MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0350970 9

ornl

ORNL/TM-12050

OAK RIDGE NATIONAL LABORATORY

MARTIN MARIETTA

Reverse Automatic Differentiation of Modular FORTRAN Programs

J. E. Horwedel

OAK RIDGE NATIONAL LABORATORY
CENTRAL RESEARCH LIBRARY
CIRCULATION SECTION
4500N ROOM 175
LIBRARY LOAN COPY
DO NOT TRANSFER TO ANOTHER PERSON
If you wish someone else to see this report, send in name with report and the library will arrange a loan.

MANAGED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-12050

REVERSE AUTOMATIC DIFFERENTIATION OF MODULAR FORTRAN PROGRAMS

J. E. Horwedel

Computing and Telecommunications Division
at Oak Ridge National Laboratory
P.O. Box 2008
Oak Ridge, TN 37831-6370

Date Published: March 1992

Prepared by the
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831
managed by
MARTIN MARIETTA ENERGY SYSTEMS, INC.
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-84OR21400

MARTIN MARIETTA ENERGY SYSTEMS LIBRARIES

3 4456 0350970 9

CONTENTS

Page

LIST OF FIGURES iv
LIST OF TABLES v
ACKNOWLEDGMENTS vii
ABSTRACT ix

1. INTRODUCTION 1

2. GRESS 2
    2.1. OVERVIEW OF GRESS 2
    2.2. STATEMENT ADJOINTING 4

3. DESCRIPTION OF THE MODULAR DIFFERENTIATION TECHNIQUE (MDT) 6

4. GENSUB STRUCTURE AND DESIGN 9
    4.1. LINKING FLOATING-POINT ASSIGNMENT STATEMENTS 10
    4.2. LINKING MODULES 12

5. APPLICATION OF MDT USING THE GRESS GENSUB OPTION 13

6. CONCLUSIONS AND RECOMMENDATIONS 15

7. REFERENCES 16

APPENDIX A. GENSUB SAMPLE PROBLEM 19

APPENDIX B. NEW GRESS RUN-TIME LIBRARY ROUTINES 23

LIST OF FIGURES

Page

1. Processing steps for a GRESS application 3
2. Creating and solving a statement adjoint matrix 4
3. Dependency graph for DIST 7
4. Computational graph for a three-module program 7
5. Computational graph for program with common blocks 9
6. Structure of a statement frame 11
7. Statement frames form a linked-list structure 11
8. Structure of module frames with input variable row numbers only 12
9. Memory required to store derivatives using MDT vs reverse mode 14

LIST OF TABLES

Page

B.1. New GRESS run-time library routines for the GENSUB option 23

ACKNOWLEDGMENTS

The author would like to thank B. A. Worley of the Engineering Physics and Mathematics Division for his continued guidance in this work. The author would also like to acknowledge R. Q. Wright of the Computing and Telecommunications Division for his helpful technical discussions. Thanks are also due to Steven M. Robbins for his assistance in developing the GENSUB option.

ABSTRACT

Several software systems are available for implementing automatic differentiation of computer programs. The forward mode of automatic differentiation is limited by computational intensity and computer memory. The reverse mode, or adjoint approach, is limited by computer memory and disk storage. A modular technique for derivative computation that can significantly reduce memory required to compute derivatives in a complex FORTRAN model using the reverse mode of automatic differentiation is discussed and demonstrated.

1. INTRODUCTION

The calculation of derivatives necessary for sensitivity analysis, or for the optimal solution of systems of nonlinear equations, continues to be an important research objective. Several software systems have been developed or proposed for implementing automatic differentiation of computer programs. The forward mode of automatic differentiation is efficient for calculating derivatives for a large number of dependent variables with respect to a few independent variables. As the number of independent variables increases, the computational complexity, as measured in execution time and memory requirements, renders the forward mode impractical. The reverse mode, or adjoint approach, is efficient for derivatives of a few dependent variables with respect to thousands of independent variables; however, available memory and disk storage generally limit the application of the reverse mode to problems with less than a few million floating-point assignments. The fundamental problem with the reverse mode of automatic differentiation is that the accumulation of derivatives for every floating-point assignment is required. A code that uses 3 min of execution time to perform 50 million floating-point assignments could easily need more than 1 gigabyte (GB) to store the accumulated derivatives.[1-2]

GRESS (the GRadient Enhanced Software System) was designed to apply automatic differentiation to large-scale FORTRAN programs in the nuclear industry without requiring significant changes to the coding.[3-4] GRESS provides two methods of calculating and reporting derivatives. The CHAIN option implements the forward mode of automatic differentiation to calculate the derivatives of a variable with respect to a user-selected subset of the input data. The ADGEN option incorporates the reverse mode or adjoint sensitivity analysis methods to calculate derivatives of selected variables with respect to thousands of input parameters. When the ADGEN option is chosen, partial derivatives for every arithmetic assignment statement in the model are stored in memory or output to a data set. Matrix-solving routines are then used to calculate and report derivatives and sensitivities for selected results.

In this paper a modular differentiation technique (MDT) is presented that uses both forward and reverse modes to restrict the growth of execution time and storage requirements, thus extending the size of problems to which the reverse mode of automatic differentiation can be applied. MDT is implemented using GRESS and provides a compromise between the forward mode with its computational limitations and the reverse mode with its excessive memory or storage requirements.

The effectiveness of the MDT in propagating derivatives through a computer program rests on the degree of modularity in the program. Most existing large-scale FORTRAN programs do not have the degree of modularity necessary to apply MDT in an automated fashion. The approach described in this paper is to provide the basic tools to allow one to

implement MDT on a module-by-module basis in an existing code or in the development phase for a new code.

A new GRESS option, GENSUB, that permits the processing of individual program modules (i.e., a do loop, subroutine, function, or a sequence of subroutines) for calculating derivatives is used to demonstrate MDT. GENSUB is designed to allow derivatives to be propagated in either forward or reverse mode. The decision on whether to use forward or reverse mode can be made by the user or by the software.

In Sect. 2 GRESS is described. In Sect. 3 MDT is presented. The GRESS GENSUB option is described in Sect. 4. The application of MDT using the GENSUB option is demonstrated in Sect. 5.

2. GRESS

GRESS uses a method referred to as statement adjointing to process the arithmetic assignment statements in a FORTRAN program. The first part of this section provides an overview of the GRESS system. The remainder of the section describes statement adjointing as implemented in GRESS.

2.1. OVERVIEW OF GRESS

In a FORTRAN program, calculated variables are mathematical functions of previously defined variables and data. GRESS uses a precompiler to interpret FORTRAN statements and determine the mathematical operations embodied in them. As each arithmetic assignment statement in a program is interpreted, information necessary to allow the calculation of derivatives is generated. The result of the precompilation step is a new FORTRAN program that can produce derivatives for any REAL (i.e., single- or double-precision) variable calculated by the model. Consequently, GRESS enhances FORTRAN programs by adding the calculation of derivatives along with the original output. GRESS accepts a majority of ANSI-X3.9 FORTRAN 77, including subroutines, common blocks, data statements, read statements, user functions, intrinsic functions, block data subprograms, single-precision variables, double-precision variables, and equivalence statements. GRESS does not process COMPLEX variable types or statement functions. Specific limitations are discussed in ref. 3. GRESS is available from the Radiation Shielding Information Center at Oak Ridge National Laboratory and is operational on VAX/VMS computer systems. The author has implemented test versions of GRESS on VAX/ULTRIX, CRAY/UNICOS, IBM RISC/6000 Workstations, and Sun Workstations; however, GRESS has only been rigorously tested in the VAX/VMS environment.

The steps used to process a code with GRESS are illustrated in Fig. 1. A FORTRAN model is input to the GRESS precompiler to create an enhanced program. The enhanced model is compiled in the usual manner and then linked with a library of GRESS utility routines. When the enhanced model is executed, derivatives are calculated for each arithmetic assignment statement immediately before the statement is executed.

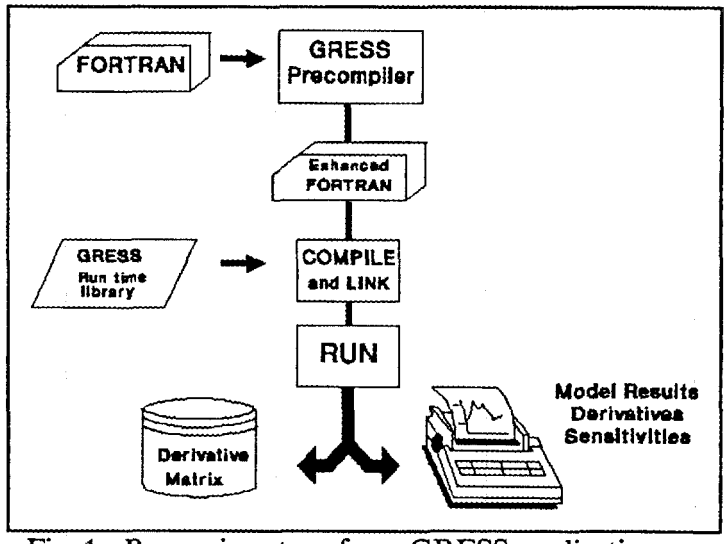

Fig. 1. Processing steps for a GRESS application.

Derivatives from a GRESS-enhanced model can be used internally (e.g., for iteration acceleration) or externally (e.g., for sensitivity studies). In this paper, we focus on the calculation of derivatives of output variables with respect to input parameters.

GRESS provides two methods for calculating derivatives. The CHAIN option calculates the derivatives of a variable with respect to a user-selected subset of the input data by repeated application of the chain rule in the forward mode. The CHAIN option calculates derivatives as the model is executing and is the recommended option when the user is only concerned with a very small number of input parameters. The ADGEN option incorporates the adjoint methods long used by nuclear engineers to calculate the derivatives of selected model responses with respect to thousands of input parameters.[5-8] When the ADGEN option is chosen, partial derivatives for every floating-point assignment statement in the model are output to a data set or stored in memory. Matrix-solving routines are then used to calculate and report derivatives for selected results. The ADGEN option provides the user with the capability to calculate and report the derivatives of any calculated model result with respect to all data input to the model. An important advantage of the adjoint method over the chain rule method is that the derivatives of selected model results can be calculated with respect to thousands of input parameters at a cost comparable to that of executing only a few model runs. To approximate the same information by direct parameter perturbations would require separate model runs for each input parameter.

2.2. STATEMENT ADJOINTING

An arithmetic assignment statement has one dependent variable (the term on the left of the equal sign) and one or more independent variables (the terms on the right); therefore, it is most efficient to calculate derivatives for an arithmetic assignment statement using the reverse mode or adjoint method. This method will be referred to as statement adjointing.

Statement adjointing is seen as an improvement over other methods of automatic differentiation because it reduces memory requirements and computational intensity during the execution of the enhanced program. Many tools for automatic differentiation require storage of 16 to 20 bytes per floating-point operation.[9] Statement adjointing reduces that storage to 16 to 20 bytes per floating-point assignment. Not only does this reduce storage, but it also reduces the number of calculations required to calculate derivatives during the execution of the enhanced code.

To implement statement adjointing, the GRESS precompiler creates and then solves an adjoint matrix for each assignment statement as it is processed (Fig. 2). Once the adjoint matrix for a statement is solved for the derivatives of the term on the left with respect to the variables on the right, the FORTRAN statements necessary to calculate those derivatives during execution are generated.

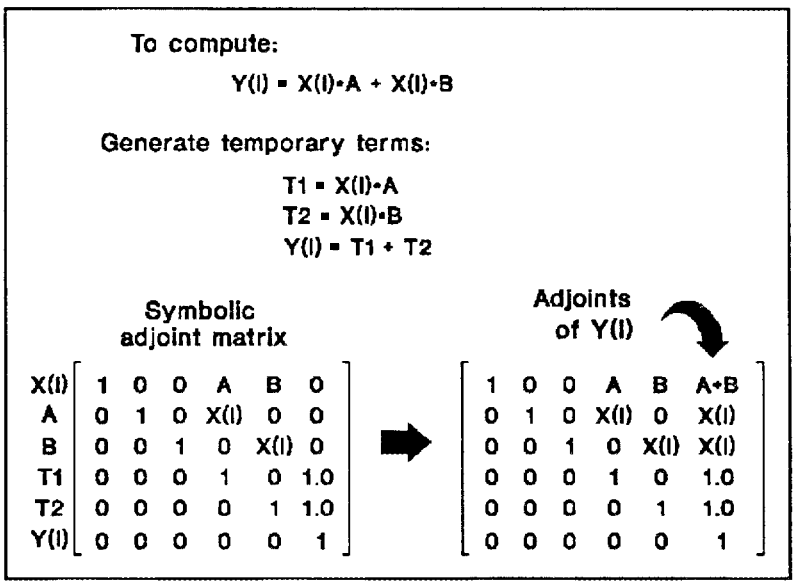

Fig. 2. Creating and solving a statement adjoint matrix.

During the precompilation step GRESS makes a single pass through a FORTRAN program. Statements defining REAL variables are parsed (1) to determine mathematical operations and (2) to solve the adjoint matrix for the statement. GRESS generates FORTRAN statements that compute the partial derivatives of the term on the left with respect to the REAL variables on the right. The original statement is output, followed by a subroutine call for processing the partial derivatives. The following sequence of FORTRAN is used to demonstrate precompilation.

```
        .
        .
        .
      DO 10 I=1,4
      Y(I)=X(I)*A + X(I)*B
10    CONTINUE
        .
        .
        .
```

Though the program generated by the precompiler appears more complicated, the partial derivatives that GRESS stores in the DX array are easy to find and verify.

```
        .
        .
        .
      DO 90002 I=1,4
      DX(1)=A+B
      DX(2)=X(I)
      DX(3)=X(I)
      Y(I)=X(I)*A + X(I)*B
      CALL LOCNXX(1,4,Y(I),X(I),A,B)
90002 CONTINUE
        .
        .
        .
```

The partial derivatives are initially stored in the DX array. Subroutine LOCNXX is a GRESS routine generated when the adjoint option is selected. When the adjoint option is selected,the partial derivatives are moved into a buffer for later processing. The interested reader is referred to the GRESS User's Manual for more information on the adjoint and CHAIN options. The GENSUB option will be discussed later in this text.

3. DESCRIPTION OF THE MODULAR DIFFERENTIATION TECHNIQUE (MDT)

A major limitation when considering the ADGEN option is problem size as measured by execution time. Since the partial derivatives for every real assignment statement are accumulated, the amount of data storage can be prohibitive. A program that performs 50 million assignments may require as much as 1 GB of storage.[1-2]

GRESS was designed to work with existing programs. GRESS treats a program as a single unit or module. Previously, GRESS could not be used to solve for derivatives from a subroutine or function independent of the rest of the program. GRESS was also limited in that it could not apply the reverse mode to solve for derivatives after each pass in an iterative type code. Solving for derivatives between iterations is of utmost importance in many applications and is a means to reduce memory required to store accumulated derivatives. MDT and the GENSUB option were developed to solve some of these problems.

To implement MDT it is necessary for a program to be thought of as a sequence of modules that are linked. Initially we will consider the link to a module to be the argument lists and return values. Later we will discuss the processing of variables provided to a module via common blocks.

A module can be considered to be any sequence of FORTRAN statements. Any module can be represented by a computational graph. As an example, consider the following formula for DIST:

$$DIST = \sqrt{\left(\sum_{1}^{4}\left(y_i^2\right)\right)^3} \; . \tag{1}$$

A computational graph for this equation is shown in Fig. 3. The squares in the computational graph represent arithmetic assignment statements. The reverse mode of automatic differentiation requires the accumulation of derivatives for every floating-point assignment that is dependent on a declared parameter.

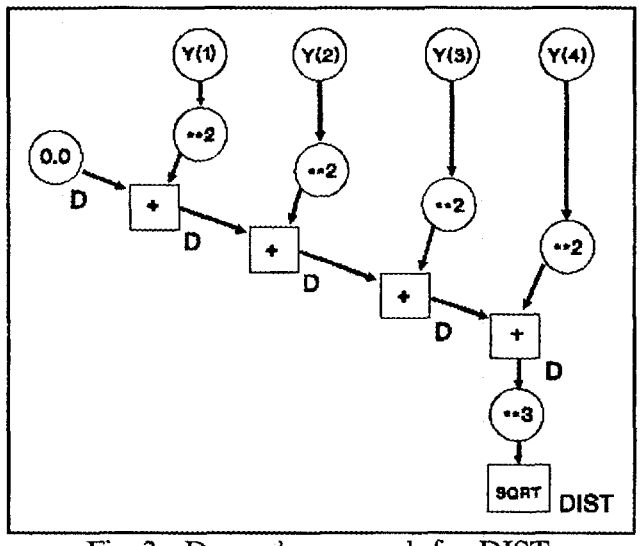

Fig. 3. Dependency graph for DIST.

In modular form, the DIST formula could be coded as a FORTRAN subroutine or function with the Y array as input and DIST as the calculated result.

A computer program can be represented as a sequence of modules, each with its own computational graph. Each module is assumed to have input and output. For simplicity we are ignoring global variables and common blocks. For the purposes of this section we are requiring input and output to be on argument lists to the module.

Figure 4 shows a computational graph for a computer program with three modules, each with its own computational structure.

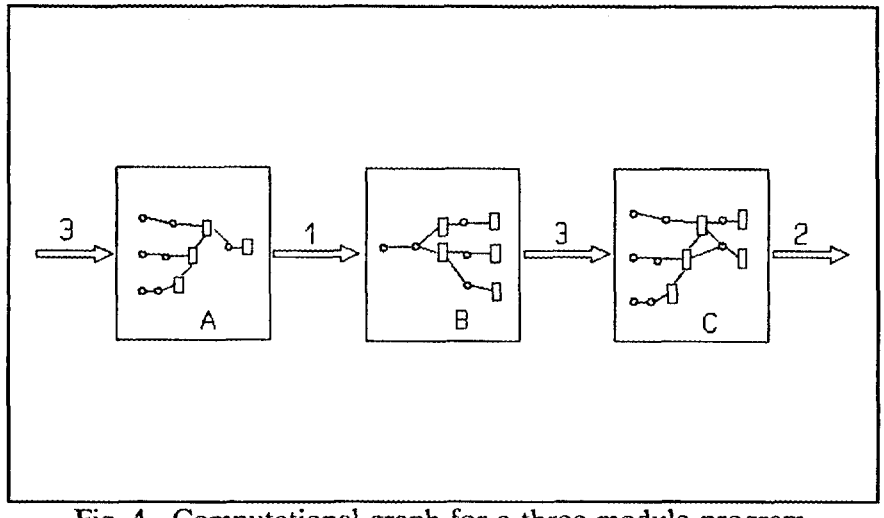

Fig. 4. Computational graph for a three-module program.

The digits on the links indicate the number of input and output variables for each module. Module A has three inputs and one output, B has one input and three outputs, and C has three inputs and two outputs.

With most reverse-mode implementations, modules A, B, and C would be run sequentially with all derivatives from all floating-point operations stored in memory or on disk. GRESS is somewhat improved in that only derivatives from floating-point assignments are stored; however, for large-scale problems the amount of storage required renders the reverse mode impractical.

MDT is designed to work with each module independently. Once a module is completed, then either forward or reverse mode is used to calculate the derivatives of the output from the module with respect to the input. Only the derivatives of the output with respect to the input need to be stored.

In most practical cases, a simple test can be used to determine whether to use forward or reverse mode to calculate the derivatives for a module. If the number of input parameters is greater than the number of output variables, reverse mode of automatic differentiation should be used. If the number of output variables is greater, forward mode should be used. The decision on whether to use forward or reverse mode does not have to be made a priori, it can be determined when the module is finished. For the example in Fig. 4, reverse mode would be applied to module A, forward mode to module B, and reverse mode to module C.

Most large FORTRAN programs are modular in design; however, common blocks provide a mechanism by which modules can share global variables that are not provided on the link to the module. It seems to be common practice to have thousands of common block variables available to every subroutine in the program. When processing a module we only have to be concerned with global variables that are accessed or stored during the execution of a module. Though a single module may have thousands of common block variables, only a subset may actually be used as dependent or independent variables. Variables that are used can be determined during execution of the module. Figure 5 shows the computational graph for a computer program with global variables available to modules.

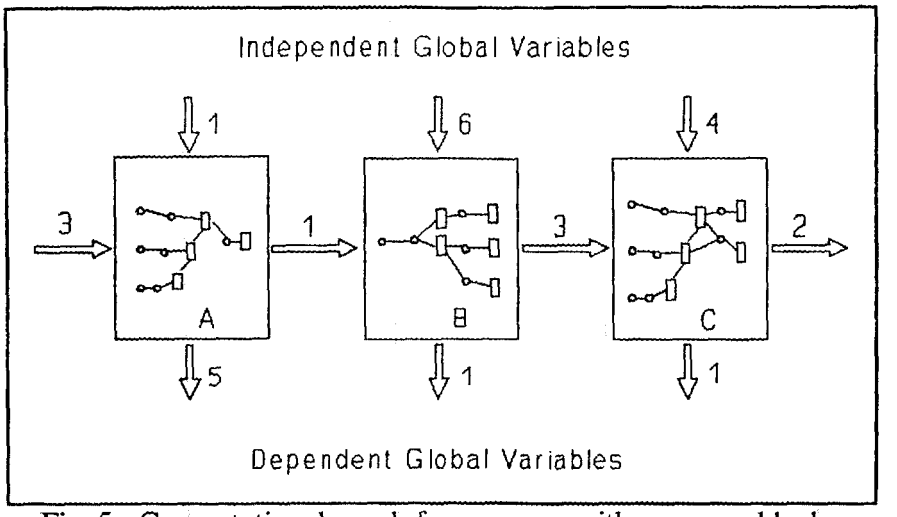

Fig. 5. Computational graph for program with common blocks.

When determining whether to use forward or reverse methods for calculating the derivatives for a module, the independent global variables are counted as input variables, and the dependent global variables are counted as output variables. If global variables are included, module A should use forward mode, and modules B and C should use reverse mode.

For MDT to be feasible, the number of variables on the links between modules must be small compared with the number of variables within the modules. The more modular a code system, the more effectively one could implement MDT. A module can be as simple as a subroutine or function; however, the composition of a module is arbitrary. For example, in a code that does hundreds of iterations, each iteration could be treated as a module. Though in the long term completely automating MDT is recommended, the intent in this paper is to test MDT with existing technology.

4. GENSUB STRUCTURE AND DESIGN

The chain rule of differentiation has a natural linked-list structure. In designing GENSUB we took advantage of this natural structure for storing derivatives in memory. A linked-list structure can easily be traversed in forward or reverse directions, thus allowing the forward or reverse method of automatic differentiation to be selected after the module is completed.

The GRESS GENSUB option was developed to permit testing of MDT. GENSUB is used to process a subset of a program (i.e., a do loop, subroutine, function, a sequence of subroutines, or a whole program) for calculating derivatives of dependent variables with respect

to independent variables. GENSUB allows the processing of program units as small as a do loop, and as large as an entire program. GENSUB can use either forward or reverse chaining, depending on which is most efficient for the given application.

GENSUB takes advantage of the modularity in many FORTRAN programs. A module may be thought of as any sequence of FORTRAN statements with identifiable input and output. A subroutine or function with input and output communicated via argument lists would be an obvious example of a module. However, the input and output could also be in common blocks. While a module is considered to be a sequence of statements, a program can be thought of as a sequence of modules that are linked. The following section presents the linked-list structure used to store and calculate derivatives for a single module, followed by a discussion of the linking of modules.

4.1. LINKING FLOATING-POINT ASSIGNMENT STATEMENTS

GENSUB was designed to work efficiently for small- to medium-sized modules. The strategy is to minimize work during the sequential processing assignment statements. Information necessary to propagate derivatives is accumulated without reduction in a structure that allows solving for derivatives in either the forward or reverse mode.

Each time an assignment statement is executed, a GRESS library routine will construct a **Statement Frame**, as illustrated in Fig. 6. A sequence of Statement Frames contains all the information necessary to apply the chain rule of differentiation in either forward or reverse mode to calculate derivatives of selected dependent variables with respect to variables that are input to a module. Memory required to hold the Statement Frames and solve for derivatives is allocated dynamically.

As shown in Fig. 7, Statement Frames form a linked-list structure that can be traversed in either the forward or reverse direction. The information stored in the linked-list structure can be used to calculate the derivative of any calculated result with respect to any term on the right-hand side of an assignment statement.

When a module is completed, derivatives are calculated for output variables with respect to input variables. Input and output variables for a module are identified by the user through the insertion of subroutine calls to the GRESS library. The user also inserts a call to a routine that calculates the derivatives by application of the chain rule. The chain routine will determine whether forward or reverse mode is best for a given application. The user can also specifically request forward or reverse mode. Once the derivatives are calculated for the output variables with respect to the input variables, the memory used to hold the Statement Frames is released.

Statement pointer (SP)
Pointer to previous Statement Frame (PSF)
Pointer to next Statement Frame (NSF)
Number of non-zero derivatives (NT)
Derivative value (DV)
Pointer to right-hand-side term (RHS)
. . . DV and RHS are repeated NT times

Statement pointer is the address of a floating-point variable defined by the statement.
Pointer to right-hand-side term is an address of a variable on the right of an equal sign.

Fig. 6. Structure of a Statement Frame.

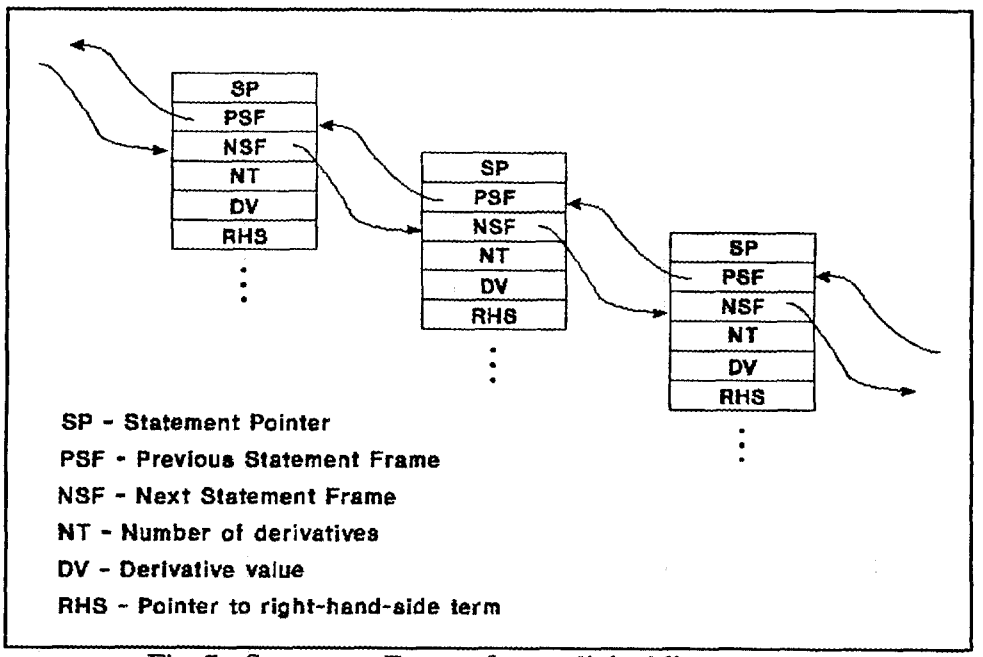

Fig. 7. Statement Frames form a linked-list structure.

4.2. LINKING MODULES

Linking modules is analogous to linking statements. To link statements we constructed Statement Frames using a linked-list structure. Module Frames can also be stored in a linked-list structure.

A sequence of statements define a module. When the Statement Frames within a module are solved using the chain rule, in either forward or reverse mode, the information necessary to construct a Module Frame is retained. A Module Frame contains the derivatives of the dependent variables output from the module with respect to the independent variables input to the module. The output from a module has three components: (1) derivatives, (2) dependent variables, and (3) independent variables. Each dependent and independent variable is assigned a row number. A row number is stored in a random access data structure using the associated variable's address as a key for later retrieval.

When a Module Frame is constructed, dependent variables that are not dependent on declared parameters are dropped. Parameters can be declared either by the user or automatically, as any real variable input via a read statement. The row numbers for the dependent variables can be determined by position; therefore, the row numbers for dependent variables do not have to be saved in the Module Frame. The structure of a Module Frame is shown in Fig. 8.

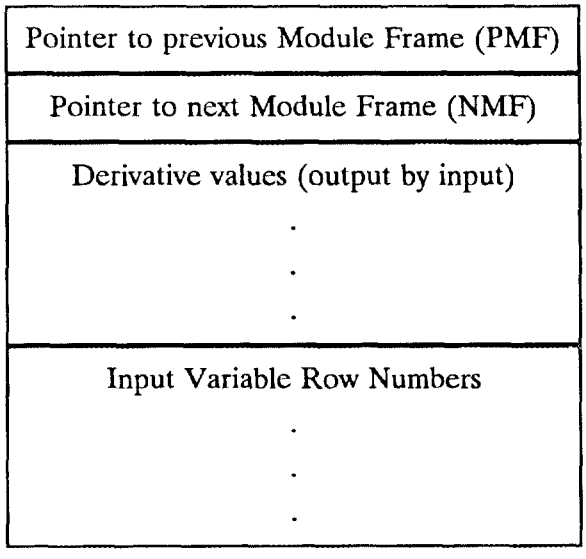

Fig. 8. Structure of Module Frames with input variable row numbers only.

The Module Frames for a program contain all the information necessary to propagate derivatives by either forward or reverse mode. There is flexibility in how to process a Module Frame. For the demonstration problem, Module Frames are stored in memory until the program is finished, and then the derivatives for the results of interest with respect to parameters are calculated. However, it should be possible to output the Module Frames to disk or to pipe them to another process. When a module is defined as a sequence of modules, then the Module Frames for the sequence can be resolved to a single Module Frame by forward or reverse mode of automatic differentiation.

The amount of savings in terms of execution time and memory requirements will depend on the application. For MDT to be effective, the size of the Module Frames must be small compared with the size of the Statement Frames. In other words, the number of variables on the links between modules must be kept at a minimum. For a program that could be represented as a sequence of modules, a significant reduction in storage could be realized with skillful implementation of MDT.

5. APPLICATION OF MDT USING THE GRESS GENSUB OPTION

An automated implementation of MDT would probably identify dependent and independent variables from argument lists and common blocks. GENSUB provides a mechanism by which MDT can be tested; however, identification of input and output variables to a module, as well as common block variables, must be made by the user through the insertion of subroutine calls to the GRESS run-time library. This section describes the application of MDT to a demonstration program. Results are presented and discussed.

If derivatives are to be calculated with the GENSUB option, independent variables must be declared at the beginning of the section of code being processed. Independent variables must have been assigned values before the section of code through which derivatives are to be propagated is executed. For example, if GENSUB is used to calculate the derivatives of the results from a subroutine with respect to the REAL variables provided as arguments into the subroutine, those arguments will have to be identified as independent variables on entry to the subroutine.

A run-time routine is also used to identify dependent variables. Dependent variables can be any floating-point variable calculated in the subroutine or section of code through which the derivatives are propagated.

The user must supply a two-dimensional, single-precision array for storing the derivatives. The array should be dimension N by M, where N is the number of dependent variables, and M is the number of independent variables declared in the subsection of the program. At the end

of the subsection (e.g., function or subroutine) being processed with the GENSUB option, the user should insert a call to a chain routine with the result array as an argument. The chain routine will apply the chain rule in either forward or reverse mode to solve for the derivatives of the dependent variables with respect to the independent variables. The derivatives will be returned to the calling program in the array provided by the user.

By default, the GENSUB option uses dynamic allocation. Depending on the application, the operating system, and the computer resources available, pre-allocating memory may be more efficient. The first time a section of code is processed, the chain routines provide information about memory usage. This information can be used in subsequent executions to specify or estimate the amount of memory to pre-allocate. GRESS routines are available for pre-allocating memory. Upon return from the chain routines the memory allocated for storing and propagating derivatives is released.

To demonstrate MDT, a sample problem with a main program and two subroutines was selected. Each subroutine is called per iteration in the main program. The number of iterations can be varied. Four parameters and one dependent variable are retained after each iteration. Three methods were used to process the sample problem: (1) the GRESS ADGEN option, used to implement reverse mode on the entire program; (2) the GENSUB option, treating each iteration as a module; and (3) the GENSUB option, treating each subroutine as a module. The sample problem selected is the test program provided on the GRESS distribution diskette. Of importance in this paper is that there are two subroutines and no global variables. Shown in Fig. 9 is a plot of the maximum amount of memory required to store derivatives using each method as a function of the number of iterations. Method 1 is provided for comparison because ADGEN requires the accumulation of derivatives for every arithmetic assignment statement.

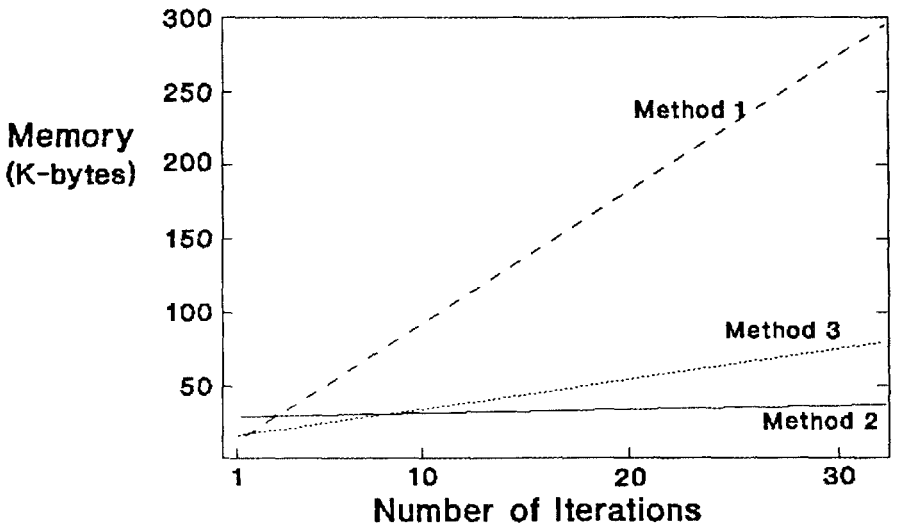

Fig. 9. Memory required to store derivatives using MDT vs reverse mode.

The results clearly demonstrate the fundamental problem with the reverse mode of automatic differentiation; that is, the memory required to store derivatives is proportional to execution time. Interestingly, Method 3 also shows a linear growth, though not as steep as Method 1. For this application, as the number of iterations increases, Method 2 would be the most feasible in terms of memory requirements. Memory requirement using Method 2 increases by 52 bytes per iteration. With an iterative code using Method 2, the expected increase per iteration would be the size of the Module Frame used to link each iteration.

The results in Fig. 9 demonstrate that MDT is both practical and feasible. Though the sample problem is very limited in that it does not include common blocks and does require hand intervention in identifying modules, the results are very encouraging. Automating the procedure so that common block variables and variables on argument lists are automatically included as dependent or independent variables is conceptually straightforward. However, having the flexibility of allowing the user to identify modules is also desirable.

The conclusion that Method 2 would be best can only be made for this application. The comparison between two different implementations of MDT raises the question as to whether it would be viable to automatically process a code to determine which method would be most appropriate. Much of the information required may not be available until execution of the model. It may be more feasible to develop tools to enable the user to implement MDT in a semi-automated fashion.

6. CONCLUSIONS AND RECOMMENDATIONS

As implemented using the GRESS GENSUB option, MDT can significantly reduce the memory required to compute derivatives in a complex FORTRAN model, thus extending the size of problem to which the reverse mode can be applied. MDT provides a compromise between the forward mode with its computational limitations and the reverse mode with its excessive memory requirements.

The capability of automatically identifying modules and dependent and independent variables should be implemented; however, the flexibility of allowing the user to define modules and variables should be maintained.

The GENSUB option is a valuable addition to GRESS in that it allows the processing of individual program modules, as well as the testing of MDT. By using GENSUB it is possible to implement MDT in an existing code on a module-by-module basis or during the development of a new code.

Further investigation is required to determine to what level MDT should be automated. It is recommended that MDT be applied to a large-scale application. The results from such an application would provide invaluable information for future development and applications.

7. REFERENCES

1. J. E. Horwedel, *Matrix Reduction Algorithms for GRESS and ADGEN*," ORNL/TM-11261, Martin Marietta Energy Systems, Inc., Oak Ridge Natl. Lab., 1989.

2. J. E. Horwedel, R. J. Raridon, and R. Q. Wright, *Sensitivity Analysis of AIRDOS-EPA Using ADGEN With Matrix Reduction Algorithms*, ORNL/TM-11373, Martin Marietta Energy Systems, Inc., Oak Ridge Natl. Lab., 1989.

3. J. E. Horwedel, *GRESS Version 2.0 User's Manual*, ORNL/TM-11951, Martin Marietta Energy Systems, Inc., Oak Ridge Natl. Lab., 1991.

4. J. E. Horwedel, "GRESS, A Preprocessor for Sensitivity Studies of Fortran Programs," pp. 243-250 in *Proc. of SIAM Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Breckenridge, Col., January 6-8, 1991 (1991).

5. E. M. Oblow, *An Automated Procedure for Sensitivity Analysis Using Computer Calculus*, ORNL/TM-8776, Union Carbide Corp., Nucl. Div., Oak Ridge Natl. Lab., 1983. (Available from the National Technical Information Service, U.S. Dept. of Commerce, 5285 Port Royal Road, Spring Field, VA 22161).

6. E. M. Oblow, F. G. Pin, and R. Q. Wright, "Sensitivity Analysis Using Computer Calculus: A Nuclear Waste Application," *Nucl. Sci. Eng.* **94**, 46 (1986).

7. B. A. Worley, R. Q. Wright, F. G. Pin, and W. V. Harper, "Application of an Automated Procedure for Adding a Comprehensive Sensitivity Calculation Capability to the ORIGEN2 Point Depletion and Radioactivity Decay Code," *Nucl. Sci. Eng.* **94**, 180 (1986).

8. B. A. Worley, "Experience with the Forward and Reverse Mode of GRESS in Contaminant Transport Modeling and Other Applications," pp. 307-314 in *Proc. of SIAM Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, Breckenridge, Colorado, January 6-8, 1991 (1991).

9. A. Griewank, "Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation," Preprint MCS-P228-0491, Mathematics and Computer Science Division, Argonne National Laboratory (1991).

APPENDIX A

GENSUB SAMPLE PROBLEM

The sample problem called j1a.f provided with the GRESS distribution diskette was selected for the examples in this paper. This sample program was used to obtain the results shown in Fig. 9 for Method 1. The sample program was modified for implementation with the gensub option for Methods 2 and 3. Calls to library routines to identify local and global variables were inserted. A call to a routine to calculate the derivative of the final result with respect to the input parameters was also inserted. Below is a listing of the program prepared for Method 2. Directives to the SYMG precompiler and calls to GRESS library routines are shown in bold.

```
*gensub
C               SYMG/GRESS SAMPLE PROBLEM B.1.1
C
C       Purpose: To test single precision real number mathematical
C                operations supported by the GRESS run-time library.
C                Comments denote operation code(s) that are tested
C                in the next line of code.
C
      DIMENSION X(4),F(4,4,4)
        imalloc=0
C
      PRINT*,'** GRESS SAMPLE PROBLEM B.1.1    **'
      PRINT*,'*                                *'
      PRINT*,'* PLEASE ENTER:                  *'
      PRINT*,'*     1.3  3.0  4.0   4.5        *'
      READ(5,*) (X(I),I=1,4)
C
C Declare X array as global independent variables (locpgg)
C
      do 99 i=1,4
        call locpgg(x(i))
 99   continue
      PRINT*,'X(I),I=1,4)',X
      LOOP1 = 4
C
C lsumo specifies the number of iterations
C
      print*,' enter lsumo'
        read(5,*)lsumo
      print*,' lsumo =',lsumo
```

```
C
C TEST OPCODE # 2, 4
C
      D = 0.0
      DO 1 I = 1,LSUMO
C
C DECLARE D and array x to be local independent variables
C         (i.e., input to module)
C
            call genapxx(x,4)
            IF(I.GT.1) call genpxx(d)
C
C After first time through, pre-allocate memory
C
      if(imalloc.ne.0) call allocgg(imalloc,icalloc)
      CALL SUB1(I,A,B,X)
      CALL SUB2(I,F,X,LOOP1)
      FS = 0.0
      DO 2 J = 1,LOOP1
      DO 2 K = 1,LOOP1
      DO 2 L = 1,LOOP1
C
C TEST OPCODE # 18
C
      FS = FS + F(L,K,J)
    2 CONTINUE
      BFS = B + FS
      D = D + BFS
C
C DECLARE D to be a dependent variable to keep
C
      call genresxx(d)
C
C Create a module link
C
      call chainlink(dx,imalloc,icalloc)
C
C Record memory requirements.  Imalloc is memory in bytes.
C    Icalloc is memory in words preset to zero.
C
      write(60,1000)imalloc,icalloc
1000  format(1x,2i10)
    1 CONTINUE
      WRITE(6,9) D
    9 FORMAT(1H ,'D',1PE16.8)
C
C DECLARE D to be a global dependent variable (response)
C
```

```
      call potrgg(D)
C
C Solve for derivatives of D with respect to elements of x
C and return the values in DX array.
C
      CALL solvgg(dx)
      print *,' dx = ',(dx(i),i=1,4)
      STOP
      END
      SUBROUTINE SUB1(I,A,B,X)
      DIMENSION X(4)
C
C TEST OPCODE # 24, 27
C
      FX = X(1)/X(2) + X(3)*X(4)
      RANN = 0.0
      CALL GETRAN(RANN)
      A = RANN * FX
C
C TEST OPCODE # 42, 39, 43, 40
C
      B = ATAN(ABS(A/(A+3.1))) - SIN(SQRT(A/X(4)))
C
C TEST OPCODE # 41, 38
C
      C = ALOG(ABS(B)) + EXP(B/(B+2.5))
C
C TEST OPCODE # 45, 44
C
      B = A*B/C + ALOG10(ABS((C+A)/B)) + COS(ABS(C)/C**2)
C
C TEST OPCODE # 34, 33
C
      B = A**2/ABS(B)**1.02 * B
      RETURN
      END
      SUBROUTINE SUB2(I,F,X,LOOP1)
      DIMENSION X(4),F(4,4,4)
      DO 1 II = 1,LOOP1
      DO 1 J = 1,LOOP1
      DO 1 K = 1,LOOP1
      RANN = 0.0
      CALL GETRAN(RANN)
      FXR = X(3)**2/COS(RANN**2) - SQRT(RANN*X(4)*X(2))
C
C TEST OPCODE # 46
C
      FXR = FXR*X(4) + MAX(X(1),X(2),X(3),X(4))
```

```
C
C TEST OPCODE # 47
C
      FXR = FXR - MIN(X(1),X(2),X(3),X(4))
C
C TEST OPCODE # 10, 55, 54
C
      FXR = FXR * FLOAT(MIN0(K,J,II))/FLOAT(MAX0(K,J,II))
      FXR = FXR*X(1)*X(1)*RANN*RANN
      F(K,J,II) = X(1)**RANN / EXP(RANN) + FXR*EXP(2.001*RANN)
    1 CONTINUE
      RETURN
      END
```

APPENDIX B

NEW GRESS RUN-TIME LIBRARY ROUTINES

Shown in Table B.1 are new GRESS run-time library routines that were added to test MDT. These routines have only undergone limited testing. Run-time library routines are used to control the application of the enhanced code. The following pages provide a description of each run-time library function. The format is one run-time library function per page. These routines are in addition to routines described in ref. 3.

Table B.1. New GRESS run-time library routines for the GENSUB option

Name	Purpose
LOCPGG	Defines a global independent variable
CHAINLINK	Creates a Module Frame
POTRGG	Defines a global dependent variable
SOLVGG	Solves derivatives in Module Frames

GENSUB Library Routine

Name: CHAINLINK(DERIVATIVE,MEM,ZMEM)

Function: To create a Module Frame

Arguments:

(1) DERIVATIVE - array to contain derivatives
(2) MEM - amount of memory used in bytes
(3) ZMEM - amount of memory preset to zero used in words

Argument type:

(1) A two-dimensional REAL array
(2) INTEGER
(3) INTEGER

Comment: CHAINLINK will apply the chain rule in either forward or reverse (adjoint) mode, depending on whether there are more local responses or more local parameters.

How to use it: Insert CALL CHAINLINK at the end of the section identified as a module. The derivatives of the responses declared using GENRESXX with respect to parameters declared using GENPXX, GENAPXX, or GENDPXX for the module will be calculated, and a Module Frame will be created. DERIVATIVE must be a two-dimensional array, with the first dimension being the number of local dependent variables (responses) and the second dimension being the number of local independent variables (parameters).

GENSUB Library Routine

Name: LOCPGG(VAR)

Function: To declare VAR to be a global independent variable

Arguments:

(1) VAR - variable to be declared a global parameter

Argument type:

(1) REAL

How to use it: Insert CALL LOCPGG after the variable has been initialized or defined. Parameters for a GENSUB application must be independent of the section of enhanced code through which derivatives are to be propagated. That means that the call to LOCPGG must occur upon entering the subprogram or section of code that has been enhanced. Also, parameters that appear on the left of assignment statements will automatically be redefined as variables; therefore, the assignment statement that defines the parameter must not be part of the enhanced code.

Example:

(1) Declare Y to be a global parameter for a GENSUB application.

```
*gensub
    SUBROUTINE ALPHA(Y,R)
    CALL LOCPGG(Y)
      .
      .
      .
```

GENSUB Library Routine

Name: POTRGG(VAR)

Function: To declare VAR to be a global dependent variable (response)

Arguments:

(1) VAR - variable to be declared a global response

Argument type:

(1) REAL

How to use it: Insert CALL POTRGG after the variable has been defined. Responses for a GENSUB application should be dependent on the section of enhanced code through which derivatives are to be propagated.

Example:

(1) Declare Y to be a global response for a GENSUB application.

```
*gensub
    SUBROUTINE ALPHA(Y,R)
        .
        .
        .
    Y = B*R**2
    CALL POTRGG(Y)
        .
        .
        .
```

GENSUB Library Routine

Name: SOLVGG(DERIVATIVE)

Function: To calculate the derivatives for a GENSUB application by applying the chain rule in reverse mode (adjoint mode) to Module Frames.

Arguments:

(1) DERIVATIVE - array to contain derivatives

Argument type:

(1) A two-dimensional REAL array

How to use it: Insert CALL SOLVGG at the end of the section of enhanced code through which derivatives have been propagated. The derivatives of the responses declared using POTRGG with respect to parameters declared using LOCPGG for the subsection of code enhanced for GENSUB will be calculated and returned in the array DERIVATIVE. DERIVATIVE must be a two-dimensional array with the first dimension being the number of dependent variables (global responses) and the second dimension being the number of independent variables (global parameters). A one-dimensional array is sufficient if there is only one dependent variable or one parameter.

INTERNAL DISTRIBUTION

1. B. R. Appleton
2. S. A. Bartell
3. J. M. Bownds
4. R. W. Brockett (Consultant)
5. J. J. Dongarra
6. J. J. Dorning (Consultant)
7. M. B. Emmett
8. D. M. Hetrick
9-13. J. E. Horwedel
14. D. Ingersoll
15. J. E. Leiss (Consultant)
16. R. McLean
17. T. Mitchell
18. N. Moray (Consultant)
19. M. D. Morris
19. L. F. Norris
20. E. M. Oblow
21. F. G. Pin
22. C. H. Shappert
23. R. C. Ward
24. R. M. Westfall
25. G. E. Whitesides/R. P. Leinius
26. B. A. Worley
27. R. Q. Wright
28. Central Research Library
29. ORNL Y-12 Research Library
30. Document Reference Section
31. Laboratory Records Department
32. Laboratory Records ORNL (RC)
33. ORNL Patent Office

EXTERNAL DISTRIBUTION

34. Office of the Assistant Manager for Energy Research and Development, DOE Field Office, Oak Ridge, P.O. Box 2008, Oak Ridge, TN 37831
35. Laura McDowell-Boyer, Grand Junction Office, P.O. Box 2567, Grand Junction, CO 81502
36. J. R. Cook, Interim Waste Technology Division, Savannah River Laboratory, P.O. Box 616, Aiken, SC 29802
37. J. E. Dennis, Jr., Rice University, Department of Mathematical Sciences, P.O. Box 1892, Houston, TX 77251
38. John M. Kallfelz, Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland
39. D. W. Muir, IAEA Nuclear Data Section, P.O. Box 200, A-1400 Vienna, Austria
40. A. D. Yu, Interim Waste Technology Division, Savannah River Laboratory, P.O. Box 616, Aiken, SC 2980

41-50. Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831