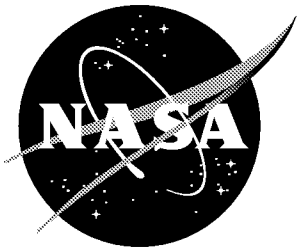
NASA/TM–2002–211738



Some User's Insights Into ADIFOR 2.0D

Daniel P. Giesy
Langley Research Center, Hampton, Virginia

June 2002

The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

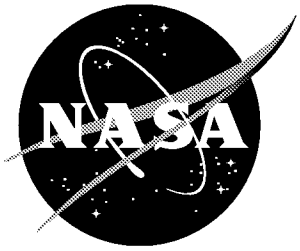
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA STI Help Desk at (301) 621–0134
- Phone the NASA STI Help Desk at (301) 621–0390
- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076–1320

NASA/TM–2002–211738



Some User's Insights Into ADIFOR 2.0D

Daniel P. Giesy
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681–2199

June 2002

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076–1320
(301) 621–0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161–2171
(703) 605–6000

Abstract

Some insights are given which were gained by one user through experience with the use of the ADIFOR 2.0D software for automatic di erentiation of Fortran code. These insights are generally in the area of the user interface with the generated derivative code – particularly the actual form of the interface and the use of derivative objects, including "seed" matrices. Some remarks are given as to how to iterate application of ADIFOR in order to generate second derivative code.

1 Background

Engineering systems are often studied using computer models. These models will typically contain parameters which can be used to represent design decisions (e.g., the size of structural members, the gains in a control system) and/or operating conditions (e.g., temperature, initial conditions, payload mass). The computer model can be used to analyze the engineering system by assigning values to the design and operating parameters and calculating performance measures (such as measures related to the stability of dynamical system) or simulating how the system will behave under the conditions defined by the parameter values. The computer model can be further used for design purposes by programming the computer to select suitable values for the design parameters to achieve desired engineering properties such as strength, performance, etc.

Mathematically, the computer model is a function. The independent variables are the design and operating parameters. The dependent variables are the various system quality metrics (which could be anything from an eigenvalue to a complete response time history) which the computer model computes from the input parameters.

A critically important tool in using the computer model for analysis and design is the knowledge of the sensitivity of the system quality metrics to variation in the design and operating parameters; i.e., the gradient of the function. The sensitivities are, themselves, valuable information to know about the system. A point design of a system might have good quality measures; but, if the system is highly sensitive to parameter variation, small errors in those parameters might change the system into one of poor quality. The gradients also give useful information to some design tools such as optimization techniques.

Once one has a computer model of an engineering system, there are various options as to how to calculate the gradient; for example, numerical differentiation, analytical derivation of the gradient formulae and writing a computer program to evaluate them, or automatic differentiation. Numerical differentiation (e.g., divided forward differences) is inherently inaccurate. Inaccuracies can be reduced to some extent, but only at substantial additional computational expense (such as 2 or 4 point symmetric divided difference formulae). Particularly for realistic computer models of complex engineering systems, derivation of analytic gradients is unacceptably labor intensive and error prone. This makes automatic differentiation by systems such as ADIFOR, the subject of this note, particularly attractive. The derivative values calculated are numerically exact, not approximate; the tedious work of analytic differentiation is done by computer, not by error-prone hand calculations; and the computational time required to calculate gradients in this manner is comparable to the forward difference method of numerical differentiation, which is the fastest, but least accurate, method of numerical differentiation.

A group of researchers in the Guidance and Control Branch (GCB) of the NASA Langley Research Center has for several years now been considering the problem of applying neighboring optimal control techniques to guidance problems (see, for example, [1, Chapter 6]). In this application, it is desired not only to determine the control which results in an optimal trajectory, but also to determine how to apply corrections to that control to take account of perturbations in the operating parameters or drift away from the nominal optimal trajectory and still have an approximately optimal trajectory. Implicit function theory is applied to the gradient of an optimal Lagrangian of a discrete approximation to the problem to represent the sensitivity of design variables to variation in operating parameters. If the Lagrangian of the discretized optimal control problem is represented by $\mathcal{L}$ while the design variables (including the Lagrange multipliers) and the operating parameters are represented, respectively, by v and w, then the optimal value $\mathrm{v}^*(\mathrm{w})$ of v corresponding to a given value of w is found by solving the nonlinear necessary condition equation $\mathcal{L}_v(\mathrm{v}, \mathrm{w}) = 0$ for v (subscripts represent differentiation). Solving this equation using a Newton-Raphson iteration requires not only the gradient $\mathcal{L}_v$, but the also Hessian $\mathcal{L}_{vv}$. Then calculating the sensitivity of $\mathrm{v}^*(\mathrm{w})$ to variation is w requires also the mixed second partial derivative $\mathcal{L}_{vw}$.

This research group has for over a year and a half been using ADIFOR to

find first and second derivatives of components of the computer model which are then used to build up the Lagrangian gradient, Hessian, and mixed second partial derivative. This represents the time frame in which the group has been testing their techniques using computer models which are too complex to allow the practical use of hand-derived first and second derivative formulae.

The Lagrangian gradient is used to find an initial approximation to the nominal optimal trajectory. The Lagrangian gradient and Hessian are used to find the accurately converged nominal optimal trajectory needed as the starting point in the sensitivity calculation. The Lagrangian Hessian and mixed second partial derivative of the Lagrangian are being used in determining the corrections to the controls which are needed to generate a linear approximation to the neighboring optimal trajectories.

ADIFOR has proved to be an enabling technology in carrying this study forward. It was during the course of the study just outlined that the user's insights which are the subject of this note were developed.

2 Introduction

ADIFOR is an acronym for Automatic DIfferentiation of FORtran. "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" [2, p. 3]. Reference [2] tells how to obtain, install, and run ADIFOR. Reference [2] and other information about ADIFOR can be found at the web site `http://www.cs.rice.edu/~adifor/`. It is assumed that the reader of this note is familiar with reference [2]. In speaking of Fortran 77, an effort will be made to use the vocabulary of the standards document ANSI X3.9-1978 [3].

One who reads the phrase "new FORTRAN 77 subroutines that compute the derivatives of f" might suppose that the ADIFOR generated code returns the partial derivatives of the scalar components of f with respect to the scalar components of x. Actually, the ADIFOR generated code expects the user to supply one or more sets of coefficients; and, for each set of coefficients supplied, the ADIFOR generated code returns the linear combination of all the partial derivatives of the scalar components of f with respect to the scalar components of x using this set of coefficients. By proper choice of the coefficients, the individual partial derivatives can be recovered, but other

derivative related information such as directional derivatives can also be returned by the ADIFOR generated code. One of the purposes of the present note is to clarify the instructions given in [2] as to how the user chooses and passes those sets of coefficients and how to associate each returned derivative related datum with the correct set of coefficients and the correct component of f .

In other words, the purpose of this note is to supplement the information given in [2] on how to make use of the "new FORTRAN 77 subroutines that compute the derivatives." Specifically, the user must write a driver routine which calls the ADIFOR generated subroutines that compute the derivatives. This note provides information which the user needs in order to write a driver routine. Clarification is given in two areas related to communication between the user's program and the ADIFOR generated derivative code:

1. Variables in ADIFOR generated subroutine calling sequences or in ADIFOR generated common blocks which were not in the user's original code.

2. The use of "derivative objects".

Caveats: This note will deal exclusively with ADIFOR generation of code to calculate dense Jacobian matrices (i.e., ADIFOR preprocessor option `AD_FLAVOR` has its default value `dense`, [2, Chapter 9]). Anything in this note which is specific to a computer operating system will refer to the UNIX operating system.

Section 3 is a review of the basics of ADIFOR and an overview of its use. Notation is introduced here which will be used throughout the remainder of this note. Section 4 contains a description of syntactic and semantic features of the interface between the ADIFOR generated derivative and the user's driver code of which the user must be aware in order to write that driver code. Section 5 speaks to the information content of Fortran data objects involved in the interface between ADIFOR generated derivative code and the user's driver code. Particular attention is given to the initialization of seed matrices and the interpretation of data returned by ADIFOR generated code in the derivative objects for the dependent variables. A small example is presented in section 6 to illustrate some of the concepts presented in the preceding part of this note. Section 7 is devoted to remarks on applying ADIFOR to ADIFOR generated first derivative code to generate code which

will calculate second derivatives. This presents some difficulties which one would not expect to encounter in processing ordinary code with ADIFOR.

3 Review of ADIFOR basics and overview of usage

The starting point for using ADIFOR to generate code to calculate derivatives is a collection of Fortran subprograms written in (almost) ANSI standard Fortran 77 source code contained in one or more files (see [2, Section 3.3, "Acceptable FORTRAN 77 Source Files"]) which defines the dependency of the function(s) to be differentiated on some variable(s). For purposes of this discussion, this functional dependency will be represented abstractly as f(x) where it is to be understood that x might represent one or more input scalar values and f might represent one or more output scalar values. The scalar components of this abstract f might be contained in in a single Fortran variable or array, or might be the result of collecting together any number of Fortran variables and/or arrays of arbitrary size and number of dimensions. The same is true of the abstract x. The Fortran variables and arrays which make up f are called the *dependent variables* and the Fortran variables and arrays which make up x are called the *independent variables.* However, the calculation of f must be accomplished by a single call to a single subroutine subprogram (not function subprogram) which will be called the *top-level routine.*

In addition to the restrictions in [2, Section 3.3], the user must insure that no conflicts arise between names in the user's code and names in the ADIFOR generated code. The ADIFOR generated Fortran code is written in files. In the course of doing this, ADIFOR creates many names such as file names, subprogram names, variable names, parameter names, and common block names. Many of these ADIFOR generated names start with the same two characters, the *ADIFOR naming characters.* The default naming characters are "`g`" and "`_`"; these can be changed by ADIFOR preprocessor options `AD_PREFIX` and `AD_SEP`, respectively. Some of these ADIFOR generated names (examples include `g_p_` and `g_pmax_`) are hardwired in ADIFOR. Other ADIFOR generated names are created by prepending the ADIFOR naming characters to a name which comes from the user's input. If a file named `code.f` is processed, ADIFOR writes a file named `g_code.f`.

If a subroutine named `xdot` is processed, ADIFOR produces a subroutine named `g_xdot`. If a variable named `x` is processed, ADIFOR uses the name `g_x` for its derivative object and, sometimes, the variable name `ldg_x` for a variable to hold the value of the first dimension of this derivative object (the rule here seems to be that the letters `ld` are prepended to the name of the derivative object – this is the only case of which the author is aware in which the ADIFOR naming characters are used other than at the beginning of the name).

Since ADIFOR generates code by adding new lines of code to the user's code (with the exception that the user's `SUBROUTINE` or `FUNCTION` line is modified), the generated code contains all the user's original names and also the ADIFOR generated names. The user must select ADIFOR naming characters so that no naming conflict arises between the user's names and the ADIFOR generated names. The same caution also applies to file names.

Except for Fortran 77 intrinsic functions (like `sin` and `log`), all functions and subroutines on which the top-level routine depends either directly or indirectly must be included in the collection of subprogram files passed to ADIFOR, [2, §3.2, p. 17]. This includes routines which the user normally calls from libraries (e.g., LAPACK [4]). The values of the Fortran variables and/or arrays which make up `x` may be passed into the top-level routine by placing the variables and/or arrays in the calling sequence of the top-level routine, or in common blocks, or some of each. The variables and/or arrays which make up `f` may be returned to the calling routine in the same manner. Although ADIFOR permits multiple subprograms to reside in a single subprogram file, the experience gained during the work mentioned in section 1 indicates that there are advantages to placing each subprogram in its own file. A file containing multiple subprograms can be processed into files each containing a single subprogram using the UNIX utility `fsplit`.

The presence of `Tab` characters in Fortran source code passed to ADIFOR has been know to make ADIFOR crash. `Tab` characters can be introduced into text files such as those which contain Fortran source code by text editors such as the UNIX editor `vi` without deliberate action on the part of the code writer. If this problem arises, the UNIX utility `expand` can be used to replace `Tab` characters in a file by an equivalent number of spaces.

The user's wishes are then made known to ADIFOR through the provision of two items of information. The first is a *composition* file which lists the names of those source code files. The rules governing composition files are given in [2, §3.2, p. 17]; for an example, see [2, Figure 4.3, p. 22]. The second

is a list of ADIFOR preprocessor options which, for purposes of this note, will be assumed to be contained in a *script* file (ADIFOR also allows the option of passing some or all of them as command line parameters). The use of preprocessor options is covered in [2, §3.1, pp. 17ff] with an example in [2, Figure 3.1, p. 16].

At a minimum, the script file must set 5 preprocessor options:

`AD_PROG` This option must be set to the name of the composition file; e.g.:
`AD_PROG=example.cmp`

`AD_TOP` This option must be set to the name of the top-level routine; e.g.:
`AD_TOP=calcxhat`

`AD_IVARS` This option must be set to a (comma separated) list of the independent variables, the Fortran variable and/or array names which make up x; e.g.:
`AD_IVARS=w1,w2,y`

`AD_DVARS` This option must be set to a (comma separated) list of the dependent variables, the Fortran variable and/or array names which make up f .

`AD_PMAX` This option must be set to the integer which will be used in ADIFOR generated derivative code as the value of the `INTEGER` parameter (default name "`g_pmax_`") which it uses as the first dimension of many of the derivative objects. Proper choice of this option is based on the user's determination of just exactly what derivative-related information the user wishes the ADIFOR generated code to calculate. How to choose `AD_PMAX` will be explained at more length in section 5 of this note.

Even if ADIFOR is already installed on the user's computer, the user needs to be familiar with the information in [2, Section 2.1.1, "Unix Installation and Configuration"]. This gives information on how the user's computer environment must be configured in order to run the `Adifor2.1` command which generates the Fortran code for derivative calculation. On the author's (Linux) computer, this is accomplished by executing the following commands, either by including them in a log-in initialization file (e.g., `.cshrc`) or by executing them directly in a window where the ADIFOR processing is to take place:

```
setenv AD_HOME /home/dgiesy/ADIFOR_2.0D/ADIFOR2.0D
setenv AD_LIB /home/dgiesy/ADIFOR_2.0D/ADIFOR2.0D.PGI.lib
setenv AD_OS Linux86
setenv PATH $AD_HOME/bin:$PATH
if ($?MANPATH == 0) then
setenv MANPATH '':$AD_HOME/man
else
setenv MANPATH $AD_HOME/man:$MANPATH
endif
```

(If these commands are places in a script to be executed in the working window, the script must be executed using the UNIX `source` command.) Of these commands, the user must tailor the first three to the user's own computer environment; the remainder are generic. Once the user's computer environment has been configured, the files with the derivative code are generated using a command of the form:

`% Adifor2.1 AD_SCRIPT=`*script_file_name*

The user should be aware that the execution of the ADIFOR command generates many of files in the directory in which it is executed, and also generates a subdirectory of the working directory whose default name is `AD_cache`. Assuming the default ADIFOR naming characters "`g_`" are used, the derivative calculating code which was generated by the ADIFOR run is contained in files which have names of the form `g_*.f` (the `*` in `g_*.f` is the UNIX "wild card" character and stands for an arbitrary character string). These files are useful to the user. The remaining files and subdirectory might pose a problem if the user needs to rerun ADIFOR (for example, after editing some of the original source code or when making a second run to generate second derivative code). The `AD_cache` subdirectory is supposed to contain information which provides an incremental reprocessing capability, [2, p. 40]. However, the GCB group has found a situation (changes to an `INCLUDE` file) where ADIFOR failed to detect that a subroutine needed to be reprocessed. The people at Rice University who maintain ADIFOR have been notified of this problem and will, presumable, fix it in future versions.

Until then, and for safety's sake and at a possible cost in processing time, the procedure of removing all ADIFOR generated auxiliary files and subdirectories after an ADIFOR processing run has been adopted. As long as the default ADIFOR naming characters, `g_`, are used, this can be accomplished on a UNIX system by the single command:

```
% \rm -r AD_cache/ g_*.[aA]* .*.f *.f~
```
Note that every period and tilda in this command is significant – without them, some important files might be deleted.

The user then examines the derivative code generated by ADIFOR for information needed to make use of it. The user must:

1. Determine which of the AIDFOR generated routines and which (if any) of the user's original routines in the configuration file are needed by the derivative counterpart of the top level routine. A software tool (FTNCHEK) which can be used to generate the subprogram dependency tree of a subroutine is discussed in the Appendix.

2. Determine what the derivative objects are in the calling sequence of the top level routine and in ADIFOR generated common blocks.

3. Determine the calling sequence of the derivative counterpart of the top level routine.

4. Write driver code to interface with the ADIFOR generated derivative code.

5. Incorporate the driver code and the ADIFOR generated derivative code into an application program.

6. Compile, link and load, and run the application package.

Information from stage 1 is used to determine which subroutines must be examined at stage 2. Information from stages 2 and 3 are necessary to stage 4. Information from stage 1 can also be used to avoid compiling and linking unnecessary subprograms at stage 6. Once the application program with the incorporated driver code and ADIFOR generated derivative code is compiled, the configuration information set previously is also used during the "link and load" phase to access the ADIFOR run-time libraries needed by the ADIFOR generated derivative code.

4 Semantics of the interface to ADIFOR generated code

For each subroutine in each file named in the user's composition file (or, at least, each subroutine which is determined by ADIFOR to contribute to the

dependency of f on x), ADIFOR generates a derivative subroutine whose name is gotten by prepending the ADIFOR naming characters to the subroutine name. The purpose of this section is to give the user information about how to use the ADIFOR generated subroutines. The most common scenario is that the user only needs to know how to call the derivative subroutine corresponding to the top-level routine, and this is the case which will be emphasized here.

The reader is assumed to be familiar with [2, Section 2.3.2, "Variable Nomination"]. There, the notion of an *active* variable is defined; the collection of active variables contains any of the Fortran data items (scalar variables or arrays) which fall on the chain of dependency between the independent data items and the dependent data items (including the independent and dependent data items themselves). Constraints of syntax may demand that other variables also be declared active (this will be illustrated in section 6). ADIFOR scans the code to be differentiated and determines which variables it will consider to be active. All remaining variables are called *passive*.

The user needs to know which of the variables involved in the interface between the user's driver code and the ADIFOR generated derivative code are active. It is difficult to tell the user how to determine in advance of actually running ADIFOR which of the variables will be active. For example, it is possible for a variable which does not depend on the independent variables to be declared active by ADIFOR. This might happen if the top-level routine calls some subroutine more than once, and if a truly active parameter is passed to the subroutine in one of the calls, then for syntactical consistency every parameter passed to the subroutine in the same parameter position in other calls must be treated as active even if it is not in the chain of dependency between the independent variables and the dependent variables. Thus, the user needs to process the code to be differentiated with ADIFOR and examine the generated derivative code to determine which variables are being treated as active.

With each active Fortran scalar or array data item, ADIFOR associates a vector or array which is also named by prepending the ADIFOR naming characters to the name of the Fortran data item. In reference [2], these new Fortran data objects are variously called "directional gradient objects", "gradient objects", "derivative objects", or (if the active Fortran data item was declared with ADIFOR preprocessing option `AD_IVARS` to be an independent variable) "seed matrices". So, if an active Fortran data object is named `xhat`, then its derivative object will be named (under default options) `g_xhat`. This

section is devoted to providing the reader with information about which of these derivative objects need to be declared in the user's driver software and/or used in calls which the user makes to the ADIFOR generated derivative code. Section 5 will discuss the information contained in these derivative objects.

Every derivative object is an array which has one more dimension than its *parent*, the Fortran data object which gave rise to it. Thus, each active scalar has a one-dimensional derivative object, i.e., a vector; each active two-dimensional matrix has a three-dimensional array for its derivative object; and so on. If the parent of a derivative object was passed to the top-level routine as a parameter in its argument list, the first dimension of the derivative object is specified by the user supplied driver code. ADIFOR specifies the first dimension of all other derivative objects. These specifications will be discussed in section 5. The remaining dimensions (if any) are copied directly from the dimensions of its parent.

Active variables may be partitioned into three categories:

1. Variables which are local to the top-level routine or to one of the subprograms subordinate to it.

2. Variables which are global by virtue of being located in common blocks.

3. Variables which are dummy arguments to the top-level routine.

What the user needs to know about and do with a derivative object depends on which of these categories its parent belongs to.

4.1 Local variables

The user does not need to do anything about these variables.

Caveat: This assumes that the user calls only the top-level routine. The more advanced user who is calling one of the subordinate derivative subroutines directly must treat any active variables appearing in its dummy argument list under category 3.

4.2 Variables in common

The user must be concerned about any common block which is declared both in the user's driver program for the top level subroutine and in any of the

subroutines named in the composition file to be passed to ADIFOR. The user must determine the answers to two questions:

1. Does the common block contains any active variables? If so,

2. does the user need either to initialize any of the derivative objects for active variables in the common blocks or to access any values in any of the derivative objects after the derivative code has executed?

The second question will be addressed in section 5. To answer the first question, the user must search the ADIFOR generated derivative code for the corresponding ADIFOR generated derivative object common block. The name for the ADIFOR generated block is formed from the name for the user's original common block by the usual prepending of the ADIFOR naming characters.

If the answer to the first question is "yes", almost certainly the answer to the second question is also "yes". If the answer to both questions is "yes", the user must declare the ADIFOR generated derivative object common block in the driver code. This means not only copying the ADIFOR generated `common` statement into the driver program but also copying the size declarations for all the data arrays in the common block. In addition, the declarations

```
integer g_pmax_
parameter (g_pmax_ = ...)
```

which occur in the ADIFOR generated code must also be included in the driver program. The parameter `g_pmax_` is used by ADIFOR as the first dimension of gradient objects in common blocks (and of gradient objects whose parents are local variables). Note that if more than one common block is involved, they all use the same declarations for `g_pmax_` and the `g_pmax_` declarations should be copied into the driver program only once.

4.3 Dummy arguments to the top-level routine

There is a connection between the dummy argument list of a user-supplied subroutine which has been passed to ADIFOR and the dummy argument list of the corresponding derivative subroutine generated by ADIFOR. The first argument to the derivative subroutine is an ADIFOR generated `INTEGER` variable whose default name is `g_p_` which the user must set when the derivative subroutine is called by the user's driver program. The value to choose for `g_p_` is explained in section 5.

The remaining dummy arguments of the derivative code are directly related to the dummy arguments of the original code. The nature of the relation depends on whether the original dummy argument was declared to be active or passive by ADIFOR. A passive dummy argument of the original code is simply copied directly into the dummy argument list of the derivative code without any change or additional arguments.

An active argument of the original code is also copied directly into the argument list of the derivative code. However, two more arguments are put in the argument list of the derivative code directly after this copy of the original argument. Perhaps this is best illustrated by example. If the active argument is `xhat`, then at the place in the argument list of the original code where `xhat` appears, the derivative code has the three arguments:

```
xhat, g_xhat, ldg_xhat
```

This again assumes that the default naming characters were used. If ADIFOR had processed this example under preprocessing options `AD_PREFIX=h` and `AD_SEP=$`, the three arguments in the derivative code would have been:

```
xhat, h$xhat, ldh$xhat
```

Suppose further that in the original code `xhat` has been declared:

```
double precision xhat(5,3)
```

The user must decide what the first dimension of `g_xhat` is going to be – this subject is covered in section 5. Suppose for the purpose of illustration that the first dimension has been chosen to be 20. Then, in the driver code, the user must declare `g_xhat` to be a double precision array of dimensions (20,5,3). Further, prior to the call to the derivative subroutine, `ldg_xhat` must have been set to 20 (or the literal value 20 must be used in the actual call).

This must be repeated for each active argument in the calling sequence of top-level routine (or whatever derivative subroutine the user wishes to call). Notice that this depends on which of the dummy arguments to the original top-level subroutine have been declared active by ADIFOR. Thus, the user must first apply ADIFOR to the code to calculate f (x) before the user can determine all the needed information to write the driver program for the derivative code.

Observe that every parameter of the original subroutine appears in the calling sequence of the derivative subroutine. Furthermore, all of the code functionality of the original subroutine is duplicated in the derivative subroutine. This means that when the derivative subroutine is called, then all the original subroutine calculations are performed, and returned to the user just

as the original subroutine did. Some derivative information is also calculated and returned. This is the subject of section 5.

5 Information in ADIFOR generated Fortran data items

This section supplements and elaborates on [2, Section 2.4, "Functionality of ADIFOR 2.0-Generated Code"] and [2, Appendix A, "Seed Matrix Initialization"]. Information will be given as to which of the derivative objects needs to be initialized, and to what, prior to a call to ADIFOR generated derivative code; how to determine the proper value for the ADIFOR preprocessor option `AD_PMAX` and the proper value to pass to the ADIFOR generated dummy argument `g_p_`; and what derivative information is being returned by the ADIFOR generated derivative code.

5.1 Scalar f, vector x

For simplicity, first consideration will be given to the case that the dependent variable `f` is a scalar residing in Fortran variable `f`, and that the vector `x` of independent variables resides in a single Fortran vector array `x`. Assume that the top-level routine starts out:

```
subroutine func(n,x,f)
integer n
double precision x(*), f
```

Further assume that this subroutine has been processed by ADIFOR with `x` specified as the independent variable, `f` as the dependent variable, and with the option `AD_PMAX` set to 20. Then the user's interface to the derivative code could well be defined by the Fortran statements:

```
subroutine g_func(g_p_, n, x, g_x, ldg_x, f, g_f, ldg_f)
  integer n
  double precision x(*), f
  integer g_pmax_
  parameter (g_pmax_ = 20)
  integer g_p_, ldg_f, ldg_x
  double precision g_f(ldg_f), g_x(ldg_x, *)
```

The linear combination of partial derivatives which is the "natural" output of derivative related information from the ADIFOR generated derivative subroutine `g_func` for this case is the vector inner product between the gradient of f and a vector of coefficients, say v, which contains the same number of elements as the vector x of independent variables. These coefficients are passed by the user to the derivative subroutine by storing them as a row of the gradient object `g_x` of the independent variable. The function of the first dimension of the gradient objects is to provide storage for the sets of coefficients used by the ADIFOR generated subroutines in the linear combinations of partial derivatives which they compute and to provide storage for intermediate and final computed results which use theose coefficient sets. **The freedom to select what will be in the rows of the seed matrices (the gradient objects for the independent variables) gives the user a great deal of flexibility in what derivative related information will be calculated.**

If the user wants the partial derivative of f with respect to one of the scalar variables in x, the user can choose v to be the "elementary" vector the same size as x with all zeros except for a single 1 in the proper position. If the user wants a directional derivative of f, the user can choose v to be a unit vector pointing in the desired direction. If the user already knows $\partial x/\partial w$ and wants to calculate $\partial f/\partial w$ by the chain rule:

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial x}\frac{\partial x}{\partial w},$$

the user can find one component of $\partial f/\partial w$ by choosing v to be the transpose of a column of $\partial x/\partial w$. In fact, the user can calculate all of $\partial f/\partial w$ by setting each row of the derivative object of x to the corresponding column of $\partial x/\partial w$; i.e., `g_x` is set to $(\partial x/\partial w)'$, the transpose of $\partial x/\partial w$. If this is done and `g_func` is called, `g_f` will contain $(\partial f/\partial w)'$ (*cf.* [2, p. 10, p. 46]).

The user can do this because one of the features of the derivative code generated by ADIFOR is that, in one call, it will calculate the inner product of the gradient of f with as many vectors as the user wishes. When the user calls an ADIFOR generated subroutine, the user tells the subroutine how many linear combinations of the gradient the user wishes the code to calculate by the user's setting of the actual argument value corresponding to the dummy argument `g_p_`. Suppose the user wishes the complete gradient of f. A natural way to do this is to initialize `g_x` to the identity matrix. Since row i of the identity matrix is a vector with a 1 in its ith component and zeros

elsewhere, its inner product with the gradient of f is $\partial f / \partial x_i$. This value is returned to the user by g_func in vector g_f as array element g_f(i).

The user could even combine the calculation of the full gradient with one or more directional derivatives or other gradient-vector inner products in a single call by properly initializing the seed matrix (gradient object of the independent variable). The only caution here is that the larger the seed matrices are, the longer the derivative code takes to execute and the more memory it requires.

Choice of AD_PMAX: The value the user assigns to the ADIFOR preprocessing option AD_PMAX is used by ADIFOR as an upper bound for the number of linear combinations of partial derivatives which the derivative code will be required to compute on any one call. The derivative objects which are passed to derivative code through the calling sequence reside in arrays which are declared by the user to have whatever first dimension the user chooses, and that first dimension is passed to the derivative code through calling sequence parameters like ldg_x and ldg_f in the example just given. However, ADIFOR must assign the first dimension to derivative objects whose parents are in common blocks or are local to the original code. The user's AD_PMAX value is assigned to ADIFOR generated INTEGER parameter g_pmax_, and ADIFOR uses this parameter for the first dimension of common block and local variable gradient objects. The gradient code will check if the number of inner products requested in g_p_ is greater than the value of g_pmax_. If it is, then ADIFOR generated derivative objects for local and common block variables are not large enough for the requested calculation, an error condition exists, and the ADIFOR generated code will stop execution.

Recapitulation: In order to generate code to calculate derivative related values of a scalar valued function f of a single vector variable x and put it to use in an application, the user must:

1. Prepare the ADIFOR script and composition files. The user must know an upper bound on the number of gradient-vector inner products which the user will want the gradient code to calculate on any one call, and assign that number to the ADIFOR preprocessing option AD_PMAX.

2. Use ADIFOR to generate the gradient code. The user must:

   (a) Prepare the computer environment for ADIFOR by setting the proper environment variables as in [2, Section 2.1.1, "Unix Installation and Configuration"].

(b) Execute ADIFOR by a command of the form:
`% Adifor2.1 AD_SCRIPT=`*script_file_name*
For ADIFOR 21.D, it is probably safest to do this in a working directory from which has been removed any files and subdirectories generated by any earlier execution of `Adifor2.1`.

3. Write the driver code for the gradient code. The user must:

   (a) Examine the ADIFOR generated gradient code to determine the exact form of the interface. This includes the calling sequence of the gradient counterpart of the top-level routine and any ADIFOR generated common blocks containing any gradient object.

   (b) Initialize gradient objects according to section 5.5. In particular, for each call to the gradient code, determine what gradient-vector inner products are desired. Declare arrays `g_x` and `g_f` of large enough first dimension and (for `g_x`) appropriate second dimension. Set `g_p_` to the number of these products. Initialize the rows of `g_x` with the multiplier vectors for these products.

   (c) After the call to the gradient code, harvest the result corresponding to row `i` of `g_x` from `g_f(i)`.

4. Compile and link the application using the ADIFOR generated derivative code and referencing the ADIFOR run-time libraries, etc. [2, Section 2.1.1, "Unix Installation and Configuration"] in the link phase.

Notation convention: This note has been written (and, for the most part, will continue to be written) as if actual arguments in calls to ADIFOR generated gradient code have the same names as the dummy arguments in the code being called. Of course, the user can name variables in the driver code with any legal Fortran 77 variable name. The use of the same name for actual arguments in the driver code and dummy arguments in the ADIFOR generated derivative code is just for clarity of identifying which goes with which.

5.2 Scalar f, arbitrary independent variables

Next, consideration is given to the case that the dependent variable is a single scalar variable, but the independent variables for differentiation are

distributed over several Fortran data objects which may be a mixture of any number of scalar variables, vectors, matrices, and/or arrays of higher dimension. When an illustration is needed, the example of `subroutine func1` will be used as a top-level routine. This subroutine starts out:

```
      subroutine func1(n,x,y,z,f)
      integer n
      double precision x(*), y, z(3,5), f
```

Assume that the function of the dummy argument `n` is to pass to `func1` the actual length of the vector being passed in argument `x`. Assume that this subroutine has been processed by ADIFOR with the independent variables specified to be `x`, `y`, and `z`; the dependent variable specified to be `f`; and the option `AD_PMAX` set to 35. Then the user's interface to the derivative code could well be defined by the Fortran statements:

```
      subroutine g_func1(g_p_, n, x, g_x, ldg_x, y, g_y, ldg_y,
     *z, g_z, ldg_z, f, g_f, ldg_f)
         integer n
         double precision x(*), y, z(3, 5), f
         integer g_pmax_
         parameter (g_pmax_ = 35)
         integer g_p_, ldg_f, ldg_y, ldg_x, ldg_z
         double precision g_f(ldg_f), g_y(ldg_y), g_x(ldg_x, *),
     *   g_z(ldg_z, 3, 5)
```

The "natural" output of derivative related information from the ADIFOR generated derivative subroutine `g_func1` for this case is again a collection of linear combinations of the partial derivatives of `f` with respect to each variable or array element in the independent variables. But, because of the greater complexity of the data structures containing the independent variables as opposed to the previous single vector case, some clarification is in order.

The gradient of `f` has one scalar component for each variable and each array element of an array in the list of independent variables. It is convenient to think of this gradient as being stored in a collection of variables and arrays which is *conformable* with the independent variables; by which is meant that the hypothetical collection of gradient variables and arrays matches the list of independent variables one for one in number of dimensions and size. (An inspection of ADIFOR generated code reveals that the gradient is not, in fact, stored in this manner – this is just a mental picture.) Referring to the

example `g_func1`, lets give names to these hypothetical variables and arrays: `dfdx` represents a vector the same length as `x`, `dfdy` represents a scalar, and `dfdz` represents a 3 by 5 matrix. Then, for example, `dfdz(2,3)` is imagined to contain the partial derivative of `f` with respect to `z(2,3)`.

If `g_func1` is to form a linear combination of the partial derivatives of `f`, it needs coefficients for that linear combination. Think of these coefficients as being stored in another collection of Fortran data objects which is conformable with the independent variables. This is not hypothetical, it actually happens; and it is the user who provides this data to `g_func1`. For each set of coefficients for which the user wants the linear combination, the user picks an integer to use as a first index in the gradient objects for the independent variables and stores the coefficients in the gradient objects using that integer as the first subscript. Thus, the user's third set of coefficients would be stored in locations `g_x(3,i)` for `i` = 1, ..., `n`; `y(3)`; and `g_z(3,i,j)` for `i` = 1,...,3 and `j` = 1, ..., 5.

Some vocabulary from the lexicon of Fortran 95 (see, e.g., [5, Section 6.4.4, pp. 165ff]) is introduced here:

The **rank** of an array is the number of dimensions in the array. For consistency, a scalar is considered to have rank 0.

The **extent** of an array in one of its dimensions is the number of elements in that dimension.

The **shape** of an array is an integer vector containing the same number of elements as its rank, where each element of the vector is the extent in the corresponding dimension. For consistency, the shape of a scalar is a vector of length 0, sometimes referred to as the empty vector.

The **size** of an array is the product of its extents, i.e., the total number of elements in the array.

An **array element** is any one of the scalar elements which make up the array.

An **array section** of an array is defined by specifying one non-empty subset of the allowable subscripts for each dimension and forming the subarray which uses the subscripts from these subsets in every possible way. If these subscript subsets contain only one element each, then

only an array element is specified, and the word "section" is not used for this case.

Using this vocabulary, the relationship of a derivative object to its parent can be restated. The rank of the derivative object is larger by one than the rank of its parent, and its shape results from prepending a new dimension to the shape of the parent. That new dimension is either supplied by the user (in the case that the parent of the derivative object is a dummy argument in the calling sequence of the top level routine) or is set by ADIFOR as the value of the ADIFOR preprocessor option `AD_PMAX`.

A terminology is introduced here. By the `i`*th layer* of a gradient object is meant the array section (or, possibly, array element) of the gradient object defined by fixing its first index at the value `i` while allowing any other indices to take on all of their allowable values. For two-dimensional arrays such as were used in the example in section 5.1, *layer* (as used here) is synonymous with *row*. Each layer of a gradient object has exactly the same size and shape as its parent object. Using this terminology, the coefficients of the third linear combination of the previous paragraph are stored in the third layer of the gradient objects for the independent variables.

The linear combinations are now formed by multiplying, for each `i` between 1 and the user input value of `g_p_`, each of the hypothetical gradient variables and arrays element by element with the corresponding data found in layer `i` of the gradient objects for the independent variables. These products are then added up, and the result stored in layer `i` of `g_f`. The final effect is **as if** the following **hypothetical** Fortran 77 code fragment had been executed:

```
      integer g_i_
*
*     Code to calculate the gradient values and place them
*     in the hypothetical variables and arrays
*     "dfdx", "dfdy", and "dfdz".
*
      do g_i_ = 1, g_p_
        g_f(g_i_) = 0.0d+00
        do i = 1, n
          g_f(g_i_) = g_f(g_i_) + dfdx(i)*g_x(g_i_,i)
        enddo
        g_f(g_i_) = g_f(g_i_) + dfdy*g_y(g_i_)
```

```
      do i = 1, 3
        do j = 1, 5
          g_f(g_i_) = g_f(g_i_) + dfdz(i,j)*g_z(g_i_,i,j)
        enddo
      enddo
    enddo
```

As before, the user must decide which linear combinations involving the partial derivatives of f to request from the derivative code. And, as before, the user can find the partial derivative of f with respect to a chosen one of its independent scalar variables by setting a layer of the independent variable gradient objects to all zeros except for a single 1 in the position corresponding to that one scalar variable. However, if the user wishes a complete gradient of f, then the user must contend with the limitation of ADIFOR that the layers of the gradient objects are indexed by a single integer variable, so that the user must arbitrarily impose an enumeration on all of the desired partial derivatives.

This will be illustrated by reference to the example from earlier in this section. The problem is to make a call to `g_func1` which returns the derivatives of `f` with respect to `y` and each of the scalar components of `x` and `z`. The mathematical symbol n is used to to represent the number of components of `x` actually in use, and this value is contained in Fortran variable `n`. So, the variable `x` contains n scalar elements, the variable `y` contains 1 scalar element, and the variable `z` contains 15 scalar elements.

Imposing an enumeration on the desired $n+16$ partial derivatives means selecting one of the $(n+16)!$ permutations of these $n+16$ scalar elements. For purposes of illustration, the order chosen here is to first take the elements of `x` in index order, then the single element `y`, then the elements of `z` as Fortran traditionally stores them, i.e., in column major order. The variables are then considered to be in the order `x(1)`, ..., `x(n)`, `y`, `z(1,1)`, `z(2,1)`, `z(3,1)`, `z(1,2)`, ..., `z(3,5)`. This means that the user must initialize layer 1 of the seed matrices so that the linear combination of partial derivates returned is just $\partial f / \partial x_1$. Layer $n+1$ is set to return $\partial f / \partial y$, layer $n+2$ to return $\partial f / \partial z_{1,1}$, etc. Further assume that all calls to the code have $n \leq 30$, so $n+16 \leq 46$. A minimal value of ADIFOR preprocessor option `AD_PMAX` for this application is 46. The following Fortran 90 code fragment shows how `g_x`, `g_y`, and `g_z` can be initialized to instruct `g_func1` to calculate the gradient of f:

```
!
! The gradient objects have previously been given dimensions
! g_x(46,30), g_y(46), g_z(46,3,5), and g_f(46)
```

```
!
! At this point, code should be included to initialize the variables
! n, x, y, and z as they would have been initialized for a call to
! the parent routine, subroutine func1.
!
! The following lines initialize the ADIFOR generated variables:

g_p_ = n+16
ldg_x = 46
ldg_y = 46
ldg_z = 46
ldg_f = 46

! For starters, completl y zero out the gradient objects:

g_x = 0.0d+00
g_y = 0.0d+00
g_z = 0.0d+00
g_f = 0.0d+00                    ! This one should be unnecessary, but i s
                                 ! done as a safety measure.

! Set selected elements of the seed matrices to 1.0 so that the
! desired derivative val ues will be returned in g_f.
!
! The INTEGER variable "ln" represents the derivative object layer
! number and steps through the enumeration of the derivatives.

ln = 0
do i = 1, n
  ln = ln+1
  g_x(ln,i) = 1.0d+00
enddo
ln = ln+1
g_y(ln) = 1.0d+00
do j = 1, 5
  do i = 1, 3
    ln = ln+1
    g_z(ln,i,j) = 1.0d+00
  enddo
enddo

! That's it. Now make t he call that calculates the derivatives.

call g_func1(g_p_, n, x, g_x, ldg_x, y, g_y, ldg_y, &
& z, g_z, ldg_z, f, g_f, ldg_f)
```

```
! Now the first n+16 elements of g_f contain the partial derivatives of
! f with respect to the variables x(1), ..., x(n), y, z(1,1), z(2,1),
! z(3,1), z(1,2), ..., z(3,5) in that order.
```

5.3 Arbitrary dependent variables, arbitrary independent variables

Finally, consideration is given to the case that the dependent variables and the independent variables for differentiation are each distributed over several Fortran data objects which may be a mixture of any number of scalar variables, vectors, matrices, and/or arrays of higher dimension. Conceptually, this is actually a fairly small step from the case considered in section 5.2. Basically, what was done in the case of a scalar dependent variable is done in the present case on an element by element basis to each element of each of the arrays of dependent variables.

Specifically, suppose that two of the dependent variables are specified by the Fortran code lines:

```
double precision f1, f2(5,3,7)
common /out1/ f1, f2
```

Then, assuming that the ADIFOR script file included the option specification `AD_PMAX=244`, the corresponding derivative object will be specified by the code lines:

```
integer g_pmax_
parameter (g_pmax_=244)
double precision g_f1(g_pmax_), g_f2(g_pmax_,5,3,7)
common /g_out1/ g_f1, g_f2
```

Everything which was said in section 5.2 about the relation of `g_f` to `f` and the gradient objects for the independent variables applies without modification to the relation of `g_f1` to `f1` and the gradient objects for the independent variables of the present example. The same is also true on an element by element basis for `g_f2`. So, for example, after the derivative code of this example is called, `g_f2(131,2,3,4)` contains the linear combination of the partial derivatives of `f2(2,3,4)` with the coefficients found in layer number 131 of the gradient objects of the independent variables input by the user to the derivative code.

5.4 Gradient of a function of several Fortran vector variables

This section focusses on the case that the user wishes to calculate the Jacobian of the dependent variables and that the independent variables reside in several Fortran scalar and/or vector variables. This is a special case of the case covered in section 5.3. This case is common enough, and there is a seed matrix generation technique which is elegant enough, that it deserves special mention. This section provides expansion and (hopefully) clarification of ideas presented in [2, Appendix A.3].

The general technique will be illustrated by an example. Suppose that the independent variables are `w`, `x`, `y`, and `z` where `w` is a vector of length 2, `x` and `y` are scalars, and `z` is a vector of length 3. Then, counting each component of a vector variable, there are a total of 7 independent scalar variables, and to represent the gradient of the dependent variable(s), gradient objects will need 7 layers. Declare the seed matrices to have only the necessary dimensions: `g_w` has dimensions (7,2), `g_x` and `g_y` each have dimension (7) (which, for purposes of this discussion, should be thought of as matrices of dimensions (7,1); i.e., column vectors), and `g_z` has dimensions (7,3). As stated in section 5.2, an enumeration must be imposed on the 7 scalar components of these 4 Fortran variables. A natural way to enumerate the scalar components is to picture the vector and scalar Fortran variables as column vectors (the scalars being vectors of one component) and imagine them to be stacked up in the order they have been being considered; i.e., `w`, `x`, `y`, `z`. Each component of this stack corresponds to one layer in each of the gradient objects. The seed matrices are to be initialized so that layer `i` of each dependent variable gradient object contains the partial derivative of that dependent variable with respect to component `i` of this imaginary stack of independent variables.

The easiest way to picture the proper initialization of the seed matrices to accomplish this is to imagine the seed matrices placed side-by-side in the already established order:

`g_w`, `g_x`, `g_y`, `g_z`

Collectively, these matrices cover a 7 by 7 matrix. It is no coincidence that the collective matrix is square – going through this exercise with an arbitrary collection of vector and scalar independent variables will always result in a collection of seed matrices whose horizontal juxtaposition will collectively amount to a square matrix whose order is exactly the number of individual scalar components from all the independent variables collectively.

The correct initialization of the seed matrices is the one which makes this horizontal juxtaposition of seed matrices be the identity matrix. In the example being considered here, the seed matrices

```
g_w, g_x, g_y, g_z
```

should be initialized to

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

respectively.

Some programmers have found it a little tricky to get the ones in just the correct locations of the seed matrices. This is particularly true in case the dimensions are parameterized by variables instead of being predetermined constants as they are in the example which is being used in this section. An alternative to individually initializing seed matrices for each independent variable is to initialize a single identity matrix and then use pieces of it for the seed matrices.

To continue the example of this section, suppose that a 7 by 7 matrix `dpi` (mnemonic for "double precision identity") has been defined in the user's driver code and set to the identity matrix. Suppose that the `SUBROUTINE` declaration line of the top-level subroutine is:

```
      subroutine tls(w,x,y,z,f)
```

Suppose that the interface to the ADIFOR generated counterpart is:

```
      subroutine g_tls(g_p_,w,g_w,ldg_w,x,g_x,ldg_x,y,g_y,ldg_y,
     *                 z,g_z,ldg_z,f,g_f,ldg_f)
```

Then, once `w`, `x`, `y`, and `z` have been initialized, a Fortran 77 call which would calculate the desired gradient of `f` would be:

```
      call g_tls(7,w,dpi(1,1),7,x,dpi(1,3),7,y,dpi(1,4),7,
     *           z,dpi(1,5),7,f,g_f,7)
```

This passes the proper submatrices of the identity matrix for each of the seed matrices. Using Fortran 90 syntax, the same task could be accomplished with the call:

```
call g_tls(7,w,dpi(:,1:2),7,x,dpi(:,3),7,y,dpi(:,4),7,&
 &         z,dpi(:,5:7),7,f,g_f,7)
```

Here, explicitly specified *sections* of the `dpi` matrix are passed in each seed matrix argument slot.

5.5 Preparing to call ADIFOR generated subroutines

In this section, consideration is given to initializations the user must make in the driver code to call the top-level gradient subroutine generated by ADIFOR.

It has already been seen that all data objects which were present in the interface between higher level code and the user's top-level routine are also present in the interface between the user's driver code and the gradient software generated by ADIFOR. These data objects must be initialized exactly as they are if the top-level routine is to be used to calculate a value of the function. The ADIFOR generated derivative code also used three types of ADIFOR generated variables to interface with the user:

1. An integer variable (default name, `g_p_`).

2. For each variable in the interface to the top-level routine which is declared to be active by ADIFOR, a corresponding gradient object (for example, if `xhat` is an active variable, then the default name of the corresponding gradient object is `g_xhat`).

3. For each gradient object which occurs in the calling sequence of the gradient code corresponding to the top-level routine, an integer variable (for example, if the gradient object is named `g_xhat` in the ADIFOR generated derivative code, then the associated integer variable is called `ldg_xhat` there.

The type 1 integer variable, `g_p_`, must be initialized by the user to the number of linear combinations of the partial derivatives of each scalar component of the dependent variables the user wishes to calculate using some user supplied coefficients. This value must never be larger than the value given to the ADIFOR preprocessor option `AD_PMAX`. Reminder: In the user's driver code, if the user is using the ADIFOR dummy parameter name `g_p_` for the actual argument name, then the user must include an `INTEGER` declaration for the variable `g_p_`. Otherwise, the Fortran compiler will, by default, declare `g_p_` to of type `REAL`.

For every gradient object in the calling sequence of the gradient code corresponding to the top-level routine, the user must include a declaration

of that array in the user's driver code; and in that declaration, the user must choose a value for the first dimension. That value must be at least as big as the largest value of `g_p_` which the user will ever use with that gradient object. The value of that first dimension declaration must be passed to the gradient code for the top-level routine in the corresponding type 3 integer variable. Helpful hint: for gradient objects in ADIFOR generated common blocks, the first dimension has been assigned by ADIFOR and the user should just copy the ADIFOR generated declarations for these variables and common blocks into the user's driver code.

Of the type 2 variables, those belonging to independent variables are referred to in reference [2] as *seed matrices*. The user must initialize these seed matrices. The subject of initializing seed matrices is covered in sections 5.1 through 5.4 of this Technical Memorandum as well as, for example, sections 2.5 and 4 and appendix A of reference [2].

Of the remaining gradient objects, the safe recommendation is to set them all to zeros. If the parent of a gradient object is being used to input data to the top-level routine, then zeroing out its gradient object is necessary. (This assumes that the data in the parent object does not depend on any of the independent variables. If this assumption is false, then the user probably has not chosen the correct top-level routine.) If the parent object is never referenced by the top-level routine or any of its subordinate subprograms until one of them sets values in it, then there is no theoretical reason why the associated gradient object would need to be initialized; but no harm can come from zeroing it out.

6 An example

In the following example, a 2 by 2 matrix Y (contained in the Fortran array `y`) depends on the 2 by 2 matrix X (contained in the Fortran array `x`) according to the formula

$$Y = X'AX$$

where A is a 2 by 2 constant matrix (contained in Fortran array `a`) and X' is the transpose of X. This calculation is made in two steps with intermediate results stored in a 2 by 2 workspace matrix W (contained in the Fortran array `w`):

$$W = AX$$

$$Y = X'W$$

The calculation is done by `subroutine test` whose dummy argument list includes the arrays `x`, `y`, and `w`. The coefficient array `a` is communicated to `subroutine test` through common block `/dataset/`. The top-level routine is in file `test.f`:

```
subroutine test(x,y,w)
double precision a(2,2), x(2,2), y(2,2), w(2,2)
common /dataset/ a
call DGEMM( 'N', 'N', 2, 2, 2, 1.0d0, a, 2, x, 2, 0.0d0, w, 2 )
call DGEMM( 'T', 'N', 2, 2, 2, 1.0d0, x, 2, w, 2, 0.0d0, y, 2 )
return
end
```

The subroutine, `DGEMM`, called twice by `test.f`, is a matrix-matrix multiply routine from the BLAS (Basic Linear Algebra Subprograms, [6] and [7]).

The ADIFOR script file `test.adf` for this top-level routine is:

```
AD_PROG=test.cmp
AD_TOP=test
AD_IVARS=x
AD_DVARS=y
AD_PMAX=4
AD_OUTPUT_DIR=.
```

Notice that the single independent variable `x` and the single dependent variable `y` are both 2 by 2 matrices. The complete gradient $\partial Y/\partial X$ is desired, and X is made up of 4 scalar variables (`x` is made up of 4 array elements), so the ADIFOR preprocessor option `AD_PMAX` has been set to 4. The script file references the composition file `test.cmp`:

```
test.f
dgemm.f
lsame.f
xerbla.f
```

The two files `lsame.f` and `xerbla.f` contain subroutines called by subroutine `DGEMM` (which is in `dgemm.f`) and so must be available for ADIFOR to scan. As it turns out, they are not involved in the dependency of `y` on `x`, so no derivative code is generated for either of them.

After the user's computer environment has been prepared according to the instructions in [2, Section 2.1.1], the gradient code can now be generated using the command:

```
% Adifor2.1 AD_SCRIPT=test.adf
```

Following the execution of the `Adifor2.1` command, many new files (some hidden, i.e., named with an initial period) and a subdirectory (`AD_CACHE`) appear in the working directory. Of interest to the user are files related to those listed in the composition file and containing the derivative code. Two are generated, `g_test.f` and `g_dgemm.f`. The user must also take note of what is NOT generated; even though the composition file contains `lsame.f` and `xerbla.f`, there is NO `g_lsame.f` or `g_xerbla.f` generated. This indicates that ADIFOR has determined that `lsame.f` and `xerbla.f` were not involved in the dependency of `y` on `x`.

Information which the user needs to write a driver for the derivative code is contained in the top-level derivative routine, `g_test`. This is in file `g_test.f` which (minus some ADIFOR generated prefacing comments) looks like:

```
      subroutine g_test(g_p_, x, g_x, ldg_x, y, g_y, ldg_y, w, g_w, ldg_
     *w)
        double precision a(2, 2), x(2, 2), y(2, 2), w(2, 2)
        common /dataset/ a
        double precision d1, d2
        integer g_pmax_
        parameter (g_pmax_ = 4)
        integer g_p_, ldg_x, ldg_y, ldg_w
        double precision g_x(ldg_x, 2, 2), g_y(ldg_y, 2, 2), g_w(ldg_w,
     *2, 2), g_a(g_pmax_, 2, 2)
        common /g_dataset/ g_a
        integer g_ehfid
        save /g_dataset/
        external g_dgemm
        data g_ehfid /0/
C
        call ehsfid(g_ehfid, 't est','g_test.f')
C
        if (g_p_ .gt. g_pmax_) then
          print *, 'Parameter g_p_ is greater than g_pmax_'
          stop
        endif
        d1 = 1.0d0
        d2 = 0.0d0
        call g_dgemm(g_p_, 'N', 'N', 2, 2, 2, d1, a, g_a, g_pmax_, 2, x,
     * g_x, ldg_x, 2, d2, w, g_w, ldg_w, 2)
        d1 = 1.0d0
        d2 = 0.0d0
        call g_dgemm(g_p_, 'T', 'N', 2, 2, 2, d1, x, g_x, ldg_x, 2, w, g
     *_w, ldg_w, 2, d2, y, g_y, ldg_y, 2)
```

```
      return
    end
```

.

An examination of the dummy parameter list of `subroutine g_test` shows which of the dummy parameters to `subroutine test` have been promoted to active status by ADIFOR. It is seen that the independent variable `x`, the dependent variable `y`, and the workspace array `w` are all active. Scanning `subroutine g_test` for common blocks, it is discovered that ADIFOR has generated a new common block, `/g_dataset/` and placed a derivative object, `g_a`, for array `a` in it. The user will need to take this into account in writing the driver code.

While the code to be processed by ADIFOR must be in Fortran 77 and the resulting derivative code is also, the driver code could be in any language which permits of compilation into compatible binaries. The following example driver code is written in Fortran 90. In the following listing of file `driver.f90`, the leading line numbers are not part of the actual code, but have been added for purposes of subsequent discussion:

```
 1  program driver
 2  !
 3  ! The code which is processed by ADIFOR must be Fortran 77,
 4  ! but it can be compiled by a Fortran 90 compiler and driven
 5  ! by a Fortran 90 driver.
 6  !
 7  ! The following declarations have been lifted from g_test.f
 8  ! and edited
 9  !
10  double precision a(2, 2), x(2, 2), y(2, 2), w(2, 2)
11  common /dataset/ a
12  integer, parameter :: g_pmax_ = 4
13  integer  g_p_, ldg_x, ldg_y, ldg_w
14  double precision  g_x(4, 2, 2), g_y(4, 2, 2),&
15   &g_w(4, 2, 2), g_a(g_pmax_, 2, 2)
16  common /g_dataset/ g_a
17  save /g_dataset/
18  !
19  !  Although the first layer of g_y could be used to return a single
20  !  linear combination of partial derivatives for the second call to
21  !  g_test, a new array is introduced for didactical reasons.  It
22  !  need not follow the ADIFOR naming convention.
23  !  The new array is greatly overdimentioned.
24  !
25  double precision gradobj(9, 2, 2)
```

```
26  !
27  ! Now put initial data values in original code arrays
28  !
29  a = reshape((/6.0d0, 1.0d0,1.0d0,6.0d0/),(/2,2/))
30  x = reshape((/3.0d0, -1.0d0,2.0d0,-7.0d0/),(/2,2/))
31  !
32  ! Set up the seed matrix for a full gradient calculation
33  !
34  ! Layer 1 computes (partial y) / (partial x(1,1))
35  g_x(1,:,:) = reshape((/1.0d0,0.0d0,0.0d0,0.0d0/),(/2,2/))
36  ! Layer 2 computes (partial y) / (partial x(2,1))
37  g_x(2,:,:) = reshape((/0.0d0,1.0d0,0.0d0,0.0d0/),(/2,2/))
38  ! Layer 3 computes (partial y) / (partial x(1,2))
39  g_x(3,:,:) = reshape((/0.0d0,0.0d0,1.0d0,0.0d0/),(/2,2/))
40  ! Layer 4 computes (partial y) / (partial x(2,2))
41  g_x(4,:,:) = reshape((/0.0d0,0.0d0,0.0d0,1.0d0/),(/2,2/))
42  !
43  ! Zero out remaining gradient objects
44  !
45  g_y = 0.0d0
46  g_w = 0.0d0
47  g_a = 0.0d0
48  !
49  ! set integer arguments to g_test
50  !
51  g_p_ = 4
52  ldg_x = 4
53  ldg_y = 4
54  ldg_w = 4
55  call g_test(g_p_, x, g_x, ldg_x, y, g_y, ldg_y, w, g_w, ldg_w)
56  !
57  !  Now, y=x'*a*x and g_y contains dy/dx.
58  !  Code to use the Jacobian array could be put here.
59  !
60  ! Set up the seed matrix so g_test will calculate a linear
61  ! combination of the partial derivatives of y.
62  ! Put the desired coefficients in layer 1.
63  g_x(1,1,1) = 2.0d+0
64  g_x(1,2,1) = -3.5d+0
65  g_x(1,2,1) = 0.3d+0
66  g_x(1,2,2) = 1.414d+0
67  !
68  ! Zero out remaining gradient objects again
69  !
70  gradobj = 0.0d0
```

```
71  g_w = 0.0d0
72  g_a = 0.0d0
73  !
74  ! set g_p_ for this call
75  !
76  g_p_ = 1                                  ! Only using the first layer !!!
77  !
78  !  "Leading dimension" parameters may be entered into the calli ng
79  !  sequence as liter als.  Notice that these must match the values
80  !  used in the dimensioning statements and not the g_p_ value:
81  !
82  call g_test(g_p_, x, g_x, 4, y, gradobj, 9, w, g_w, 4)
83  !
84  !  Now, y=x'*a*x (again!) and the first layer of gradobj contai ns
85  !     the linear combination of the partisl derivatives of y wit h
86  !     respect to the elements of x with the coefficients entered into
87  !     the first layer of g_x.
88  !  Code to use the l inear combination of partial derivatives could
89  !     be put here.
90  !
91  end program driver
```

In `program driver`, lines 10-17 show data object specifications which have been borrowed (edited to remove unneeded variables and translated to Fortran 90) from `g_test`. The array declared in line 25 will be used as a gradient object for the dependent variable and does not follow the ADIFOR naming convention; that is no problem, the names of the variables in the driver program can be anything. Only the names of common blocks and subroutines called (i.e., global symbols) and the dimension and type specifications of objects in common blocks need to match what is given in the ADIFOR generated code.

Lines 34-41 initialize the seed array `g_x`. For example, line 37 says that the second layer of the seed array will be the 2 by 2 matrix

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}.$$

This is what is needed to select the partial derivative with respect to `x(2,1)` from the gradient of `y`.

In lines 45-47, the other gradient objects are set to zero. Line 47 is necessary since `a` is constant, but is an input value being treated as active. The other two are just for safety. In lines 51-54, the integer arguments to

`g_test` are being set to their proper values for this call, which occurs at line 55.

In the remainder of the driver, another call to `g_test` is prepared and made, this time to calculate a single linear combination of the partial derivatives of `y` with a set of coefficients. The coefficients are set in the first layer of `g_x` in lines 63-66. The array `gradobj` is used for the dependent variable gradient object. Even though it has 9 layers, only one is used. However, `g_test` is told about the 9 layers in the call at line 82.

With the ADIFOR environment still in place (so that the operating system knows about the value of environment variables like `$AD_LIB`), the driver program and necessary subroutines can be compiled, linked, and run by the commands:

```
% f90 -c driver.f90
% f90 -c g_test.f
% f90 -c g_dgemm.f
% f90 -c lsame.f
% f90 -c xerbla.f
% f90 -o runfile driver. o g_test.o g_dgemm.o lsame.o xerbla.o\
%    $AD_LIB/lib/ReqADInt rinsics-$AD_OS.o\
%    -L $AD_LIB/lib -lADI ntrinsics-$AD_OS
% runfile
```

The `f90` command indicates invocation of a Fortran 90 compiler. It is not necessary to compile and load `test.f` and `dgemm.f` since their functions are duplicated in `g_test.f` and `g_dgemm.f`. However, since `dgemm.f` called `lsame` and `xerbla` and these subroutines were not differentiated, `g_dgemm.f` also calls them and their files must be included in the compile and link.

7 Generating code to calculate Hessians

ADIFOR 2.0D can be used to generate code for the calculation of gradients. This is useful. However, there are some applications such as the one which motivated this study (section 1) which need Hessians and other second derivative information. The developers of ADIFOR have, in version 3.0, a processor which can generate Hessian information. However, at the level of support and documentation which exists as of the time of this writing, ADIFOR 3.0 is not a viable option for the general user. So, an alternative is explored.

ADIFOR 2.0D accepts as input Fortran 77 code which calculates a function and produces as output Fortran 77 code which calculates the linear

combinations of the partial derivatives of that function. Suppose that one needs Fortran 77 code which calculates the Hessian (or other second derivative related information) of that function. The obvious thought is to apply ADIFOR 2.0D again, this time to the Fortran code which it produced on the first application. This is what was done in the study mentioned in section 1.

The second application of ADIFOR presents some problems which are usually not present in the first application. As commented in section 2, ADIFOR generates code which calculates linear combinations of the partial derivatives of the dependent variables with respect to the independent variables. It follows that if one applies ADIFOR to ADIFOR generated code naming a derivative object of the first dependent variable as the new dependent variable, the resulting code will calculate linear combinations of the partial derivatives with respect to the second set of independent variables of linear combinations of the partial derivatives of the dependent variables with respect to the first set of independent variables. The complexity of this description of second derivative code capability is reflected in the complexity of the code, itself. While comments will be made in section 7.3 illustrating the full generality of the second derivative code in a simple case (scalar dependent variable, independent variables in a single Fortran vector variable), the main burden of the present comments will be directed at explaining how to set the seed matrices to generate just the individual second partial derivatives and where to harvest them. This is the information which was found sufficient to generate and use code to calculate the second derivative information needed in the study mentioned in section 1.

Attention will be given to the following items:

1. To avoid duplicating names of variables, etc., the naming character pair used in the second application of ADIFOR must be changed from that used in the first. For example, if default naming characters are used in the first application, then one might use the ADIFOR preprocessing option `AD_PREFIX=h` in the second.

2. The user must be aware of, and make accommodations for, one idiosyncrasy of the default way in which ADIFOR 2.0D deals with exceptions, i.e., points of non-differentiability.

3. The user must choose independent and dependent variables for both ADIFOR runs.

4. Derivative objects, including seed matrices, are generated by both ADIFOR runs, and the user must deal with questions of initialization of and interpretation of final answers in these derivative objects.

Nothing more need be said about duplicate name avoidance. The remaining points are addressed in sections 7.1 – 7.3.

7.1 Consequences of ADIFOR exception handling

This Technical Memorandum has heretofore ignored the topic of ADIFOR exception handling (e.g., how ADIFOR generated code warns the user if asked to provide the derivative of `ABS(X)` at `X = 0.0D+00`). For that, the user is referred to [2], especially Appendix B. However, there is one consequence of ADIFOR treatment of exceptions which has an impact on iterated applications of ADIFOR.

Under default preprocessing options, ADIFOR generates a variable whose name, independent of the setting of preprocessor option `AD_PREFIX`, is `g_ehfid`. By default, ADIFOR generated code includes lines of the form:

```
      integer g_ehfid
      data g_ehfid /0/
C
      call ehsfid(g_ehfid, 'g_func','h_g_func.f')
```

The call to `ehsfid` provides information to the ADIFOR exception handling subroutines. The character strings `'g_func'` and `'h_g_func.f'` which appear in the call to `ehsfid` arise because this code fragment came from a file named `h_g_func.f` which was generated by applying ADIFOR 2.0D to file `g_func.f` with preprocessing options including `AD_TOP=g_func` and `AD_PREFIX=h`. The problem arises since, despite the specification of `AD_PREFIX=h`, the variable name `g_ehfid` is being used. If the code for `g_func` was generated by ADIFOR from the code for `func` using default settings of ADIFOR preprocessing options, a similar code fragment, specifically including the declaration and initialization of the variable `g_ehfid`, would have been generated in `g_func`. When ADIFOR 2.0D generates file `h_g_func.f` from `g_func.f`, it both copies the existing declaration and initialization of `g_ehfid` and generates a new one. Fortran compilers treat this duplicate declaration (and initialization) of the same variable as an error.

One solution for this problem is to suppress the generation of this exception handling information in one or both of the ADIFOR runs. This can be accomplished by including the ADIFOR preprocessor option

`AD_EXCEPTION_FLAVOR=performance`,

see ([2, p. 40]), in one or both of the ADIFOR runs. The choice used in the study mentioned in section 1 was to include this option in the first derivative generation run. This has the advantage that the code generated in this fashion compiles without error. A disadvantage is that some attempt to evaluate the derivative of a function at a point of non-differentiability may go unreported to the user.

7.2 Choice of dependent and independent variables

In naming dependent variables for iterated applications of ADIFOR, the user should remember that, in addition to ADIFOR generated code to calculate derivative-related information for the dependent variable(s), the ADIFOR generated subprograms contain a complete copy of all the calculations of the original code. Thus, if a subroutine named `func` is processed by ADIFOR with variable `f` named as a dependent variable and default preprocessor options, then a subroutine named `g_func` is generated whose calculations include both all of the calculations done for the dependent variable `f` in the original subroutine `func` and the additional calculations necessary to generate the proper values for inclusion in the derivative object `g_f`. If `g_func` is then itself subjected to ADIFOR processing with `AD_PREFIX=h` in order to generate code which will calculate second derivative information for the dependent variable `f`, `g_f` must be specified as a dependent variable in this second ADIFOR run. This second derivative information will then be placed in the gradient object `h_g_f`. It is probably unnecessary to specify `f` as a dependent variable in this second ADIFOR run unless the user is playing some very esoteric games with the dependent variable lists or the seed matrices. The subroutine `h_g_func` will contain all of the calculations from `g_func` related to `g_f`. If values are chosen for the seed matrices as will be explained in section 7.3.2 in order to calculate Hessian information for `f` in the gradient object `h_g_f`, these same values will result in the calculation of gradient values for `f` in the gradient object `g_f`. If `f` had also been specified as a dependent variable in the second ADIFOR run, the gradient object `h_f` would have been generated, and, if the seed matrices `g_x` and `h_x` are set

to calculate the Hessian of `f` with respect `x`, executing `h_g_f` would have made two independent calculations of the gradient of `f`, placing copies of the gradient in both `g_f` and `h_f`.

That said, one situation comes to mind where one might wish to declare `f` to be a dependent variable in both the first and second applications of ADIFOR. If the desired second derivative information includes only mixed second partial derivatives from two disjoint sets of independent variables, and if the gradients with respect to all variables in both sets is desired, then one would want to include `f` as a dependent variable in both applications of ADIFOR. Then the gradient object `g_f` would contain first derivatives of `f` with respect to the first set of independent variables and the gradient object `h_f` would contain first derivatives of `f` with respect to the second set of independent variables.

7.3 Information content of derivative objects

For purposes of illustration in this section, it is supposed that a subroutine named `func` is the top-level subroutine in a Fortran 77 software package which computes the value returned in a scalar or array variable `f` (and possibly other variables) as a function of the values passed into `func` in a scalar or array variable `x` (and possibly other variables). It is desired to compute Hessian information consisting (at least in part) of the (double and mixed) second partial derivatives of (the components of) `f` with respect to (the components of) `x`. This is to be done with two applications of ADIFOR.

The first application of ADIFOR will be to `func.f`, the file containing the subroutine `func`, (and possibly to other files) using default naming characters so that the file `g_func.f` (and possibly others) is generated containing a subroutine named `g_func`. This application of ADIFOR will name `f` as a dependent variable and `x` as an independent variable. This application will include the ADIFOR preprocessing option which prevents generation of the variable name `g_ehfid`:

```
AD_EXCEPTION_FLAVOR=performance
```

The input needed by subroutine `g_func` will include the input variable `x` to the parent subroutine `func` and the seed matrix `g_x`, and the output produced by subroutine `g_func` will include the the same output `f` produced by the parent subroutine `func` and the derivative object `g_f`. With proper initialization of `g_x` (and the other seed matrices, if any), execution of `g_func`

will set information in g_f which includes the value of the gradient of f with respect to x at the given input values.

The second application of ADIFOR will be to g_func.f, naming g_func as the top level routine. This application will include the preprocessing option AD_PREFIX=h and name g_f as a dependent variable and, again, x as an independent variable. A file named h_g_func.f will be generated which will include the code for ADIFOR generated subroutine h_g_func. The inputs to subroutine h_g_func will include all of the inputs to its parent subroutine g_func including both the independent variable x and the previously generated seed matrix g_x and also a newly generated seed matrix h_x. The output of h_g_func will include all of the output of its parent subroutine g_func including the original dependent variable f, and the first derivative information which is determined by the contents of the seed matrix g_x (and, perhaps, other g_ seed matrices) and placed in the derivative object g_f. The output of h_g_func will also include second derivative information in the derivative object h_g_f. This will depend on the contents of the seed matrices g_x and h_x (and possibly other g_ and h_ seed matrices).

In the terminology of section 5.2, a derivative object is made up of layers which are the same shape as the parent object. Each layer of the seed matrices (derivative objects for the independent variables) provides coefficients used in forming linear combinations of the gradients of each dependent variable. These are placed in the corresponding layer of the derivative object of the dependent variable. The first subscript position of the derivative objects is used to index through the layers. This relates derivative objects to their parents. The next topic to be addressed is the relationship between second derivative objects (like h_g_f) and their grandparents (for this example, f).

For purposes of illustration, suppose that f has rank 1, so an array element of f can be references using a single subscript, as in f(k). Then its (first) derivative object g_f is of rank 2, and g_f(j,k) references linear combinations of the partial derivatives of f(k) with respect to the independent variables declared in the first application of ADIFOR using coefficients stored in layer j of the g_ seed matrices. The second application of ADIFOR adds another dimension of layers to the derivative object g_f when it forms the (second) derivative object h_g_f. A typical element here might be referenced as h_g_f(i,j,k). It is a linear combination of the partial derivatives of g_f(j,k) with respect to the independent variables named in the second application of ADIFOR with coefficients stored in layer i of the h_ seed matrices. Relating this back to the original function, this means that

`h_g_f(i,j,k)` contains a sum of terms of the form

$$a_{ix} b_{jy} \frac{\partial^2 f_k}{\partial x \partial y}$$

where `x` is a variable from the second set of independent variables, `y` is a variable from the first set of independent variables, a_{ix} is from the `x` position of layer `i` of the `h_` seed matrix containing `x`, and b_{jy} is from the `y` position of layer `j` of the `g_` seed matrix containing `y`. This will be illustrated in more detail in two specific examples.

7.3.1 Scalar f, vector x

Suppose that `f` is a scalar and `x` is a vector. Suppose also that the first application of ADIFOR names only `f` as dependent variable and only `x` as independent variable, and that the second application of ADIFOR names only `g_f` as dependent variable and only `x` as independent variable. Suppose that `AD_PMAX` is specified to be 1 in both runs. Then `h_g_f` is (effectively) a scalar, and `g_x` and `h_x` are (effectively) vectors of the same order as `x`. If `g_x` and `h_x` are initialized with coefficients from the column vectors s_g and s_h, respectively, then execution of the second derivative code will result in calculation of the number

$$(s_h)' \left[\frac{\partial^2 f}{\partial x^2} \right] (s_g).$$

7.3.2 Second derivatives of a function of several Fortran vector variables

This section will illustrate the generation of second derivative information for a function of several variables which are all contained in Fortran scalar or vector data objects. If the technique of this section is applied to generating second derivative code by two applications of ADIFOR using the same `AD_IVARS` setting for both, then a Hessian would be calculated. However, it is not necessary to restrict `AD_IVARS` to be the same in both ADIFOR applications, and a more general case will be illustrated.

This illustration will continue the illustration presented in section 5.4. In addition to the independent variables `w`, `x`, `y`, and `z` introduced there, suppose that the dependent variable has 24 scalar components contained in a Fortran array variable `f` dimensioned (6,4). For simplicity, it will be assumed that

the computation of `f` is complete in the single subroutine `func`, and that the interface to subroutine `func` is:

```
subroutine  func(w,x,y,z, f)
```

All four variables, `w`, `x`, `y`, and `z`, with their total of 7 scalar components will be declared as independent variables in the first ADIFOR differentiation, while only the three variables, `x`, `y`, and `z`, with their total of 5 scalar components will be declared as independent variables in the second ADIFOR differentiation.

For the first differentiation, a composition file, `first.cmp`, is used which contains a single line:

```
func.f
```

The script of ADIFOR preprocessing options is contained in file `first.adf` which includes the lines:

```
AD_PROG=first.cmp
AD_TOP=func
AD_IVARS=w,x,y,z
AD_DVARS=f
AD_PMAX=7
AD_EXCEPTION_FLAVOR=perf ormance
```

This will generate the file `g_func.f` containing the subroutine `g_func`. The `SUBROUTINE` statement for subroutine `g_func` is:

```
      subroutine  g_func{g_p_,  w,  g_w,  ldg_w,  x,  g_x,  ldg_x,  y,  g_y,
     *ldg_y,  z,  g_z,  ldg_z,  f,  g_f,  ldg_f)
```

For the second differentiation, a composition file, `second.cmp`, is used which contains a single line:

```
g_func.f
```

The script of ADIFOR preprocessing options is contained in file `second.adf` which includes the lines:

```
AD_PROG=second.cmp
AD_TOP=g_func
AD_IVARS=x,y,z
AD_DVARS=f
AD_PMAX=5
AD_PREFIX=h
```

This will generate the file `h_g_func.f` containing the subroutine `h_g_func`. If ADIFOR does not find it necessary to declare `f` or any of the seed matrices in subroutine `g_func` as being active, the `SUBROUTINE` statement for subroutine `h_g_func` is:

```
subroutine h_g_func(h_p_, g_p_, w, g_w, ldg_w, x, h_x, ldh_x, g_x,
*ldg_x, y, h_y, ldh_y, g_y, ldg_y, z, h_z, ldh_z, g_z, ldg_z, f,
*g_f, h_g_f, ldh_g_f, ldg_f)
```

The user is now faced with the task of writing code which will call subroutine `h_g_func` with the correct information to cause calculation of the desired value. First, clarification is given of what the desired output of `h_g_func` is.

The mathematical vector `s` is used to represent the 7 scalar values `w(1)`, `w(2)`, `x`, `y`, `z(1)`, `z(2)`, and `z(3)` in the 4 Fortran variables `w`, `x`, `y`, and `z`; and `t` is used to represent the 5 scalar values in the 3 Fortran variables `x`, `y`, and `z`. The 7 index positions of `s` correspond to the 7 layers of the `g_` derivative objects and the 5 index positions of `t` correspond to the 5 layers of the `h_` derivative objects. If $\mathtt{f}_{ij}$ represents the scalar function whose value is returned by subroutine `func` in the `(i,j)` component of Fortran array `f`, then it is desired that a call to `h_g_func` return the gradient

$$\frac{\partial \mathtt{f}_{ij}}{\partial \mathtt{s}}$$

in the array section `g_f(:,i,j)` and the transposed second derivative matrix

$$\left(\frac{\partial^2 \mathtt{f}_{ij}}{\partial \mathtt{t} \partial \mathtt{s}}\right)'$$

in the array section `h_g_f(:,:,i,j)`. (The colon, `:`, used here as an array subscript, is a special case of the subscript triplet notation of Fortran 90 as in [5, p. 170]. It indicates that the indicated subarray uses the entire declared range of the dimension in which the colon occurs. In this, it is identical to the use of the colon as an array subscript in Matlab©.)

This can be accomplished by passing sections of an order 7 identity matrix for the first derivative (`g_`) seed matrices and sections of an order 5 identity matrix for second derivative (`h_`) seed matrices as in section 5.4. The `INTEGER` variables `h_p_` and `g_p_` are passed as 5 and 7, respectively. The output objects are declared:

```
double precision f(6,4), g_f(7,6,4), h_g_f(5,7,6,4)
```

All the `ldg_` parameters are entered as 7's and all the `ldh_` parameters are entered as 5's. The Fortran variables `w`, `x`, `y`, and `z` are initialized to the values at which the function and its first and second derivatives are to be calculated. Then subroutine `h_g_func` can be called, and the desired results are returned in `f`, `g_f`, and `h_g_f`.

The user is again cautioned that ADIFOR may create derivative objects for data objects which the user believes to be inactive. The user must be alert for any cases where this occurs in the calling sequence of the (first or second) derivative of a top level routine or in a common block. The user must declare these extra derivative objects (with their common blocks, where appropriate) in the driver code and should initialize them to zeros.

8 Summary

ADIFOR 2.0D is a Fortran code processor. It converts Fortran 77 code which calculates a (possibly vector-valued) function f of a vector argument x into Fortran code which calculates information related to the sensitivity of f to variation in x (including, but not limited to, $\partial f / \partial x$). The principal reference for ADIFOR 2.0D is [2].

This note gives one user's experience with using the resulting ADIFOR generated code. Emphasis is given to items the user needs to know to write a driver code which will call the ADIFOR generated derivative software. Explanations are given on how to deal with ADIFOR generated common blocks. The dummy parameter list of ADIFOR generated subroutines is explained.

The ADIFOR generated code contains data structures called derivative objects, some of which are seed matrices which the user must initialize and some of which contain gradients and other sensitivity information which the ADIFOR generated code calculates for the user. This note has explained how to initialize seed matrices and, based on what is in the seed matrices, how to interpret the answers returned by the ADIFOR generated code.

By taking extra precautions, it is possible to apply ADIFOR to code which, itself, has been generated by ADIFOR. The result is code which can calculate second derivative information for the original function f. This process has been explained and some illustrations given.

Appendix: FTNCHEK as a call tree generator

FTNCHEK is a Fortran 77 program checker. It is designed to detect certain errors in a Fortran program that a compiler usually does not. For purposes of this note, the one of its many capabilities which is emphasized is its ability to scan a collection of files containing Fortran 77 source code and produce a call tree. From this, it can be determined what subprograms are required by a specific subroutine.

FTNCHEK was designed by, and is maintained by, Dr. Robert Moniot, professor at Fordham University. It is freely available from Netlib by visiting the URL `ftp://netlib.org/fortran` and downloading file `ftnchek.tgz`. More information may be found at Dr. Moniot's web site:

`http://www.dsm.fordham.edu/~ftnchek`

By default, `ftnchek` produces voluminous output which, while of value in detecting programming errors in the code, does not provide the calling tree information in a compact format. To get the calling tree for all Fortran 77 subprograms contained in `*.f` files in the current working directory, execute the command:

```
ftnchek -nocheck -calltree=tree,no-prune *.f > call_tree
```

The desired information can then be found in the file `call_tree`.

NASA Langley Research Center
Hampton, VA 23681-2199
March 21, 2002

References

[1] Bryson, Jr., Arthur E.; and Ho, Yu-Chi: *Applied Optimal Control, Revised Printing*. Hemisphere Publishing Corporation, Taylor & Francis, 1900 Frost Road, Suite 101, Bristol, PA 19007, 1975.

[2] Bischov, Christian; Carle, Alan; Hovland, Paul; Khademi, Peyvand; and Mauer, Andrew: *ADIFOR 2.0 Users' Guide (Revision D)*. Tech. Rep. CRPC-95516-S, Center for Research on Parallel Computation, Rice University, CRPC – MS 41, 6100 Main

Street, Houston, Texas 77005-1892, June 1998, Available from URL = `http://www.cs.rice.edu/~adifor/AdiforDocs.htm`.

[3] American National Standards Committee on Computers and Information Processing, X3: *American National Standard Programming Language FORTRAN*. American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018, 1978.

[4] Anderson, E.; Bai, Z.; Bischof, C.; Blackford, S.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenny, A.; and Sorensen, D.: *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, third ed., 1999, Errata available at URL = `http://www.netlib.org/lapack/html/errata.lug`.

[5] Adams, Jeanne C.; Brainerd, Walter S.; Martin, Jeanne T.; Smith, Brian T.; and Wagener, Jerrold L.: *Fortran 95 Handbook*. The MIT Press, Cambridge, MA, 1997.

[6] Dongarra, J. J.; Du Croz, J.; Duff, I. S.; and Hammarling, S.: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, vol. 16, 1990, pp. 1–17.

[7] Dongarra, J. J.; Du Croz, J.; Duff, I. S.; and Hammarling, S.: Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, vol. 16, 1990, pp. 18–28.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 2002	Technical Memorandum

4. TITLE AND SUBTITLE
Some User's Insights Into ADIFOR 2.0D

5. FUNDING NUMBERS
728-30-30-01

6. AUTHOR(S)
Daniel P. Giesy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681–2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L–18184

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA/TM–2002–211738

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified-Unlimited
Subject Category 61 Distribution: Nonstandard
Availability: NASA CASI (301) 621–0390

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Some insights are given which were gained by one user through experience with the use of the ADIFOR 2.0D software for automatic differentiation of Fortran code. These insights are generally in the area of the user interface with the generated derivative code – particularly the actual form of the interface and the use of derivative objects, including "seed" matrices. Some remarks are given as to how to iterate application of ADIFOR in order to generate second derivative code.

14. SUBJECT TERMS
Automatic Differentiation, Fortran, ADIFOR

15. NUMBER OF PAGES
49

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	