs a *destructive copy.* That is, if sparse_object(`VPTR`) had been previously allocated (via `SPSD` or as a result of being an output argument of some other SparsLinC routine), the previous information in sparse_object(`VPTR`) is lost, and the dynamically allocated memory where that information resided is deallocated.

Arguments

`VPTR`	(output) `INTEGER` Upon exit, sparse_object(`VPTR`) contains a copy of the sparse vector represented by `INDVEC` and `VALVEC`.
`INDVEC`	(input) `INTEGER` array, dimension (`LEN`) Indices of the nonzero values of the sparse vector. (We assume that indices are ≥ 1; therefore, `INDVEC` entries ≤ 0 would be incorrect and would result in a runtime error.)
`VALVEC`	(input) `REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX]` array, dimension (`LEN`) Nonzero values of the sparse vector.
`LEN`	(input) `INTEGER` `LEN` ≥ 0 is the number of nonzeros in the sparse vector. If `LEN` $= 0$, `VPTR` is initialized to point to the vector of all zeros and `INDVEC` and `VALVEC` are not referenced.

SSPXDQ, DSPXDQ, CSPXDQ, ZSPXDQ

```
SUBROUTINE SSPXDQ (XVEC, INLEN, VPTR, OUTLEN, INFO)
```

Purpose

Extracts sparse_object(`VPTR`) into the Fortran Nonsparse Format vector `XVEC`.

Arguments

`XVEC`	(output) `REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX]` array, dimension (`INLEN`) On exit, if `INFO` $= 0$, `XVEC(1:INLEN)` will contain a dense representation of sparse_object(`VPTR`). If `OUTLEN` $<$ `INLEN`, then `XVEC(OUTLEN+1:INLEN)` is initialized to all zeros. If `INFO` $\neq 0$, `XVEC` is not referenced.
`INLEN`	(input) `INTEGER` Length of `XVEC`.
`VPTR`	(input/output) `INTEGER` Pointer to the SparsLinC Sparse Format vector. If `VPTR` equals `NULL`, it is initialized to point to the vector of all zeros (which is why it might be an output argument).

OUTLEN	(output) INTEGER Largest index in the nonzero index set in sparse_object(VPTR). This value will always be returned, whether XVEC is initialized or not. See the description of INFO below.
INFO	(output) INTEGER If INLEN < OUTLEN, INFO will be set to -1, and XVEC is not referenced. Otherwise, INFO is set to 0, and XVEC(1:INLEN) is initialized to a Fortran Nonsparse Format copy of sparse_object(VPTR).

SSPXSQ, DSPXSQ, CSPXSQ, ZSPXSQ

```
SUBROUTINE SSPXSQ (INDVEC, VALVEC, INLEN, VPTR, OUTLEN, INFO)
```

Purpose

Extracts sparse_object(VPTR) into the Fortran Sparse Format vector represented by the two arrays, INDVEC and VALVEC.

Arguments

INDVEC	(output) INTEGER array, dimension (INLEN) On exit, if INFO = 0, INDVEC(1:OUTLEN) contains the indices of the nonzero entries of sparse_object(VPTR). If INFO = 0 and OUTLEN < INLEN then INDVEC(OUTLEN+1:INLEN) is not referenced. If INFO $\neq$ 0, INDVEC is not referenced.
VALVEC	(output) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (INLEN) On exit, if INFO = 0, VALVEC(1:OUTLEN) will contain the nonzero entries of sparse_object(VPTR). If INFO = 0 and OUTLEN < INLEN then VALVEC(OUTLEN+1:INLEN) is not referenced. If INFO $\neq$ 0, VALVEC is not referenced.
INLEN	(input) INTEGER Length of INDVEC and VALVEC.
VPTR	(input/output) INTEGER Pointer to the SparsLinC Sparse Format vector. If VPTR equals NULL, it is initialized to point to the vector of all zeros (which is why it might be an output argument).
OUTLEN	(output) INTEGER Number of nonzeroes in sparse_object(VPTR). This value will always be returned, whether INDVEC and VALVEC are initialized or not. See the description of INFO below.
INFO	(output) INTEGER If INLEN < OUTLEN, INFO will be set to -1, and INDVEC and VALVEC are not referenced. Otherwise, INFO is set to 0, and INDVEC(1:OUTLEN) and VALVEC(1:OUTLEN) are initialized to the Fortran Sparse Format copy of sparse_object(VPTR).

SSPXMQ, DSPXMQ, CSPXMQ, ZSPXMQ

```
SUBROUTINE SSPXMQ (XVEC, INLEN, MULT, VPTR, OUTLEN, INFO)
```

Purpose

Adds the weighted contents of sparse_object(VPTR) to the Fortran Nonsparse Format vector XVEC, where MULT is the multplicative weight (i.e., XVEC = XVEC + MULT * sparse_object(VPTR)). For example, say XVEC is a vector of length 7 containing all ones, MULT = 2.0, and sparse_object(VPTR) is as follows:

Index Array:	1	3	4	7
Value Array:	11.0	33.0	44.0	77.0

Subsequent to the call to this routine, XVEC would contain the following:

(23.0, 1.0, 67.0, 89.0, 1.0, 1.0, 155.0)

Arguments

XVEC	(input/output) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] array, dimension (INLEN) On exit, if INFO = 0, XVEC(1:INLEN) will have added to it the weighted contributions of the values in sparse_object(VPTR), with MULT specifying the weight. If INFO $\neq$ 0, XVEC is not modified.
INLEN	(input) INTEGER Length of XVEC.
MULT	(input) REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX] Multiplier.
VPTR	(input/output) INTEGER Pointer to the SparsLinC Sparse Format vector. If VPTR equals NULL, it is initialized to point to the vector of all zeros (which is why it might be an output argument).
OUTLEN	(output) INTEGER Largest index in the nonzero index set in sparse_object(VPTR). This value will always be returned, whether XVEC is modified or not. See the description of INFO below.
INFO	(output) INTEGER If INLEN < OUTLEN, INFO will be set to -1, and XVEC is not modified. Otherwise, INFO is set to 0, and XVEC(1:INLEN) is modified as described above.

SSPXAQ, DSPXAQ, CSPXAQ, ZSPXAQ

```
SUBROUTINE SSPXAQ (XVEC, INLEN, VPTR, OUTLEN, INFO)
```

Purpose

Adds the contents of sparse_object(**VPTR**) to the Fortran Nonsparse Format vector **XVEC** (i.e., **XVEC** = **XVEC** + sparse_object(**VPTR**)). (**SPXA** is identical to the **SPXMQ** routine with **MULT** = 1.0; see the documentation for **SPXMQ**.)

Arguments

`XVEC`	(input/output) `REAL [DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX]` array, dimension `(INLEN)` On exit, if `INFO` = 0, `XVEC(1:INLEN)` will have added to it the values in sparse_object(`VPTR`). If `INFO` $\neq$ 0, `XVEC` is not modified.
`INLEN`	(input) `INTEGER` Length of `XVEC`.
`VPTR`	(input/output) `INTEGER` Pointer to the SparsLinC Sparse Format vector. If `VPTR` equals `NULL`, it is initialized to point to the vector of all zeros (which is why it might be an output argument).
`OUTLEN`	(output) `INTEGER` Largest index in the nonzero index set in sparse_object(`VPTR`). This value will always be returned, whether `XVEC` is modified or not. See the description of `INFO` below.
`INFO`	(output) `INTEGER` If `INLEN` < `OUTLEN`, `INFO` will be set to -1, and `XVEC` is not modified. Otherwise, `INFO` is set to 0, and `XVEC(1:INLEN)` is modified as described above.

SSPPRQ, DSPPRQ, CSPPRQ, ZSPPRQ

```
SUBROUTINE SSPPRQ (VPTR, EXT)
```

Purpose

Writes number of nonzeros as well as index/value pairs of sparse_object(**VPTR**) onto stdout or a file, with the following format:

```
Number of nonzeros = . . .
        Index    Value
        -----    -----
        . . .    . . .
        . . .    . . .
```

Arguments

`VPTR`	(input/output) `INTEGER` Pointer to the SparsLinC Sparse Format vector. If `VPTR` equals `NULL`, it is initialized to point to the vector of all zeros (which is why it might be an output argument).
`EXT`	(input) `INTEGER` Must be in the range [0,999]. If `EXT` = 0, output written is to stdout. Otherwise `EXT` is converted to its ASCII equivalent and used as the extension appended to the filename "`SPPR.`" and output is written to this file.

XSPCNF

```
SUBROUTINE XSPCNF (OPT, VAL)
```

Purpose

Allows user to customize SparsLinC for each run. The following table specifies for each parameter its name, option number, default value, and range of allowable values. "SS-bucket_size" and "CSbucket_size" are the number of entries per array in the linked list representation of a single-subscript and compressed-subscript vector respectively. For all vector linear combinations, if at the conclusion of the computation the left-hand-side vector has an SS representation and the number of its nonzero entries exceeds "switch_threshold", the vector is converted to a CS representation.

Name	OPT	Default	Range
SSbucket_size	1	8	>1
CSbucket_size	2	32	>1
switch_threshold	3	16	>1

`XSPCNF` with `OPT` = 1 or `OPT` = 2 may be called only before calling `XSPINI`. Calling `XSPCNF` with `OPT` = 1 or 2 after a call to `XSPINI` will result in a runtime error. Calls to `XSPCNF` with `OPT` = 3 can be made at any time.

Arguments

`OPT`	(input) `INTEGER` Specifies the option number associated with a given parameter as given in the above table.
`VAL`	(input) `INTEGER` The new value for the parameter specified by `OPT`.

XSPMEM

```
SUBROUTINE XSPMEM (USEDKB)
```

Purpose

Reports how many kilobytes of memory have been allocated dynamically in SparsLinC.

Arguments

`USED`	(output) `REAL`. The number of kilobytes of storage allocated for SparsLinC data structures.

XSPINI

```
SUBROUTINE XSPINI
```

Purpose

Initializes the sparse data structures by dynamically allocating memory for some SparsLinC-internal global variables. It must be called before any of the other SparsLinC routines (except for calls to `XSPCNF` with `OPT`s 1-15) and needs to be called no more than once (when called more than once, all but the first call act as no-ops).

Arguments

none

XSPFRA

```
SUBROUTINE XSPFRA
```

Purpose

Frees all memory allocated for C sparse vector data structures. **Note: all pointers to sparse directional gradient variables (`VPTR`'s) are left dangling.**

Arguments

none

Appendix D

Changes in Naming Conventions between ADIFOR 1.0 and ADIFOR 2.0

ADIFOR 1.0 and ADIFOR 2.0 create external names differently. Specifically, by default, ADIFOR 2.0 now uses a separator character of "_" in all generated names. Also, since ADIFOR 2.0 no longer attempts to clone procedures based on calling context, the derivative computing procedure for a procedure `foo` will be named `g_foo`, assuming that `AD_PREFIX` and `AD_SEP` have not been modified from their default values.

Make sure that you make any names of derivative computing procedures and gradient common blocks that appear in your derivative driver code correspond to the new naming conventions. You will be reminded by the linker if the names of procedures in your derivative driver are inconsistent. YOU WILL NOT SEE ANY WARNING, HOWEVER, IF YOU FAIL TO MAKE THE NAMES OF GRADIENT COMMON BLOCKS BE CONSISTENT BETWEEN YOUR DRIVER AND THE GENERATED DERIVATIVE CODE.

Acknowledgments

We thank Andreas Griewank of the University of Dresden and George Corliss of Marquette University for their invaluable contributions in getting the ADIFOR project started. We are grateful to the users of ADIFOR 1.0 for putting up with the shortcomings of this system and for providing us with valuable feedback. In particular we acknowledge Larry Green and his colleagues at the Multidisciplinary Optimization Branch at NASA Langley, Joe Manke of Boeing Computing Services, Gordon Pusch at Argonne National Laboratory, and Janet Rogers of the National Institute for Science and Technology. We are also indebted to John Dennis of Rice University, Jorge Moré of Argonne National Laboratory, Ken Kennedy of Rice University, Gerald Marsh of Argonne National Laboratory and Mani Salas and Tom Zang, both of NASA Langley, for their support of our work. We also acknowledge the contributions of Heike Baars, Brad Homann, Ernesto Diaz, Fred Dilley, Moe El-Khadiri, Tim Knauff, Aaron Ross, and Vitaly Shmatikov during their student internships at Argonne National Laboratory. Lastly, we thank Judy Beumer, Mike Fagan, and Gail Pieper for their careful reading of the manuscript and their suggestions for improving the presentation of this document.

Bibliography

[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 Release 2.0.* SIAM, Philadelphia, 1994.

[2] B. M. Averick, R. G. Carter, J. J. Moré, and G. L. Xue. The MINPACK-2 test problem collection. Technical Report ANL/MCS-TM-150 (Revised), Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

[3] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[4] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs, 1994. Preprint MCS-P481-1194, Mathematics and Computer Science Division, Argonne National Laboratory, and CRPC-TR94491, Center for Research on Parallel Computation, Rice University.

[5] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Report ANL/MCS-TM-159, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[6] Christian Bischof, Larry Green, Kitty Haigler, and Tim Knauff. Parallel calculation of sensitivity derivatives for aircraft design using automatic differentiation. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, AIAA 94-4261, pages 73–84. American Institute of Aeronautics and Astronautics, 1994.

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

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

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

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

[11] 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, 1991.

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

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

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

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

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

[17] J. J. Moré. On the performance of algorithms for large-scale bound constrained problems. In T. F. Coleman and Y. Li, editors, *Large-Scale Numerical Optimization*, pages 31–45. SIAM, Philadelphia, 1991.

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

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