10/1-9-92 JS (1)

ORNL/TM-11951

ornl

OAK RIDGE
NATIONAL
LABORATORY

MARTIN MARIETTA

GRESS Version 2.0 User's Manual

J. E. Horwedel

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

ORNL/TM--11951

DE92 004592

Engineering Physics and Mathematics Division

GRESS VERSION 2.0 USER'S MANUAL

J. E. Horwedel
Computing and Telecommunications Division
Martin Marietta Energy Systems, Inc.
P. O. Box 2008
Oak Ridge, TN 37831-6370

Date Published - November 1991

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

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 B. M. Horwedel and R. Q. Wright of the Computing and Telecommunications Division for their helpful technical discussions and their careful review of the manuscript. Thanks are also due to Steven M. Robbins for his help in developing the GENSUB option.

PREFACE

Manual Objectives

The primary objective of this manual is to provide a description of GRESS and to explain how to use GRESS to enhance FORTRAN 77 models for gradient calculation.

The use of the GRESS precompiler, SYMG, is presented. A complete description of how to enhance a source code for either forward or reverse propagation of derivatives using the chain rule is provided.

Programming information is also provided to aid in the installation and maintenance of the software.

Intended Audience

This manual is intended for programmers who have a basic understanding of calculus and FORTRAN 77.

1. SOFTWARE ABSTRACT

1.1. Software Identification

GRESS Version 2.0.

1.2. Description of Function

The GRESS FORTRAN precompiler (SYMG) and run-time library are used to enhance conventional FORTRAN programs with analytic differentiation of arithmetic statements.

1.3. Method of Solution

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 or subprograms by adding the calculation of derivatives along with the original output. Derivatives from a GRESS enhanced model can be used internally (e.g., iteration acceleration) or externally (e.g., sensitivity studies). By calling GRESS run-time routines, derivatives can be propagated through the code via the chain rule (referred to as the CHAIN option) or accumulated to create an adjoint matrix (referred to as the ADGEN option). A third option, GENSUB, makes it possible 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.

1.4. Restrictions

GRESS accepts a majority of ANSI X3.9-1978 standard FORTRAN 77. Limitations are presented in appendix A. Application programs with FORTRAN statements or characters not recognized by GRESS will require modification prior to precompilation. An ADGEN application requires the accumulation of derivatives; therefore, the size of problem to which it may be applied is limited by the amount of virtual memory or disk space available.

1.5. Computers

VAX, IBM RISC/6000, SUN, Hewlett-Packard 9000.

1.6. Execution Time

Execution time for both precompiler and enhanced application program are problem dependent. Execution time for application programs will increase significantly after enhancement. On a VAX 8600 computer, the precompiler will process an application FORTRAN program at a rate of approximately 1000 lines of code per 4 seconds of CPU time.

1.7. Programming Languages

FORTRAN 77; C.

1.8. Operating Systems

VAX/VMS, UNIX, AIX, ULTRIX.

1.9. Machine Requirements

GRESS can be implemented on VAX/VMS, VAX/ULTRIX, IBM RISC/6000, Hewlett-Packard 9000, and SUN computers. The computer resources required are application dependent. The amount of memory needed can increase by more than a factor of two after precompilation depending on the application. To store an adjoint matrix requires a direct access storage device. Though the amount of storage or memory necessary is application dependent, it can be excessive.

1.10. Author

J. E. Horwedel

1.11. Configuration Control Facility

Radiation Shielding Information Center (RSIC)
P.O. Box 2008
Building 6025, MS-6362
Oak Ridge, TN 37831-6362

1.12. References

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

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

2. INTRODUCTION

This chapter includes background information and a brief overview of the system. A complete description of how to use the system in applications, including sample problems, is provided in Chapter 3. Code limitations and sample problems are provided in the appendices.

2.1. Background Information

Computer programs with varying levels of complexity are being used in modeling and design activities at a rapidly increasing rate. Understanding the behavior of predictive models with respect to input data is important for (1) verifying the validity of a model, (2) determining parameters for which it is important to have accurate values, and (3) understanding the behavior of the system being modeled. This importance is increasingly recognized by modelers, reviewers, and decision makers.

Sensitivity analysis of computer model results is one approach in determining the effect of input data on model predictions. Traditionally, most sensitivity analyses have relied on direct parameter perturbations (i.e., slightly altering the value of one parameter and re-executing the model). However, sensitivity analysis is often limited to a subset of the model parameters because of the immense amount of input data used by many computer models. The selection of parameters to be included in a study is generally based on engineering analysis and judgment or expert opinion. With the complexity of present-day computer models, reliance on subjective methods for parameter selection can be a severe limitation.

For complex computer models used in engineering design and assessment, a cost-efficient procedure for identifying the parameters that are important to a given model prediction is a necessity. The GRESS code was developed to meet this need. GRESS employs a precompiler, SYMG, to add derivative-taking capabilities to FORTRAN computer programs. Early implementations of GRESS were used to calculate sensitivities of model results to user-selected input parameters, but the present version of GRESS includes the ability to calculate and report the sensitivities of model results to all input data. The GRESS technology makes it possible to rapidly perform a comprehensive sensitivity analysis of FORTRAN models.

As originally developed, GRESS used forward propagation of derivatives via the calculus chain rule referred to as the CHAIN option. Later the capability to automate the adjoint sensitivity methods (ADGEN option) into existing computer codes was developed. The ADGEN option in GRESS generates an adjoint matrix. With the ADGEN option derivatives are accumulated and solved in virtual memory or written to disk. Utility routines are used to calculate derivatives based on the adjoint approach. A third option, GENSUB, is now available that allows processing of program units as small as a do loop or as large as an entire program. GENSUB will use either forward or reverse chaining depending on which is most efficient for the given problem. The remainder of this manual discusses in detail the use of SYMG for

CHAIN, ADGEN, and GENSUB applications. Though the emphasis is on these applications, SYMG is designed to allow extension to other computer calculus applications.

2.2. Definitions

Within the scope of this manual the following terms are defined.

Adjoint Matrix. The accumulated partial derivatives from all (or a subset of all) floating point assignment statements executed in the enhanced FORTRAN program.

Dependent Variable. Any program-calculated real number variable whose value is at least partially determined by one or more independent variables.

Enhanced Model. The reference model enhanced for gradient calculation.

Forward Solution. The results that are obtained by running the code prior to precompilation.

Independent Variable. Any real number variable, input or calculated, that is explicitly declared to be a parameter.

Parameter. Same as an independent variable.

Precompilation. A line-by-line translation of a FORTRAN program into an enhanced FORTRAN program.

Reference Model. The user's source code prior to enhancement with SYMG.

Response. Any dependent variable selected by the user for gradient calculation.

2.3. System Overview

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, statement functions, block data subprograms, single precision variables, double precision variables, and equivalence statements. GRESS does not process COMPLEX variable types. Specific limitations are discussed in Appendix A.

The steps used to process a code with GRESS are illustrated in Fig. 2.1. 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.

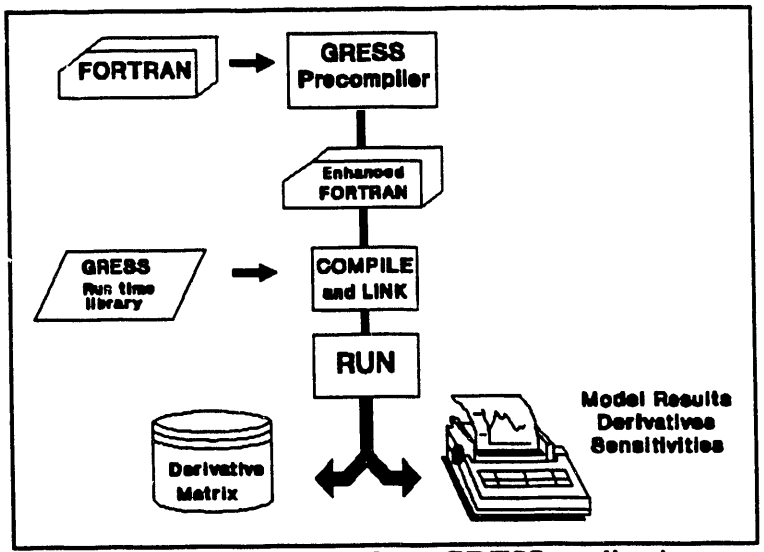

Fig. 2.1. Processing steps for a GRESS application

Derivatives from a GRESS-enhanced model can be used internally (e.g., for iteration acceleration) or externally (e.g., for sensitivity studies). GRESS can calculate and report derivatives or parameter sensitivities. The parameter sensitivities calculated by GRESS are the normalized first derivatives of output variables with respect to input parameters. The normalized sensitivity is calculated by multiplying a derivative by its associated input parameter value and dividing by the associated output value. The resulting sensitivity is therefore unitless. A normalized sensitivity of 0.1 means that, to the first order, a 1 percent change in that input parameter would cause a 0.1 percent change in the output. A report of significant sensitivities (i.e., usually those greater than 0.1) is generated.

GRESS provides two methods of calculating and reporting sensitivities. The CHAIN option calculates the sensitivities of a variable with respect to a user-selected subset of the input data by repeated application of the chain rule. The CHAIN option reports sensitivities 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 sensitivity analysis methods long used by nuclear engineers to calculate the sensitivities of selected model responses with respect to thousands of input parameters. This method, as implemented by

GRESS, is essentially the same as the reverse mode of automatic differentiation. When the ADGEN option is chosen, partial derivatives for every equation in the model are accumulated. The accumulated derivatives can be solved in virtual memory or output to a data set for later processing. Matrix-solving routines are then used to calculate and report sensitivities for selected results. The ADGEN option provides the user with the capability to calculate and report the sensitivity 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.

The first time a new model is processed, it is best to compare a few GRESS results with sensitivities estimated by perturbation methods to ensure that GRESS was applied correctly. Any differences between the GRESS-calculated analytic sensitivities and those calculated by parameter perturbation should be resolved.

2.3.1. Precompilation. During the precompilation step GRESS makes a single pass through a FORTRAN program. A symbol table entry is created for each FORTRAN symbol (i.e., variable name) as it is defined in a subprogram (i.e., function, subroutine, or main program). When a new subprogram is encountered the symbol table is re-started. As READ statements are encountered, logic is generated to initialize any impacted REAL variables. Statements defining REAL variables are parsed (1) to determine mathematical operations and (2) to create a statement table. The statement table contains the name and type of the FORTRAN variable(s) used in the assignment statement as well as a character string representation of the variable. The character string representation of a variable is the variable's name plus any dimensional information that may be included with each occurrence of the variable. For example, in the statement X(I) = Y(I)*X(J), X and Y are variable names, and 'X(I)', 'Y(I)', and 'X(J)' are character strings representing the usage of the variables. Using the statement table and the defined mathematical operations, 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 FORTRAN program is used to demonstrate precompilation.

```
C Test program input to GRESS
      DOUBLE PRECISION X(4),Y(4),A,B
      READ(5,*) (X(I),I=1,4)
      DO 10 I=1,4
      Y(I)=X(I)*A + X(I)*B
10    CONTINUE
      END
```

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.

```
      REAL DX(50)
      COMMON /ZZZZQ1/ DX
      DOUBLE PRECISION X(4),Y(4),A,B
      READ(5,*) (X(I),I=1,4)
      DO 90001 I=1,4
      CALL INNDXX(X(I),'X',I,1,1,1,1)
90001 CONTINUE
      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
      END
```

The call to subroutine INNDXX immediately following the READ statement serves to initialize the array X. The partial derivatives are initially stored in the DX array. Subroutine LOCNXX is a GRESS routine that will move the partial derivatives into a buffer for later processing. The call to subroutine LOCNXX is generated after the original FORTRAN statement so that it does not degrade optimization by the FORTRAN compiler. By default comments beginning with a 'C' or 'c' in column one are dropped by GRESS. As an option the user can direct GRESS to pass all comments through to the generated code.

As each arithmetic assignment statement is parsed, a statement table is generated. For purposes of derivative calculation, the mathematical operations to solve the equation are broken into unary and binary operations on terms in the statement table. A symbolic representation of the adjoint matrix for the FORTRAN equation is set up and solved for the result on the left of the statement, Y(I), with respect to the variables on the right, X(I), A, and B. Figure 2.2 shows a sequence of binary operations that would compute the FORTRAN statement from the example; the figure also shows the resulting symbolic adjoint matrix.

Internally GRESS creates a symbolic adjoint matrix and then symbolically solves the adjoint matrix for each assignment statement as it is processed. Because the adjoint method is used to calculate the derivatives, only symbolic addition and multiplication operations are required, which greatly simplifies the coding of the GRESS precompiler. Once the symbolic adjoint matrix is created, the derivatives of the term on the left with respect to the variables on the right are resolved. Finally, the FORTRAN necessary to calculate those derivatives during execution is generated. The user selects and controls the application by inserting FORTRAN subroutine calls to the SYMG run-time library.

To compute:

$$Y(I) = X(I) \cdot A + X(I) \cdot B$$

Generate temporary terms:

$$T1 = X(I) \cdot A$$
$$T2 = X(I) \cdot B$$
$$Y(I) = T1 + T2$$

$$\begin{array}{c} \\ \textbf{Symbolic} \\ \textbf{adjoint matrix} \end{array} \qquad\qquad \begin{array}{c} \textbf{Adjoints} \\ \textbf{of } Y(I) \end{array}$$

$$\begin{array}{c} X(I) \\ A \\ B \\ T1 \\ T2 \\ Y(I) \end{array} \begin{bmatrix} 1 & 0 & 0 & A & B & 0 \\ 0 & 1 & 0 & X(I) & 0 & 0 \\ 0 & 0 & 1 & 0 & X(I) & 0 \\ 0 & 0 & 0 & 1 & 0 & 1.0 \\ 0 & 0 & 0 & 0 & 1 & 1.0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & A & B & A+B \\ 0 & 1 & 0 & X(I) & 0 & X(I) \\ 0 & 0 & 1 & 0 & X(I) & X(I) \\ 0 & 0 & 0 & 1 & 0 & 1.0 \\ 0 & 0 & 0 & 0 & 1 & 1.0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Fig. 2.2. Creating and solving a symbolic adjoint matrix

2.3.2. Controlling the application. The user inserts application dependent subroutine calls to control the execution. For the CHAIN option the user must identify parameters and results of interest. This option is most efficient for cases involving a small number of parameters. The CHAIN application allows derivatives for an unlimited number of calculated results to be reported without greatly increasing the resource requirements.

For an adjoint application, parameters and results of interest must also be identified. Adjoint methods are most efficient for cases involving a large number of parameters with only a few results of interest. Automatic and manual declaration of parameters are included as options to the user. Results of interest must be specified by the user by insertion of subroutine calls to GRESS library routines.

Once the calls to subroutines to control the application are inserted, the enhanced code is ready for compilation with the FORTRAN compiler and link-editing. The result is an executable version of the enhanced FORTRAN model. Since CHAIN uses forward propagation of derivatives via the calculus chain rule, first derivatives and sensitivities are calculated along with the normally calculated model results. The method and format for reporting derivatives and sensitivities is under the control of the user and is discussed in detail in Chapter 3. An ADGEN application, however, accumulates a matrix of partial derivatives. The partial derivatives are either accumulated in virtual memory or output to a direct access storage device. Run-time library routines are available for solving the adjoint matrix in memory. The BSOLVE program solves the adjoint equation and calculates first derivatives from accumulated partial derivatives that have been written to disk.

2.3.3. Optimization algorithms used with ADGEN. Two optimization techniques were developed to improve the implementation of the ADGEN option by reducing the number of elements in the adjoint matrix: (1) forward reduction; and, (2) back reduction. Forward reduction eliminates those terms that are not dependent on the input data and is implemented by default. Back reduction further reduces the data stored by keeping only those terms that impact the user-selected results and is implemented by using the BREDUCE program or the REDUXX run-time library routine.

When performing a sensitivity analysis of existing FORTRAN 77 programs the user is often interested in only a subset of the actual FORTRAN equations that are solved. Figure 2.3 illustrates forward- and back-reduction algorithms applied to a sample program. Because the variable D is not declared as a parameter and is not dependent on any parameters, the forward-reduction algorithm treats D as a constant. Any partial derivatives with respect to D are set to 0.0. The result R is not dependent on variable S; therefore, back reduction drops the column associated with S from the matrix. Only the circled terms are needed to completely calculate the derivatives of the result, R, with respect to the declared parameters, Q and W.

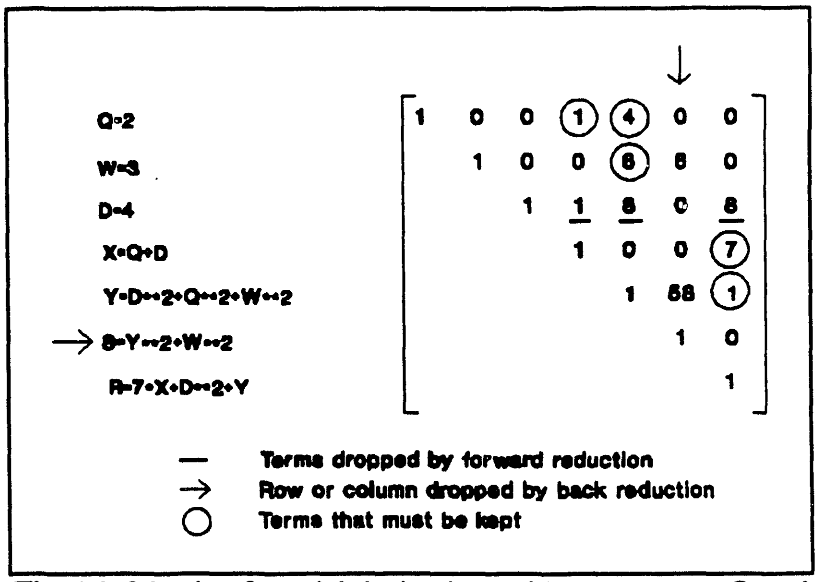

Fig. 2.3. Matrix of partial derivatives with parameters Q and W, and result R

3. APPLICATION INFORMATION

This chapter provides the basic information needed to use the SYMG precompiler. The processing steps are discussed in detail. The method for controlling the application is presented. Each command and utility routine is described with examples. Sample problems are provided in the appendices that exercise most of the major program options.

3.1. Precompilation Step

The first step in processing a code for either the CHAIN or adjoint application is the precompilation step.

3.1.1. Source code as input. The FORTRAN program to be enhanced is the input data to the precompiler. During precompilation, the code is translated and enhanced for gradient calculation. The source code must be written in FORTRAN 77 with some language restrictions. Directives to the precompiler may be inserted into the source code prior to enhancement. Subroutine calls to control the application can be inserted before or after precompilation.

SYMG is **not** a FORTRAN compiler. It is absolutely necessary that there are no FORTRAN syntax errors in the source code. Syntax errors that would cause a FORTRAN compilation to fail could cause SYMG to go into an infinite loop. If the user modifies the source code prior to enhancement, the code should be re-compiled with a FORTRAN compiler to ensure that no syntax errors were introduced.

Specific FORTRAN 77 language limitations are discussed in Appendix A. In general, those functions that would cause mathematical discontinuities are not supported. Complex functions are not supported. In this case, not supported means that derivatives will not be calculated with respect to those functions. However, in most situations, the enhanced code will still calculate the forward solution correctly.

Due to language restrictions and limitations, it may be necessary to replace lines of code with logically equivalent, but SYMG-compatible, lines of code. It is not always possible to know in advance which lines of code will need modification. Therefore, the precompilation is generally an iterative process. The user should review the code for obvious incompatibilities. After precompilation, it may be necessary to make further modifications to the source code based on error messages or other information obtained from the precompilation step. The precompilation step would then be repeated.

3.1.2. Precompiler directives. Prior to enhancement, the user may insert directives into the code to control the precompiler. With the exception of the *CHAIN directive, which is required for the CHAIN application, all other directives are optional. Table 3.1 shows the available directives and their function. An expanded discussion of each directive with examples is provided at the end of this chapter. The use of the directives in applications is shown in the appendices.

Table 3.1. Directives to the SYMG precompiler

Command	Function
*CHAIN	Invokes CHAIN option
*COMMENT OFF	Stop passing comments to enhanced code
*COMMENT ON	Causes comments to be passed to enhanced code
*DEBUG	Causes some debug information to be generated
*ECHO OFF	Stop echoing unenhanced statement in enhanced code
*ECHO ON	Echo unenhanced statement as comment in enhanced code
*GENSUB	Invokes GENSUB option
*ITABLE	Sets hashing table size
*LOCROWS	Maximum number of local rows for *OPTIMIZE option
*LOCTOT	Sets hashing table size for BSOLXX routine
*MAXPAR	Maximum number of parameters for adjoint case
*MAXRES	Maximum number of responses for adjoint case
*MAXROWS	Maximum number of rows for adjoint matrix in memory
*OPTIMIZE	Statements with one variable are chained forward in ADGEN case
*SYMG OFF	Pass input code to output with out enhancing
*SYMG ON	Resume enhancing input code
*WSPSIZE	Allows user to specify work space size

3.1.3. <u>Execution of the precompiler</u>. Once the code is prepared for precompilation, it is necessary to make the appropriate logical unit assignments and execute SYMG. Shown in Table 3.2 are the default logical unit numbers and their purpose.

Table 3.2. Default logical units used by SYMG

Data set	Default Logical Unit	Description
Enhanced code	7	SYMG created code
Input code	50	Program to be enhanced

3.1.4. <u>Data sets created during precompilation</u>. The only data set created during precompilation is the enhanced source program. The default logical unit for the enhanced code is 7. The enhanced source program is a FORTRAN 77 program.

3.1.5. Output information from precompilation. The printed output from the precompiler is written by default to logical unit 6. If the run is successful, the information provided includes the number of subroutines or functions translated, and the number of input lines. Following is an example of the output to unit 6 from a successful precompilation.

```
                        SymG  Version 2.0

          Enhancing code for derivative/sensitivity calculations.

         SYMG HAS DETERMINED THERE ARE 2 USER FUNCTIONS.
                        USER FUNCTIONS:
                           DUNKUP
                           EXPH1
      ----- NUMBER OF PROGRAM MODULES TRANSLATED =  16
       ----- NUMBER OF INPUT LINES TRANSLATED =    1482
```

Note that the names are provided for functions that SYMG has decided are user functions. This list must be reviewed carefully. SYMG does not check the list of user functions to ensure that those functions are actually in the code. Names listed that are not user defined functions indicate possible problems. Unsupported functions would be listed as user functions. Also, arrays erroneously not recognized by SYMG could appear on the list of user functions.

The following example shows the output from an unsuccessful precompilation. If the precompilation is unsuccessful, SYMG will usually generate error messages to help the user find the problem. Also, if a given line of FORTRAN is causing a problem, SYMG will include that line in the printed output as shown.

```
                        SymG  Version 2.0
          Enhancing code for derivative/sensitivity calculations
                  **** ERROR NUMBER  17   ****
         ***** DO WITHOUT LABEL NOT SUPPORTED *****
          CURRENT INPUT STATEMENT LINE NUMBER...    3

                           do i=1,4

          ... COMPILATION STOPPED ON FATAL ERROR...
```

The input line causing the difficulty is a do loop without a statment label. SYMG does not support do loops without statement labels. The user should add a statement label.

3.2. Controlling the Application

To control the application, the user inserts subroutine calls to GRESS run-time library routines. The calls to control the application may be inserted before or after precompilation. With no inserted calls the enhanced code provides the same results as the reference model. Inserted subroutine calls are used to declare the purpose of the run, define independent variables, identify results of interest, and retrieve selected gradients. The order and location of the inserted calls is extremely important. A description of each run-time library routine and any restrictions on its use is included at the end of this chapter.

3.2.1. CHAIN option. Table 3.3 includes the name and function of each run-time library routine that is available for the CHAIN application. If derivatives are to be calculated with the CHAIN option, the AUTOXX routine must be called. CALL AUTOXX must appear as the first executable line in the code.

Table 3.3. GRESS run-time library routines for the CHAIN option

Name	Purpose
AUTOXX	Specifies the number of parameters to be declared
BUSTXX	Estimates the amount for work space remaining
DEFAXX	Declares an array of parameters
DEFDXX	Declares a double precision parameter
DEFIXX	Declares a single precision parameter
DIAGXX	Prints out diagnostic information; stops if something is wrong
FILEXX	Change the default print file (e.g., from 6)
GETGXX	Retrieves gradient based on parameter name
GETNXX	Retrieves gradient based on parameter number
NINDXX	Returns the current number of declared parameters
PRNTDXX	Prints gradients and sensitivities for double precision result
PRNTXX	Prints gradients and sensitivities for single precision result

In the following example, AUTOXX is used to specify that derivatives are to be forward propagated using the chain rule. The second argument in the call to AUTOXX sets the upper limit on the number of parameters to be declared in this run to ten.

```
DIMENSION X(4),F(4,4,4),RS(8)
INTEGER I00001(5)
REAL R00001(5)
DOUBLE PRECISION D00001(5)
call autoxx(-1,10)
      .
      .
      .
```

To prepare an enhanced code for forward propagation of derivatives, it is necessary to initialize the gradient work space. This is accomplished with the AUTOXX utility routine. CALL AUTOXX specifies the total number of parameters, N, to be declared in a run and must be executed prior to defining any parameters. Calls to either DEFIXX or DEFAXX can be inserted after the location in the code that initializes the parameter value (e.g., READ(5,100) X, Y=9.9, etc.). The define routines would be used to declare N parameters, where N is the value of the second argument in the call to AUTOXX. Derivatives are then retrieved with the GET routines (i.e., GETNXX or GETGXX) or reported with the PRNTXX routine. The following example shows a partial listing of an enhanced code using DEFIXX to declare two parameters and PRNTXX to retrieve and report sensitivities.

```
INTEGER I00001(5)
REAL R00001(5)
DOUBLE PRECISION D00001(5)
call autoxx(-1,2)
      .
      .
      .
READ(5,100) X
call defixx(X,' X ')
Y=9.9
call defixx(Y,' Y ')
      .
      .
      .
D=X*Y
call prntxx(D)
STOP
END
```

3.2.2. <u>Preparing the enhanced code to create an adjoint matrix</u>. For adjoint matrix generation the user must declare the purpose of the run, declare parameters, declare potential responses, and either clear the matrix buffers or solve the matrix in memory. The name and function of utility routines used in an adjoint application are defined in Table 3.4. SETRXX is used to specify that the purpose of the run is to generate an adjoint matrix. The user must insert at least one CALL POTRXX (or POTDXX for a double precision variable) to identify a response of interest. POTRXX will create an entry in the response dictionary for the requested dependent variable. The response dictionary is kept in memory until either the adjoint matrix is solved, or it is output to disk if the accumulated derivatives are output for later processing. CLEARXX clears the matrix buffers. BSOLXX solves the matrix in virtual memory. In most situations, either CALL BSOLXX or CALL CLEARXX will be the last executed statement before ending the job.

Table 3.4. GRESS run-time library routines for an ADGEN application

Name	Purpose
CLEARXX	Clears all gradient output buffers
DECLARXX	Toggles automatic parameter declaration
DIAGXX	Prints out diagnostic information; stops if something is wrong
DLIBXX	Adds double precision variable to parameter dictionary
DLINXX	Adds double precision variable to parameter dictionary with a counter
DRAYXX	Adds double precision array to parameter dictionary with counter
POTDXX	Declares a double precision result for derivative calculation
POTRXX	Declares a single precision result for derivative calculation
RLIBXX	Adds single precision variable to parameter dictionary
RLINXX	Adds single precision variable to parameter dictionary with a counter
RRAYXX	Adds a single precision array to parameter dictionary with a counter

REAL or DOUBLE PRECISION variables input via a FORTRAN read statement are by default automatically added to the parameter dictionary. If the automatic declaration is acceptable, there are no other necessary calls to insert. However, if either the automatic declaration is not acceptable or additional variables that were not "read" are desired as parameters, variables can be added to the parameter dictionary by inserting subroutine calls to the appropriate GRESS run-time library routines. The automatic declaration feature can be toggled with the DECLARXX utility. Refer to Table 3.4 for library routines that can be used to include parameters in the parameter dictionary.

3.2.3. GENSUB option. Table 3.5 includes the name and function of each run-time library routine that is available for the GENSUB option. 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. They must have been assigned values before the section of code is executed. Run-time routine GENRESXX is used to identify responses. The user must supply a two-dimensional, single-precision result array for storing the derivatives. The result array should be dimensioned N by M, where N is the number of dependent variables, and M is the number of independent variables declared in the sub-section of the program. At the end of the sub-section (e.g., function or subroutine) being processed with the GENSUB option, the user should insert a call to subroutine CHAINGG with the result array as an argument. CHAINGG 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 result array.

Table 3.5. GRESS run-time library routines for the GENSUB option

Name	Purpose
ALLOCGG	Pre-allocates memory
CHAINFOR	Solves derivatives by forward mode
CHAINGG	Solves derivatives by forward or reverse mode
CHAINREV	Solves derivatives by reverse mode
GENPXX	Defines an independent parameter
GENAPXX	Defines a single precision array of independent parameters
GENDPXX	Defines a double precision array of independent parameters
GENRESXX	Declares a dependent variable (response)

Several examples using GENSUB are provided in the appendices.

3.3. Compiling and Linking the Enhanced Code

The FORTRAN 77 compiler and link editor used with the code prior to enhancement are also used to compile and link the enhanced code. The only difference is in the link step. The object module for the enhanced code must be linked with the appropriate GRESS run-time library. Examples of the link step are shown in the appendices. The commands for compiling and linking are dependent on the operating system and will vary.

3.4. Controlling the Execution of the Enhanced Code

The enhanced code is executed essentially the same as it was before enhancement. Calls to the GRESS run-time library control the type of application and specify additional data sets, if any, to be created. Table 3.6 shows the default logical units used by GRESS at run-time to output a generated adjoint matrix for processing with BSOLVE. The logical units used are application dependent. Conflicts between logical units used by GRESS and those required by the application program must be resolved by the user.

For an ADGEN application, the created adjoint matrix may be extremely large (i.e., in the hundreds of megabyte range). The size is dependent on the size of the user program and the amount of CPU time required to execute the enhanced model.

Table 3.6. Logical units used when an adjoint matrix is output to disk

Data Set	Logical Unit	Description
Parameter dictionary	42	selected parameters
Response dictionary	43	selected results
Adjoint matrix:		
DERIV.DAT	46	derivative values
COLUMN.DAT	47	column numbers
NPAIRS.DAT	48	non-zero derivative count

3.5. Output Information

The printed output from execution of the enhanced code is very application dependent. The majority of the printed output is usually the same output that would be generated by the reference model. Any additional output is dependent on utility routines that are called by the user in controlling the application. For example, with GRESS applications, some users insert subroutine calls to CHAIN utility routines to retrieve derivatives into arrays, and then add write statements to report the results. The output is under complete control of the user. Users should refer to the run-time library reference at the end of this chapter for descriptions of any printed output from a specific run-time library routine. Error messages are reported by default to logical unit 6.

3.6. Data Sets Created During an Adjoint Application

Data sets discussed in this section are those created and used during an ADGEN application. CHAIN applications propagate derivatives in memory and do not create any data sets unless under the explicit control of the user.

3.6.1. Parameter dictionary. The parameter dictionary is used to store the symbol name, row number, and parameter value for any parameter during an execution of the enhanced code for an adjoint application. The data set is formatted to allow easy editing with any standard editor. Parameters are automatically defined as any single or double precision real number that is input via a FORTRAN read statement. Parameters may also be manually defined by the user using run-time library routines(i.e., RLIBXX, RLINXX, RRAYXX, DLIBXX, DLINXX, or DRRAYXX). The automatic declaration of parameters is the default; however, automatic declaration can be switched on or off with the DECLARXX utility. The parameter dictionary should be output to disk only if the adjoint matrix is output by calling CLEARXX.

3.6.2. Response dictionary. The response dictionary is used to store the symbol name, row number, and value for any dependent variable declared to be a response during an execution of the enhanced code for an adjoint application. The data set is formatted to allow easy editing with any standard editor. Responses are manually defined by the user using run-time library routines POTRXX or POTDXX. The response dictionary should be output to disk only if the adjoint matrix is output by calling CLEARXX.

3.6.3. Adjoint matrix. The derivative matrix is stored in three buffers: NPAIRS, COLUMN, and DERIV. If the buffers become full, they are written to disk. NPAIRS contains the number of non-zero derivatives in a column. COLUMN contains the row numbers for each partial derivative stored in DERIV. There is a one-to-one correspondence between COLUMN and DERIV. This structure was selected because it allows reading the matrix from top to bottom for forward chaining or bottom to top for back solving. Sufficient memory must be available for these three buffers if the user wishes to solve the matrix during the execution of the enhanced code. The adjoint matrix should only be written to disk if the user is intending to solve the adjoint matrix in another job step (e.g., using the BSOLVE utility).

3.7. Solving the Adjoint Matrix

Sample problems using the ADGEN option are provided in the appendices. Provided in this section is a brief overview of the various choices available to the user. When the ADGEN option is applied to a new code, the recommended procedure is to first identify a small sample problem that executes all the major mathematical paths through the model. Then, calculate sensitivities for two or three model results with respect to all data that are input via FORTRAN read operations. GRESS will generate a report of significant sensitivities (i.e., usually those greater than 0.1). The adjoint matrix is solved by back substitution. The BSOLVE utility is used if the adjoint matrix is output to disk. BSOLVE can be run on the adjoint matrix with or without prior execution of the BREDUCE program. If the matrix is solved in memory during execution of the enhanced code, then either BSOLXX or FBSOLXX run-time library routines may be used.

3.7.1. BSOLVE program. BSOLVE calculates the derivatives of the dependent variables in the response dictionary with respect to each term in the parameter dictionary by back substitution. BSOLVE will request a cutoff value for determining which sensitivities and derivatives to report. Derivatives and sensitivities will be reported for those parameters with sensitivities greater than or equal to the cutoff value.

3.7.2. BREDUCE program. BREDUCE implements the back reduction algorithm and can be used to reduce the size of the adjoint matrix on disk. BREDUCE can only be run one time because it changes the internal structure of the adjoint matrix. BREDUCE is most useful in applications that have more than one response. In a UNIX environment, BREDUCE will create data sets DERIV1.DAT, COLUMN1.DAT, and NPAIRS1.DAT. These should be renamed to DERIV.DAT, COLUMN.DAT, and NPAIRS.DAT, respectively, prior to running BSOLVE.

3.7.3. BSOLXX, FBSOLXX run-time library routines. BSOLXX and FBSOLXX are run-time implementations of the BSOLVE utility. If sufficient memory is available the adjoint matrix can be solved during the execution of the enhanced code. The only output will be the report of sensitivities. BSOLXX or FBSOLXX can only be called once during the execution of the enhanced code.

3.8. Detailed Description of Precompiler Directives

Commands to the precompiler are used to control the creation of the enhanced code. The following pages provide a description of each command. The format is one command per page. Each page includes at least one example on how to use the routine.

All commands to the precompiler must begin with the asterisk (i.e., *) in column 1. An asterisk was chosen because it is a legal comment to FORTRAN 77 compilers and does not impede syntax checking using the FORTRAN compiler.

PRECOMPILER DIRECTIVE

Name:	*CHAIN
Purpose:	To generate enhanced code for CHAIN option.
How to use it:	Must begin in column one.
Example:	Enhance program for CHAIN option.

```
*CHAIN
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

PRECOMPILER DIRECTIVE

Name:	*COMMENTS ON/OFF
Purpose:	To cause SYMG to pass comments from code being translated to the enhanced code.
Notes:	Only comments indicated with a lowercase or uppercase C in column one are passed. Blank lines and comments indicated by an asterisk in column one are not passed. Use COMMENTS ON if you want to selectively pass comments. The default is COMMENTS OFF.
Example:	Pass comments through to enhanced code.

```
*COMMENTS ON
      DIMENSION X(100)
      COMMON/ ALPHA/ Y,Z
          .
          .
          .
```

PRECOMPILER DIRECTIVE

Name:	*DEBUG
Purpose:	To cause SYMG to generate debug information.
Notes:	The DEBUG directive will cause a list of subroutines and functions to be printed with the output from SYMG. The DEBUG directive will cause the precompiler to continue after some errors; therefore, with DEBUG it is possible that the enhanced code is not usable.
Example:	Get a list of modules processed.

```
*DEBUG
     DIMENSION X(100)
     COMMON/ ALPHA/ Y,Z
         .
         .
         .
```

PRECOMPILER DIRECTIVE

Name:	*DERIV
Purpose:	To override the default name GRESS assigns to the derivative.
How to use it:	Must begin in column one.
Example:	To change the name of the derivative variable from the default (DX) to 'DER'.

```
*DERIV DER
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

Comment: By default, GRESS will use the variable name DX to store the derivatives from an assignment statement. Occasionally, the name DX conflicts with a variable in the user's program. The *DERIV directive makes it simple to change the string that is used by GRESS to store the derivative. The only restriction is that the name assigned to the derivative must be a legal FORTRAN symbol not exceeding six characters in length (i.e., a sequence of one to six letters or digits, the first of which must be a letter).

PRECOMPILER DIRECTIVE

Name:	*ECHO ON/OFF
Purpose:	To cause SYMG to echo the line of code being translated as a comment in the enhanced code. This can be very helpful in debugging.
How to use it:	ECHO can be used to echo the entire code as comments in the enhanced code, or to selectively echo part of the enhanced code. The default is ECHO OFF. This may be useful if SYMG is not working correctly and the user wants to check a line to see if it was enhanced correctly.
Example:	Echo one assignment statement in the enhanced code as a comment.

```
DIMENSION X(100)
 COMMON/ ALPHA/ Y,Z
    .
    .
        .
  *ECHO ON
X(I)=Z*Y + 5.0
  *ECHO OFF
```

PRECOMPILER DIRECTIVE

Name:	*GENSUB
Purpose:	To generate enhanced code for GENSUB option.
Notes:	The GENSUB option allows processing of program units as small as a do loop or as large as an entire program (for derivative calculation only). GENSUB will use either forward or reverse chaining depending on which is most efficient for the given problem. The GENSUB option is ideal for enhancing a single subroutine or function for derivative calculation.
How to use it:	Must begin in column one.
Example:	Enhance function FOOBAR for GENSUB derivative calculation.

```
*gensub
   REAL FUNCTION FOOBAR(X,R)
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

PRECOMPILER DIRECTIVE

Name:	*ITABLE
Purpose:	To override the default number of rows in the offset table.
How to use it:	Must begin in column one.
Example:	Increase the number of rows in the offset table to 20

```
*ITABLE 20
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

Comment: During execution of the enhanced program for a CHAIN or ADGEN application a variable is tracked based on its address in virtual memory. For each address used as a REAL variable, a location is assigned in an offset table for storing the row number or gradient work space location that GRESS assigns to that variable. The address is hashed into a segment and offset by dividing by 32768 (i.e., 2^{15}). The segment is the integer quotient. The offset is the remainder. The segment becomes an index to a pointer vector, where a location of a row in the offset table is stored. A SEGMENT-OFFSET pair calculated in this way provides a unique key for any word address in virtual memory. The first time a segment occurs, it is assigned a location in the offset table and a pointer to that location is stored in the pointer vector. ITABLE specifies the number of rows in the offset table. For large codes the default value may be too small. A call to the GRESS run-time library routine DIAGXX will tell how many rows are being used in the offset table at the point where the call is made.

PRECOMPILER DIRECTIVE

Name:	*LOCROWS
Purpose:	To set the maximum number of local rows to be held in memory when the *OPTIMIZE directive is specified.
How to use it:	Must begin in column one.
Note:	SEE *OPTIMIZE
Example:	Set the maximum number of local rows to 9000

```
*LOCROWS 9000
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

PRECOMPILER DIRECTIVE

Name:	*LOCTOT
Purpose:	To set the maximum number of 100 word segments that are available to the BSOLXX routine for solving the adjoint matrix in memory.
How to use it:	Must begin in column one.
Note:	See the BSOLXX run-time library routine. If BSOLXX fails because there are too few segments, an error message will suggest that LOCTOT be increased. The ideal size for LOCTOT is application dependent; however, for most applications the number of rows in the adjoint matrix divided by 500 per response should be sufficient.
Example:	Increase LOCTOT to 200

```
*LOCTOT 200
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

PRECOMPILER DIRECTIVE

Name:	*MAXPAR
Purpose:	To set the maximum number of parameters to be held in memory for an ADGEN application.
How to use it:	Must begin in column one.
Note:	If the adjoint matrix is to be written to disk, the actual number of parameters can exceed MAXPAR. However, if the adjoint matrix is to be solved in memory (see BSOLXX or FBSOLXX), then the actual number of parameters must be less than MAXPAR.
Example:	Increase MAXPAR to 20000

```
*MAXPAR 20 000
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
          .
          .
          .
```

PRECOMPILER DIRECTIVE

Name:	*MAXRES
Purpose:	To set the maximum number of responses to be held in memory for an ADGEN application.
How to use it:	Must begin in column one.
Note:	If the adjoint matrix is to be written to disk, the actual number of responses can exceed MAXRES. However, if the adjoint matrix is to be solved in memory (see BSOLXX or FBSOLXX), then the actual number of responses must be less than MAXRES.
Example:	Increase MAXRES to 10

```
*MAXRES 10
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
       .
       .
       .
```

PRECOMPILER DIRECTIVE

Name:	*MAXROWS
Purpose:	To change the default value for the maximum number of adjoint matrix rows to be held in memory for an ADGEN application.
How to use it:	Must begin in column one.
Note:	If the adjoint matrix is to be written to disk, the actual number of rows can exceed MAXROWS. However, if the adjoint matrix is to be solved in memory (see BSOLXX or FBSOLXX), then the actual number of rows must be less than MAXROWS. **If the adjoint matrix is to be written to disk, MAXROWS should be divisible by 256.** If the adjoint matrix is to be written to disk, it probably is not necessary to change the default value for MAXROWS. MAXROWS is also used to estimate the amount of storage for the adjoint matrix; therefore, it may be necessary for MAXROWS to be greater than the number of rows in the adjoint matrix for some applications.
Example:	Increase MAXROWS to 262,144

```
*MAXROWS 262144
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

PRECOMPILER DIRECTIVE

Name: *OPTIMIZE

Purpose: To cause floating point assignment statements that have only one variable on the right to be forward chained during an ADGEN application.

How to use it: Must begin in column one.

Note: When considering multiple responses, forward chaining statements that have only one variable on the right reduces both the number of terms and the number of calculations required to solve the adjoint matrix. In the following FORTRAN sequence, the statement defining B has only one variable on the right, A.

```
B = A**2 + 3.0*A
  .
  .
  .
Z = 2.0 * B
```

The partial derivative of B with respect to A is placed in a local buffer of size LOCROWS (see *LOCROWS directive). When B appears on the right of the statement the partial derivative of Z with respect to A is immediately calculated and stored in the adjoint matrix. The partial derivative of B with respect to A is never put in the adjoint matrix. Rather than two rows being added to the adjoint matrix, only one row is added. The effectiveness of this algorithm greatly depends on the code being enhanced for derivative calculations and the number of responses selected.

Limitations: Can only be used with BSOLXX and FBSOLXX. Cannot be used if matrix is to be written to disk.

Example:

```
*OPTIMIZE
*LOCROWS 9000
   DIMENSION X(100)
   COMMON/ ALPHA/ Y,Z
      .
      .
      .
```

PRECOMPILER DIRECTIVE

Name: *SYMG ON/OFF

Purpose: To selectively specify lines or sections of code to be enhanced, or not enhanced.

How to use it: Can be used anywhere within a complete program module or to turn off the enhancement of enti.e subprograms. Enhancement cannot be turned off within a subroutine and then turned back on within another subroutine. The enhancement must be turned on to process the declaration section of a subprogram if it is turned on anywhere within that subprogram.

Example: Prevent one line from being enhanced

```
          DIMENSION X(100)
          COMMON/ ALPHA/ Y,Z
              .
              .
              .
      *SYMG OFF
          X(I)=Z*Y + 5.0
      *SYMG ON
```

PRECOMPILER DIRECTIVE

Name: *WSPSIZE

Purpose: To override the default work space size.

How to use it: Must begin in column one.

Example: Set the work space size equal to 2 million words:

```
*WSPSIZE 2 000 000
   DIMENSION X(100)
      COMMON/ ALPHA/ Y,Z
        .
        .
        .
```

Comment: Both CHAIN and ADGEN options use a common block area that can vary in size dependent on the application. The default work space size is usually set at 8 million 4 byte words; however, the actual size is installation dependent. The WSPSIZE command is useful if for any reason you desire a larger or smaller work space.

3.9. GRESS Run-Time Library Routines

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. Each page includes at least one example on how to use the routine.

3.9.1. GRESS ADGEN library routines.

ADGEN Library Routine

Name: BSOLXX (LUN1,LUN2,CUTOFF)

Function: To solve the adjoint matrix in virtual memory.

Comment: Solving the adjoint matrix in memory is limited by the amount of memory available. If sufficent memory is available, BSOLXX should more efficient than writing the adjoint matrix to disk.

Arguments:

(1) LUN1 - read CUTOFF from lun1. A zero value means use argument three as the CUTOFF value.
(2) LUN2 - write sensitivity report to lun2.
(3) CUTOFF - magnitude of the smallest sensitivity to report. A value of zero will result in all sensitivities being reported.

Argument Type:

(1) INTEGER
(2) INTEGER
(3) REAL

Note: WSPSIZE must be greater than (100*LOCTOT*MAXRES) + 100.

How to use it: A call to BSOLXX may be used in place of a call to CLEARXX at the end of program execution. A sensitivity report will be written to LUN2.

Example:

(1) To report sensitivities that are greater than 1.0E-4 to logical unit 95

```
      .
      .
      CALL BSOLXX(0,95,1.0E-4)
      STOP
      END
```

ADGEN Library Routine

Name:	CLEARXX
Function:	To clear the forward matrix buffers.
Arguments:	NONE
How to use it:	Generally, CALL CLEARXX should be inserted in the main program immediately before the STOP statement. CLEARXX must be the last executed call before ending the run. If the program exits at some point other than in the main program, it will be necessary to insert CALL CLEARXX at that point.

Example:

(1) Normal exit from a main program with a STOP statement

```
PROGRAM MAIN
   .
   .
   .
CALL CLEARXX
STOP
END
```

(2) Possible exit from other location

```
   .
   .
   .
IF(UNHAPPY) CALL CLEARXX
IF(UNHAPPY) STOP RETURN
END
```

ADGEN Library Routine

Name: DECLARXX (CHAR)

Function: To turn the declaration of parameters on or off.

Arguments: 'ON' or 'OFF'

Argument Type: CHARACTER

How to use it: DECLARXX is used to limit the number of parameters added to the parameter dictionary. The default is 'ON', meaning that any parameters read in will automatically be added to the parameter dictionary. To limit the number of parameters, use DECLARXX to turn off the automatic declaration at the start of the program. To have specific parameters added to the parameter dictionary, use DECLARXX to turn on the declaration of parameters immediately before the read statement in the unenhanced model.

Example:

(1) Using DECLARXX to cause only C to be declared a parameter.

```
  .
  .
CALL SETRXX('ADJOINT')
CALL DECLARXX('OFF')
  .
  .
CALL DECLARXX('ON')
READ(100,1001)C
CALL DECLARXX('OFF')
  .
  .
  .
```

ADGEN Library Routine

Name: DIAGXX (LEVEL)

Function: To print diagnostic information.

Arguments:

(1) LEVEL

Argument type:

(1) LEVEL - INTEGER

<u>LEVEL</u>	<u>Action</u>
0	Check for failure conditions. Stop if error detected.
1	Print diagnostic information. Stop if error detected.
2	Print additional diagnostic information. Stop if error detected.
3	Check ADGEN control parameters. Stop if error detected.

How to use it: Insert CALL DIAGXX at any point in the program where you wish to print diagnostic information. LEVEL specifies the type of information as well as the action taken if an error is found. The user should review the diagnostic information provided by DIAGXX for obvious inconsistencies. For example, if DIAGXX shows that the work space was set at 1,000,000 and 1,400,000 words were used, the appropriate action would be to increase the work space and re-run the problem.

Example:

(1) Using DIAGXX to check status prior to calling BSOLXX. Code will stop if an error is detected.

```
      .
      .
CALL DIAGXX(2)
CALL BSOLXX(-1,45,0.01)
      .
      .
```

ADGEN Library Routine

Name:	DLIBXX (VARIABLE, NAME)
Function:	To add a double precision variable to the parameter dictionary.
Arguments:	(1) VARIABLE to be included (2) name or description of VARIABLE
Argument type:	(1) VARIABLE - DOUBLE PRECISION (2) NAME - CHARACTER*N (N < 12)
How to use it:	Insert CALL DLIBXX after the point in the code where a value is assigned to VARIABLE.

Example:

(1) To declare X to be a parameter in an ADGEN application.

```
DOUBLE PRECISION X
  .
  .
  .
X = 2.0D+01 * Y + Z
CALL DLIBXX(X,' X ')
  .
  .
  .
```

ADGEN Library Routine

Name: DLINXX (VARIABLE, NAME, COUNTER)

Function: To add a double precision variable to the parameter dictionary with a user-defined counter.

Arguments:

(1) VARIABLE to be included
(2) name or description of VARIABLE
(3) user defined counter

Argument type:

(1) VARIABLE - DOUBLE PRECISION
(2) NAME - CHARACTER*N (N < 12)
(3) COUNTER- INTEGER

How to use it: Insert CALL DLINXX after the point in the code where a value is assigned to VARIABLE.

Example:

(1) To declare X to be a parameter in an ADGEN application with the integer ICOUNT as a user-defined counter to help identify the result in the sensitivity report.

```
DOUBLE PRECISION X
  .
  .
  .
ICOUNT=ICOUNT+1
X = 2.0D+01 * Y + Z
CALL DLINXX(X,' X ',ICOUNT)
  .
  .
  .
```

ADGEN Library Routine

Name:	DRAYXX (ARRAY, N1, N2, N3, N4, NDIMS, NAME)
Function:	To add the elements in a double precision array to the parameter dictionary.

Arguments:

(1) ARRAY with up to four dimensions
(2-5) N1-N4 dimensions 1 to 4 of ARRAY
(6) NDIMS - actual number of dimensions
(7) name or description of ARRAY

Argument type:

(1) ARRAY - DOUBLE PRECISION
(2-5) N1-N7 - INTEGER
(6) NDIMS - INTEGER
(7) NAME - CHARACTER*N (N < 12)

How to use it: Insert CALL DRAYXX after the point in the code where values have been assigned to ALL elements to be included in the parameter dictionary. However, the call must be made prior to any element of ARRAY being used to assign a value to any other variable in the FORTRAN program. The values for N1-N4 must be set to at least 1, even if that dimension does not exist. For example, if an array has only two dimensions, arguments N3 and N4 must be assigned the value of 1.

Example:

(1) To declare the elements in array X to be a parameters in an ADGEN application.

```
DOUBLE PRECISION X(10,5)
  .
  .
  .
 CALL DRAYXX( X, 10, 5, 1, 1, 2, 'X')
  .
  .
  .
```

ADGEN Library Routine

Name: FBSOLXX (LUN1,LUN2,CUTOFF)

Function: To solve the ADGEN matrix in virtual memory.

Comment: This routine does not use memory as efficiently as BSOLXX; therefore, it should only be used for small models. However, if the model is small enough, FBSOLXX should execute faster than BSOLXX.

Arguments:

(1) LUN1 - read CUTOFF from lun1. A zero value means use argument three as the CUTOFF value.
(2) LUN2 - write sensitivity report to lun2.
(3) CUTOFF - magnitude of the smallest sensitivity to report. A value of zero will result in all sensitivities being reported.

Argument Type:

(1) INTEGER
(2) INTEGER
(3) REAL

Note: WSPSIZE must be greater than (MAXROWS*MAXRES) + 100.

How to use it: A call to FBSOLXX may be used in place of a call to CLEARXX at the end of program execution. A sensitivity report will be written to LUN2.

Example:

(1) To report sensitivities that are greater than 1.0E-4 to logical unit 95

```
      .
      .
CALL FBSOLXX(0,95,1.0E-4)
STOP
END
```

ADGEN Library Routine

Name: POTRXX (VARIABLE, NAME)
POTDXX (VARIABLE, NAME)

Function: To add a variable to the response dictionary during an ADGEN application.

Arguments:

(1) (VARIABLE) program variable to be declared
(2) (NAME) or description of VARIABLE

Argument Type:

(1) X - REAL or DOUBLE PRECISION
(2) 'CHAR' - CHARACTER*n (n < 24)

How to use it: Insert CALL POTRXX immediately following the line defining the variable. CLEARXX must be called at the end of the run. Use POTDXX if X is DOUBLE PRECISION. POTDXX is used exactly the same way as POTRXX.

Comment: If the adjoint matrix is output to disk, the response dictionary is written to logical unit 43. If the adjoint matrix is solved in memory, the response dictionary is kept in memory and used to write the report of sensitivities. An error occurs if part of the response dictionary is written to disk and the adjoint matrix is not. If the user discovers that the response dictionary was written to disk when it should not have been, the user should increase the maximum number of responses using the *MAXVAR directive.

Example:

(1) Declare D(5) to be a response of interest.

```
      .
      .
D(J) = B(J)**2
IF(J.EQ.5) CALL POTRXX(D(J),'  D of 5 ')
      .
      .
```

ADGEN Library Routine

Name: RLIBXX (VARIABLE, NAME)

Function: To add a single precision variable to the parameter dictionary.

Arguments:

(1) VARIABLE to be included
(2) name or description of VARIABLE

Argument type:

(1) VARIABLE - REAL
(2) NAME - CHARACTER*N (N < 12)

How to use it: Insert CALL RLIBXX after the point in the code where a value is assigned to VARIABLE.

Example:

(1) To declare X to be a parameter in an ADGEN application.

```
REAL X
   .
   .
   .
X = 2.0D+01 * Y + Z
CALL RLIBXX(X,' X ')
   .
   .
   .
```

ADGEN Library Routine

Name: RLINXX (VARIABLE, NAME, COUNTER)

Function: To add a single precision variable to the parameter dictionary with a user defined counter.

Arguments:

(1) VARIABLE to be included
(2) name or description of VARIABLE
(3) user defined counter

Argument type:

(1) VARIABLE - REAL
(2) NAME - CHARACTER*N (N < 12)
(3) COUNTER- INTEGER

How to use it: Insert CALL RLINXX after the point in the code where a value is assigned to VARIABLE.

Example:

(1) To declare X to be a parameter in an ADGEN application with the integer ICOUNT as a user defined counter to help identify the result in the sensitivity report.

```
REAL X
   .
   .
   .
ICOUNT=ICOUNT+1
X = 2.0D+01 * Y + Z
CALL RLINXX(X,' X ',ICOUNT)
   .
   .
   .
```

ADGEN Library Routine

Name: RRAYXX (ARRAY, N1, N2, N3, N4, NDIMS, NAME)

Function: To add the elements in a single precision array to the parameter dictionary.

Arguments:

(1) ARRAY with up to four dimensions
(2-5) N1-N4 dimensions 1 to 4 of ARRAY
(6) NDIMS - actual number of dimensions
(7) name or description of ARRAY

Argument type:

(1) ARRAY - REAL
(2-5) N1-NY - INTEGER
(6) NDIMS - INTEGER
(7) NAME - CHARACTER*N (N < 12)

How to use it: Insert CALL RRAYXX after the point in the code where values have been assigned to ALL elements to be included in the parameter dictionary. However, the call must be made prior to any element of ARRAY being used to assign a value to any other variable in the FORTRAN program. The values for N2-N4 must be set to at least 1, even if that dimension does not exist. For example, if an array has only two dimensions, arguments N3 and N4 must be assigned the value of 1.

Example:

(1) To declare the elements in array X to be a parameters in an ADGEN application.

```
REAL X(10,5,5)
    .
    .
    .
 CALL RRAYXX( X, 10, 5, 5, 1, 3, 'X')
    .
    .
    .
```

ADGEN Library Routine

Name:	SETRXX (CHAR)
Function:	To specify that the purpose of the run is to generate an adjoint matrix.
Argument:	'ADJOINT'
Argument Type:	CHARACTER*7
How to use it:	CALL SETRXX must be the first executed line in the code. If CALL SETRXX is NOT made, no derivatives will be stored in the adjoint matrix buffer.

Example:

(1) Enhanced code ready for an ADGEN application.

```
   .
   .
 DATA X /4.0/
 CALL SETRXX('ADJOINT')
   .
   .
```

(1) Unenhanced code preparing for adjoint matrix generation.

```
   .
   .
 DATA X /4.0/
 CALL SETRXX('ADJOINT')
   .
   .
```

3.9.2. GRESS CHAIN library routines.

CHAIN Library Routine

Name: AUTOXX (LUN, NUMP)

Function: To set the maximum number of parameters to be declared.

Arguments:

(1) LUN = -1 is required
(2) NUMP - maximum number of parameters to be declared

Argument Types: INTEGER

How to use it: CALL AUTOXX must be made before calling any other library routines.

Example 1. Specifying a maximum of five parameters in an enhanced code. (The call to AUTOXX must be the first executable statement in the enhanced code.)

```
  .
  .
  .
DATA X /4.0/
CALL AUTOXX(-1,5)
  .
  .
  .
```

Example 2. Specifying a maximum of twenty parameters in an unenhanced code

```
  .
  .
  .
DATA X /4.0/
CALL AUTOXX(-1,20)
  .
  .
  .
```

CHAIN Library Routine

Name:	BUSTXX
Function:	To report the status of the gradient work space during a CHAIN application.
Arguments:	NONE
How to use it:	CALL BUSTXX can appear anywhere within the executable part of the program. CALL BUSTXX can be called more than once. The output from BUSTXX will be written to logical unit six by default. The default logical unit can be changed with the FILEXX routine. The gradient work space is the buffer used to propagate derivatives.

Example:

```
      DIMENSION Y(10),X(10)
        .
        .
        .
      CALL BUSTXX
        .
        .
```

CHAIN Library Routine

Name: DEFAXX (ARRAY, NELEM, NTYPE)

Function: To declare elements in an array as parameters for a CHAIN application. No gradients are computed until at least one parameter is defined. Each call adds a new array to the list of parameters. Each element in the array counts as one parameter.

Arguments:

(1) ARRAY - array to be declared a parameter
(2) NELEM - number of elements in array to be declared
(3) NTYPE - argument type (1 = single precision, 2 = double precision)

Argument Type:

(1) REAL or DOUBLE PRECISION
(2) INTEGER
(3) INTEGER

How to use it: Insert CALL DEFAXX after the array has been initialized or defined. Subroutines SETRXX and AUTOXX must be called prior to CALL DEFAXX. For multi-dimensional arrays, the number of elements specified must take into consideration how FORTRAN treats dimensioned variables. DEFAXX will define the array elements as parameters, sequentially, in order of their location in memory.

Example:

(1) Declare the elements in array Y as parameters for a CHAIN application.

```
DIMENSION Y(10),X(10)
READ(LUN,100) Y
NUMP=10
NTYPE=1
CALL DEFAXX(Y(1),NUMP, NTYPE)
      .
      .
```

CHAIN Library Routine

Name: DEFDXX (VAR, NAME)

Function: To declare a double precision parameter for a CHAIN application. No gradients are computed until at least one parameter is defined. Each call adds a new parameter.

Arguments:

(1) VAR - double precision variable to be declared a parameter
(2) name or description of VAR

Argument type:

(1) VAR - DOUBLE PRECISION
(2) NAME - CHARACTER*N (N < 12)

How to use it: Insert CALL DEFDXX after the variable has been initialized or defined. Subroutine AUTOXX must be called prior to CALL DEFDXX.

Example:

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

```
DOUBLE PRECISION Y
    .
    .
READ(LUN,100) Y
CALL DEFDXX(Y,' Y ')
    .
    .
```

CHAIN Library Routine

Name: DEFIXX (VAR, NAME)

Function: To declare a single precision parameter for a CHAIN application. No gradients are computed until at least one parameter is defined. Each call adds a new parameter.

Arguments:

(1) VAR - single precision variable to be declared a parameter
(2) name or description of VAR

Argument type:

(1) VAR - SINGLE PRECISION
(2) NAME - CHARACTER*N (N < 12)

How to use it: Insert CALL DEFIXX after the variable has been initialized or defined. Subroutine AUTOXX must be called prior to CALL DEFIXX.

Example:

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

```
          .
          .
      READ(LUN,100) Y
      CALL DEFIXX(Y,' Y ')
          .
          .
```

(2) Declare D(5) to be a parameter.

```
          .
          .
      D(J) = B(J)**2
      IF(J.EQ.5) CALL DEFIXX(D(J),' D(5) ')
          .
          .
```

Name:	DIAGXX (LEVEL)
Function:	To print diagnostic information
Arguments:	(1) LEVEL
Argument type:	(1) LEVEL - INTEGER

LEVEL	Action
0	Check for failure conditions. Stop if error detected.
1	Print diagnostic information. Stop if error detected.
2	Print additional diagnostic information. Stop if error detected.
3	Check ADGEN control parameters. Stop if error detected.

How to use it: Insert CALL DIAGXX at any point in the program where you wish to print diagnostic information. LEVEL specifies the type of information as well as the action taken if an error is found. The user should review the diagnostic information provided by DIAGXX for obvious inconsistencies. For example, if DIAGXX shows that the work space was set at 1,000,000 and 1,400,000 words were used, the appropriate action would be to increase the work space and re-run the problem.

Example:

(1) Using DIAGXX to check status prior to calling PRNTXX. Code will stop if an error is detected.

```
        .
        .
    CALL DIAGXX(2)
    CALL PRNTXX(Y)
        .
        .
```

Name: FILEXX (LUN)

Function: To alter the logical unit number for all printed output generated by run-time library routines. The default logical unit number for printed output from the run-time library routines is 6.

Arguments: LUN - Logical unit number for printed output

Argument Type: INTEGER

How to use it: If the user chooses to have all or part of the calculated gradients from a CHAIN application written to a file other than unit 6, simply call FILEXX with an integer argument specifying the desired unit number. The assignment stays active until the end of the run, or until FILEXX is called again.

Example:

(1) To print all gradient results in a CHAIN application to logical unit 90.

```
      .
      .
      .
  LUN=90
  CALL FILEXX(LUN)
      .
      .
      .
```

CHAIN Library Routine

Name: GETGXX (X, Z)

Function: To retrieve an individual derivative using symbol name.

Arguments:

(1) X - any program variable
(2) Z - an array

Argument Type:

(1) REAL or DOUBLE PRECISION
(2) REAL

How to use it: GETGXX returns the derivatives of X with respect to the N declared parameters. Z must be dimensioned by at least N to hold the derivative of X with respect to each declared parameter.

Example:

(1) Retrieve the derivative of A with respect to all declared parameters. Store the derivatives in array B.

```
      .
      .
  DIMENSION B(8)
  CALL AUTOXX(-1,8)
      .
      .
      .
  CALL GETGXX(A,B)
      .
      .
      .
```

CHAIN Library Routine

Name: GETNXX (X, N, Z)

Function: To retrieve an individual derivative.

Arguments:

(1) X - any program variable
(2) N - parameter number
(3) Z - storage location

Argument Type:

(1) REAL or DOUBLE PRECISION
(2) INTEGER
(3) REAL

How to use it: GETNXX returns the derivative of X with respect to the N^th^ declared parameter. Parameters have an "ordinal" number corresponding to the sequence in which they are declared. It is necessary to provide a REAL variable as the third argument for storing the retrieved derivative.

Example:

(1) Retrieve the derivative of A with respect to the first, second, and fourth declared parameters. Store the derivatives in the first three locations in array ZZ.

```
      .
      .
      .
  CALL GETNXX(A,1,ZZ(1))
  CALL GETNXX(A,2,ZZ(2))
  CALL GETNXX(A,4,ZZ(3))
      .
      .
      .
```

CHAIN Library Routine

Name: NINDXX (N)

Function: To return the current number of declared parameters.

Arguments: N - number of declared parameters

Argument Type: INTEGER

How to use it: At any point during program execution, the number of parameters presently declared is returned as an integer argument.

Example:

(1) To check the number of declared parameters in a CHAIN application.

```
  .
  .
  .
CALL NINDXX (N)
IF(N.GT.3) THEN
  .
  .
  .
```

Name:	PRNTDXX (X)
Function:	To print the gradients and sensitivities of a double precision variable with respect to the declared parameters.
Arguments:	Any double precision program variable
Argument Type:	DOUBLE PRECISION
How to use it:	At any point during program execution, the gradient of a double precision variable may be printed by a call to PRNTDXX.

Example:

(1) To print gradients for a double-precision variable during a CHAIN application.

```
REAL Y, Z
DOUBLE PRECISION A, X
  .
  .
CALL DEFIXX(Y,' Y ')
CALL DEFDXX(A,' A ')
  .
  .
  .
 X = 2.0 * Y + Z
 CALL PRNTDXX(X)
  .
  .
  .
```

CHAIN Library Routine

Name: PRNTXX (X)

Function: To print the gradients and sensitivities of a single precision variable with respect to the declared parameters.

Arguments: Any program variable

Argument Type: REAL

How to use it: At any point during program execution, the gradient of a dependent variable may be printed by a call to PRNTXX.

Example:

(1) To print gradients at two places in a CHAIN application.

```
      .
      .
    READ(LUN,100) Y
    CALL DEFIXX(Y)
      .
      .
      .
     X = 2.0 * Y + Z
     CALL PRNTXX(X)
      .
      .
      .
     Z = X*B
     CALL PRNTXX(Z)
      .
      .
      .
```

3.9.3. GRESS GENSUB library routines.

GENSUB Library Routine

Name: ALLOCGG(MEM,ZMEM)

Function: To pre-allocate memory for a GENSUB application.

Arguments:

(1) MEM - amount of memory in bytes to be allocated
(2) ZMEM - amount of memory to be allocated and preset to zero.

Argument type:

(1) INTEGER
(2) INTEGER

Comment: SEE CHAINGG, CHAINFOR, or CHAINREV

How to use it: Insert CALL ALLOCGG prior to the section of code enhanced for the GENSUB option. It is recommended that the first time a section of code is run, that ALLOCGG not be used. However, if the same code is used again, the values from MEM and ZMEM as returned from one of the chain routines (CHAINGG, CHAINFOR, or CHAINREV) can be used. For large sections of code, pre-allocating memory should may result in faster execution.

Example:

(1) Declare array Y to be parameters for a GENSUB application with 500,000 bytes of pre-allocated memory and an additional 4000 words of pre-allocated memory preset to zero.

```
*gensub
   SUBROUTINE ALPHA(Y,R)
   DOUBLE PRECISION Y(50)
   CALL ALLOCGG(500000,4000)
   NELEM = 50
   CALL GENDPXX(Y,NELEM)
     .
     .
     .
```

GENSUB Library Routine

Name: CHAINFOR(DERIVATIVE,MEM,ZMEM)

Function: To calculate the derivatives for a GENSUB application by applying the chain rule in forward mode (CHAIN mode).

Arguments:

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

Argument type:

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

How to use it: Insert CALL CHAINFOR at the end of the section of enhanced code through which derivatives have been propagated. The derivatives of the responses declared using GENRESXX with respect to parameters declared using GENPXX, GENAPXX, or GENDPXX 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 (responses) and the second dimension being the number of independent variables (parameters).

Example:

(1) Use CHAINFOR to calculate derivatives of Z with respect to the elements in array Y.

```
*gensub
    SUBROUTINE ALPHA(Y,R,Z)
    REAL Y(2), DERIV(1,2)
    CALL GENAPXX(Y,2)
        .
        .
        .
    CALL GENRESXX(Z)
    CALL CHAINFOR(DERIV,MEM,IMEM)
```

Name: CHAINGG(DERIVATIVE,MEM,ZMEM)

Function: To calculate the derivatives for a GENSUB application.

Arguments:

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

Argument type:

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

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

How to use it: Insert CALL CHAINGG at the end of the section of enhanced code through which derivatives have been propagated. The derivatives of the responses declared using GENRESXX with respect to parameters declared using GENPXX, GENAPXX, or GENDPXX 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 (responses) and the second dimension being the number of independent variables (parameters).

Example: Use CHAINGG to calculate derivatives of Z with respect to the elements in array Y.

```
*gensub
    SUBROUTINE ALPHA(Y,R,Z)
    REAL Y(2), DERIV(1,2)
    CALL GENAPXX(Y,2)
        .
        .
    CALL GENRESXX(Z)
    CALL CHAINGG(DERIV,MEM,IMEM
```

GENSUB Library Routine

Name: CHAINREV(DERIVATIVE,MEM,ZMEM)

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

Arguments:

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

Argument type:

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

How to use it: Insert CALL CHAINREV at the end of the section of enhanced code through which derivatives have been propagated. The derivatives of the responses declared using GENRESXX with respect to parameters declared using GENPXX, GENAPXX, or GENDPXX 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 (responses) and the second dimension being the number of independent variables (parameters).

Example:

(1) Use CHAINREV to calculate derivatives of Z with respect to the elements in array Y.

```
*gensub
   SUBROUTINE ALPHA(Y,R,Z)
   REAL Y(2), DERIV(1,2)
   CALL GENAPXX(Y,2)
     .
     .
   CALL GENRESXX(Z)
   CALL CHAINREV(DERIV,MEM,IMEM)
```

GENSUB Library Routine

Name: GENAPXX(ARRAY,NELEM)

Function: To declare NELEM of a single precision ARRAY to be parameters for a GENSUB application.

Arguments:

(1) ARRAY - array to be declared a parameter
(2) NELEM - number of elements in array to be declared

Argument type:

(1) ARRAY - REAL
(2) NELEM - INTEGER

How to use it: Insert CALL GENAPXX after the array 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 GENAPXX 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 array Y to be parameters for a GENSUB application.

```
*gensub
      SUBROUTINE ALPHA(Y,R)
      REAL Y(50)
      NELEM = 50
      CALL GENAPXX(Y,NELEM)
          .
          .
          .
```

GENSUB Library Routine

Name: GENDPXX(ARRAY,NELEM)

Function: To declare NELEM of a double precision ARRAY to be parameters for a GENSUB application.

Arguments:

(1) ARRAY - array to be declared a parameter
(2) NELEM - number of elements in array to be declared

Argument type:

(1) ARRAY - DOUBLE PRECISION
(2) NELEM - INTEGER

How to use it: Insert CALL GENDPXX after the array 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 GENDPXX 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 array Y to be parameters for a GENSUB application.

```
*gensub
      SUBROUTINE ALPHA(Y,R)
      DOUBLE PRECISION Y(50)
      NELEM = 50
      CALL GENDPXX(Y,NELEM)
        .
        .
        .
```

GENSUB Library Routine

Name: GENPXX(VAR)

Function: To declare VAR to be a parameter for a GENSUB application.

Arguments:

(1) VAR - variable to be declared a parameter

Argument type:

(1) REAL or DOUBLE PRECISION

How to use it: Insert CALL GENPXX 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 GENPXX (or GENAPXX) 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 parameter for a GENSUB application.

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

GENSUB Library Routine

Name: GENRESXX(VAR)

Function: To declare VAR to be a response for a GENSUB application.

Arguments:

(1) VAR - variable to be declared a response

Argument type:

(1) REAL or DOUBLE PRECISION

How to use it: Insert CALL GENRESXX immediately after the variable has been defined. Responses for a GENSUB application must be dependent on the section of enhanced code through which derivatives have been propagated. The derivatives of R with respect to parameters that have been declared for the subsection of code enhanced for GENSUB will be calculated by calling one of the run-time library routines CHAINGG, CHAINFOR, or CHAINREV.

Example:

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

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

4. PROGRAMMER INFORMATION

This chapter provides information directed towards the person responsible for installing and maintaining the system. Changes may be required to allow implementation at a specified site. A brief description of the subroutines and functions in the GRESS run-time library and the SYMG precompiler is provided.

4.1. SYMG - Precompiler

SYMG is the precompiler for GRESS. The SYMG source code is written entirely in FORTRAN 77. SYMG is compiled and linked using a FORTRAN 77 compiler and link- editor. Shown in Table 4.1 is the name and function of each SYMG routine. The function descriptions are brief and are intended only as an overview.

4.2. GRESS Run-Time Library

The GRESS Run-Time Library has three parts, SGLIB, CLIB, and GENLIB. SGLIB is written in FORTRAN 77. CLIB and GENLIB are written in C. SGLIB and CLIB combine to form the run-time library required for ADGEN and CHAIN applications. GENLIB is the run-time library required for GENSUB applications. SGLIB is compiled using a FORTRAN 77 compiler. GENLIB and CLIB are compiled with a C compiler. Tables 4.2, 4.3, and 4.4 provide the name and function of library routines in SGLIB, CLIB, and GENLIB.

4.3. Implementation Problems

Because of the nature of derivative propagation through FORTRAN programs, GRESS enhanced codes can require an excessive amount of computer resources. The size of the enhanced code as determined by array sizes can be controlled with directives to the precompiler. However, for some computer systems, the default array sizes generated by GRESS are too big. The first step should be to have the system manager increase the amount of memory available to the maximum allowable for the system.

Table 4.1. SYMG subroutines and functions

Name	Function
AARG	generate quoted string in enhanced FORTRAN
ADDSTR	add two strings symbolically
ARGCHK	check argument list of user (or max/min) function
ARMY	process directive to precompiler
BLANKIT	set N characters to ' ' in TARGET string
BLKCHK	check for blank line in input file
BLKGEN	generate block data subprogram
BRANCH	generate label and continuation statement
BUMP	increment symbol storage pointers
CHARAC	process FORTRAN 77 character statement
COMDEF	process common statement
COPY	copy string from input to output buffer
DECLAR	output declaration for dummy arrays
DENEST	get inner-most symbol in term or statement
DIMENS	process dimension statement
DIVSTR	divide two strings symbolically
DOSTAT	process DO statement
ECHCMT	echo line of code as comment
ERROR	generate GRESS/SYMG error message
FINISH	complete creation of GRESS enhanced code
FIXTBL	define opcodes for statement function reference
FUNCHK	check arguments to user (or max/min) function
FUNREF	process user function reference
GEN	output string to enhanced FORTRAN
GENBUT	output temporary buffer
GENLOC	generate calls to process derivatives
GENNUM	convert number to character string
GENTMP	generate temporary buffer
GETC	"Get" next character and increment pointer
GETLAB	find label in DO statement
IFTRAN	transform 'IF(P)' into 'IF(.NOT.P) GO TO LABEL'
IMPLCT	process IMPLICIT statement
INCLUDS	process include statement
INIT	initialize SYMG
INTCAL	generate call to process derivatives
INTERP	interpret arithmetic operations
ITEM	find symbol reference in statement
IWRITE	convert integer to alphnumeric
LABGET	find statement label
LOOKUP	search symbol table for match
MAKLAB	generate statement label
MATFUN	keep record of user referenced functions
MODLST	add module to module list
MOVEC	copy N characters from source to TARGET
MULTSTR	multiply two strings symbolically
MYINT	generate derivatives from statement table
NAARG	place characters into enhanced FORTRAN
NBGEN	generate number in enhanced FORTRAN
NEXTC	"Get" next character (do not increment pointer)

Table 4.1. (continued)

Name	Function
NEXTS	"read" next input line into read-ahead buffer
NONEXS	process non-executable statements
NUMLIT	check for numeric literal
OUTCH	place character into output buffer
OUTLIN	output line of FORTRAN
OUTNUM	place integer into output buffer
PARSE	parse line of code for symbols and operations
POP	remove top of stack
PRECED	determine precedence of operations
PREFIX	process module header
PUSH	"push" operation onto STACK
PUT	place integer into op-code buffer (LBUF)
PUTINT	put arithmetic operation code into LBUF
PUTNUM	put load op-code for number into LBUF
PUTREF	put load op-code for symbol into LBUF
READC	process read list
READCL	generate calls to process array reads
READCS	generate calls to process scalar reads
RECORD	record operand reference
REEDSTR	move integer string from cbuf to bufjk
REEDCHR	move character string from cbuf to bufjk
REINIT	re-initialize cbuf pointers for reprocessing
REPROC	move cbuf to temporary buffer and create new cbuf
RFUNK	add entry to stack for intrinsic function
SAVTBL	save statement table
SMATCH	check to see if statement symbol is repeated
SETDIM	store dimension information in dimension table
SHOWIT	display input and terminate execution on error
SKIPNUM	skip number when pre-processing CBUF
SMOVE	convert character numbers to integers
SPECT	test for and process type declarations
STEST	test for occurrence of string in input buffer
STFUN	process statement function reference
SUBSTR	subtract two strings symbolically
SYMBOL	extract symbol from input buffer
TCODE	find and return SYMG character code for symbol
TABSCAN	replace tabs in input buffer with spaces
TERMEX	report compiler summary and stop execution
TYPES	process type declarations
VARREF	process variable or function reference
XTEST	test alphanumeric string for key word

On most systems the size of the enhanced code can be controlled by modifying parameter statements in the include file, symg.dec. The include file, symg.dec, contains parameters that have the biggest impact on the size of the enhanced code.

```
C  DEFAULT ARRAY SIZES GENERATED in enhanced code

      PARAMETER (IWORKSZ = 8 000 000) ! IWSPSIZE
      PARAMETER (ITAB= 7)            !ITABLE!
      PARAMETER (MAXR= 2**19 )        !MAXROWS!
      PARAMETER (LROWS=1000)          !LOCROWS!
      PARAMETER (LTOT=1500)          !LOCTOT!
      PARAMETER (MAXRXS=50)           !MAXRES!
      PARAMETER (MAXP[illegible]1 000)          !MAXPAR!
      PARAMETER (LSTOT=320 000)       !LSTTOT
```

LSTOT, MAXR, and LTOT are used for ADGEN applications when the adjoint matrix is to be solved in memory. If the adjoint matrix is to be output to disk, LSTOT and LTOT can be set equal to 2, and MAXR can be set equal to 512. To solve small adjoint matrices in memory for one response, set MAXRXS equal to 1, LTOT equal to 50, and MAXR equal to a value greater than the estimated number of rows in the adjoint matrix. If the *CHAIN or *gensub directive is specified, LSTOT, MAXR, and LTOT have no impact on the size to the enhanced code.

IWORKSZ and ITAB can impact the size of the code for either CHAIN or ADGEN options. For small problems, IWORKSZ can be set as low as 10,000 and ITAB as low as 2. However, IWORKSZ and ITAB can also be controlled by precompile directives *WSPSIZE and *ITABLE, respectively.

Table 4.2. SGLIB subroutines and functions

Name	Function
ACOSXX	derivative of ACOS
ALIBXX	create entry in ADGEN parameter dictionary
ALOGXX	derivative of ALOG
ASINXX	derivative of ASIN
ATANXX	derivative of ATAN
AUTOXX	set limit on number of parameters for CHAIN
BSOLXX	solve adjoint matrix
BUSTXX	report status of gradient work space
CHAINXX	propagate derivatives forward by chain rule
CLEARXX	output any buffers before exiting program
CLOSSS	close stack segment and push segment number
CLOSXX	return gradient slot to heap
DACOSXX	derivative of DACOS
DASINXX	derivative of DASIN
DATANXX	derivative of DATAN
DBUFXX	output ADGEN matrix buffer
DECLARXX	toggle ADGEN automatic declaration on or off
DEFAXX	declare array as parameters
DEFDXX	declare double precision variable as parameter
DEFIXX	declare single precision X as parameter
DEXPXX	derivative of DEXP
DIAGXX	check status and generate diagnostic output
DLIBXX	add D to parameter dictionary
DLINXX	declare double precision array as parameters
DLOGXX	derivative of DLOG
DORIXX	kill derivatives for X and declare it as independent
DRXPXX	derivative of double raised to real
DSQRTXX	derivative of DSQRT
ERRSXX	generate error message or warning message
EXPXX	derivative of EXP
FBSOLXX	solve adjoint matrix using fixed dimensions
FILEXX	define new output unit for CHAIN option
FRDUXX	reduce size of adjoint matrix generated
FUNIXX	flag return from integer function

Table 4.2. (continued)

Name	Function
GETGXX	retrieve gradients of variable
GETNXX	retrieve gradient of variable using parameter number
ILOCXX	find derivatives for term on right of equal sign
INNDXX	double precision formatted read with indices
INNRXX	single precision formatted read with indices
INSDXX	double precision formatted read no indices
INSRXX	single precision formatted read no indices
INUDXX	double precision unformatted read
INURXX	single precision unformatted read
IOPEXX	find or open hashing table row
KILLXX	kill gradient for specified variable
NBSOLXX	solve adjoint matrix with reduction
NBUFXX	output and ADGEN matrix buffer
NINDXX	return number of declared parameters
NRDUXX	reduce size of adjoint matrix
NROWXX	retrieve and report latest row number
OPENSS	open stack segment when requested
OPENXX	open gradient slot and return its location
POTDXX	create double precision entry in response table
POTRXX	create single precision entry in response table
PREPXX	initialize control parameters
PRNTDXX	prints gradients for double precision variable
PRNTXX	prints gradients for single precision variable
RDXPXX	derivative of real raised to double
RLIBXX	add variable to parameter dictionary
RLOCXX	determine status of variable
RRAYXX	declare single precision array as parameters
SDATA	report selected stack segment data
SETRXX	set adjoint flag to be true
SQRTXX	derivative of SQRT
UPERXX	convert lower case characters to upper case
WRITXX	construct string with array name plus dimensions
ZEROSP	zero out array

Table 4.3. CLIB routines and their function

Name	Function
afunxx	process statement with temporary variable for ADGEN
amaxxx	single precision max function with derivative
aminxx	single precision min function with derivative
cfunxx	process statement with temporary variable for CHAIN
dmaxxx	double precision max function with derivative
dminxx	double precision min function with derivative
dpldxx	returns double precision value given address
funaxx	process user function reference for ADGEN
funcxx	process user function reference for CHAIN
initxx	initialize hashing table parameters for error checking.
ivldxx	returns integer value given address
loccxx	process derivatives during CHAIN application
locixx	returns address of argument
locnxx	process derivatives in statement for ADGEN
locpxx	defines local parameter
loc2xx	processes statement with two variables
mlocxx	returns row number for its argument
movrxx	moves row from a local buffer to adjoint matrix
rvldxx	returns real value given address
stdpxx	stores double precision value at address
strvxx	stores single precision value at address
zerrxx	initializes derivatives for REAL variable
zerdxx	initializes derivatives for double precision variable

Table 4.4. GENLIB routines and their function

Name	Function
amaxgg	max function with derivative
amingg	min function with derivative
closcxx	close gradient slot
allocgg	allocate memory at users request
chainfor	calculates the derivatives by forward chaining
chainrev	calculates the derivatives by reverse mode
chaingg	calculates the derivatives by forward or reverse
dmaxgg	dmax function with derivative
dmingg	dmin function with derivative
genpxx	define independent variable (parameter).
genapxx	define single precision array as parameters
gendpxx	define double precision array as parameters
genresxx	define result of interest (response)
gen0xx	initialize derivative for a constant
genrxx	defines a row for a variable
inndgg	double precision formatted read with indices
innrgg	single precision formatted read with indices
insdgg	double precision formatted read no indices
insrgg	single precision formatted read no indices
inurgg	single precision unformatted read
inudgg	double precision unformatted read
mmalloc	allocates memory
mzalloc	allocates memory preset to zero

APPENDICES

APPENDIX A - LIMITATIONS

The limitations discussed in this section are in addition to limitations discussed elsewhere in the text. Every attempt was made to include the majority of FORTRAN 77 as specified in the ANSI X3.9-1978 FORTRAN standard. Extensions to the standard may or may not work. When the user is faced with a FORTRAN statement, or sequence of statements, that is not acceptable to GRESS, it is up to the programmer to decide on the course of action. If the sequence is important, then re-programming with logically equivalent, but GRESS-acceptable FORTRAN may be required.

A.1. Function Limitations

Function limitations fall into three categories: (1) Any ANSI X3.9-1978 FORTRAN 77 function that may lead to a discontinuity; (2) Complex functions; (3) Continuous ANSI X3.9-1978 FORTRAN 77 functions that GRESS does not presently process. Fortunately, the lists are short. Shown in Table A.1., are those standard functions that are not supported by GRESS because they could lead to a discontinuity.

Table A.1. ANSI X3.9-1978 FORTRAN 77 functions not supported by GRESS due to possible discontinuity

Name	Definition
AINT	truncation of REAL
DINT	truncation of DOUBLE PRECISION
ANINT	nearest whole number of REAL
DNINT	nearest whole number of DOUBLE PRECISION
AMOD	remaindering of REAL
DMOD	remaindering of DOUBLE PRECISION
DIM	positive difference of REAL
DDIM	positive difference of DOUBLE PRECISION

Table A.2 lists the ANSI X3.9-1978 complex functions. GRESS does not support complex functions.

Table A.2. ANSI X3.9-1978 COMPLEX functions not supported by GRESS

Name	Definition
AIMAG	imaginary part of COMPLEX number
CABS	absolute value of COMPLEX number
CCOS	cosine of COMPLEX number
CEXP	exponential of COMPLEX number
CLOG	natural logarithm of COMPLEX number
CMPLX	conversion to COMPLEX
CONJG	conjugate of COMPLEX number
CSIN	sine of COMPLEX number
CSQRT	square root of COMPLEX number

Shown in Table A.3 are those standard functions that are not supported by GRESS at the present time but may be supported by future releases of GRESS.

Table A.3. ANSI X3.9-1978 FORTRAN 77 functions that may be supported in future releases of GRESS

Name	Definition
DPROD	DOUBLE PRECISION product of 2 REAL arguments

A.2. Language Restrictions

The last REAL or DOUBLE PRECISION assignment executed in a REAL or DOUBLE PRECISION function must assign the value to that function. The user must check that all REAL and DOUBLE precision functions in the code to be enhanced do not violate this restriction.

The main program must be the first routine in the program to be enhanced. If the program is broken into several files, each file may be enhanced separately. However, the main program must be the first routine in the file in which it is contained. The last FORTRAN statement in any file enhanced must be an END statement.

GRESS supports the use of include statements in the unenhanced program if they have the following syntax:

include 'filename'

The include statement cannot be the last line in the file to be enhanced.

GRESS does not support the use of intermediate scratch files. Circumstances where calculated variables are written to external data storage and then later read and used can cause results to be incorrect. At the point where terms are read in, derivatives are initialized to zero. If the dependency of a response to a parameter is propagated through a variable that is written to scratch and later read in, the derivative will most likely be wrong. We are working on methods to handle this problem; however, at present we are handling it on a case-by-case basis. Usually we are re-writing the program, prior to processing with GRESS, to remove the use of external storage.

GRESS does not support REAL or DOUBLE PRECISION arrays with more than four dimensions that are initialized with read statements. Arrays with up to seven dimensions are supported in all other situations. If an array with five or more dimensions is input via a read statement either the logic will have to be re-written to remove the array from the read statement, or the *SYMG OFF directive should be used to prevent processing of the statement.

APPENDIX B - CHAIN SAMPLE PROBLEM

The CHAIN option calculates the sensitivities of a variable with respect to a user-selected subset of the input data by repeated application of the chain rule. The CHAIN option reports sensitivities 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 flow chart shown in Fig. 2.1 illustrates the processing steps for a GRESS application. A FORTRAN 77 program is input to the GRESS precompiler (SYMG). SYMG creates a new FORTRAN program that when compiled and linked with the GRESS run-time library is capable of calculating derivatives for each floating point assignment statement along with the normally calculated result.

The user inserts application dependent subroutine calls to control the execution. For the CHAIN option the user must identify parameters and results of interest. The derivatives of selected results may be retrieved using the "get" routines (GETGXX, GETNXX), or derivatives and sensitivities may be reported using the PRNTXX routine. A simple FORTRAN program is presented as an example. The processing steps and output information necessary to perform a complete CHAIN application are shown. There are three steps in performing a CHAIN application: 1) precompile with SYMG; 2) compile and link with GRESS run-time library; and 3) execute the enhanced code. The following program, named TEST.FOR (or test.f on unix systems) is used for demonstration.

```
C  GRESS/ADGEN test program
      READ(5,*)B,C,D
      X = B + D
      Y = D**2 + B**2
      R = 7.0*X + D**2
      S = Y**2
      END
```

Assume that the FORTRAN variables B, C, and D are to be treated as independent variables or parameters. We would like FORTRAN to report the sensitivity of variables R and S to the chosen parameters. That is, we would like the first derivatives and sensitivities of variables R and S with respect to B, C, and D to be calculated and reported.

To prepare the code for an CHAIN application, the *CHAIN directive must be inserted into the code prior to enhancement. A call to subroutine AUTOXX must be inserted as the first executable statement (i.e.,after declarations such as common, dimension, equivalence, etc.) to set the upper limit on the number of parameters that will be selected. Subroutine calls should be inserted to identify parameters (DEFIXX) and to report sensitivities (PRNTXX). A description of how to use these routines and others is included Chapter 3. To further control the application the user may choose to insert additional directives to the precompiler.

```
*WSPSIZE 1 000 000
*CHAIN
*DERIV DXDY
            CALL AUTOXX(-1,3)
C  GRESS/ADGEN test program
            READ(5,*)B,C,D
            CALL DEFIXX(B,' B ')
            CALL DEFIXX(C,' C ')
            CALL DEFIXX(D,' D ')
            X = B + D
            Y = D**2 + B**2
            R = 7.0*X + D**2
            CALL PRNTXX(R)
            S = Y**2
            CALL PRNTXX(S)
            END
```

The *WSPSIZE directive controls the amount of memory allocated in words used for propagating derivatives. The default value is probably sufficient for most applications; however, *WSPSIZE is included in this example to aid the user in understanding how it is used. WSPSIZE in this example is set at 1,000,000 words. The *DERIV directive changes the variable used to store the derivative for an assignment statement from the default, DX, to the specified user specified string, DXDY.

Once the code is prepared for precompilation the source code must be associated with logical unit 50. On VAX/VMS systems this is done with an assign statement. On UNIX systems the association can be made by copying the source code to fort.50 or linking (i.e., ln) the source code to fort.50. SYMG writes the enhanced source code to logical unit 7.

The following commands will make logical unit assignments and execute SYMG to enhance TEST.FOR on a VAX/VMS computer.

```
$ASSIGN TEST.FOR for050
$ASSIGN TEST_SG.FOR for007
$RUN SYMG
```

The following commands can be used to execute symg on a UNIX system to enhance the sample program named test.f.

```
>cp test.f fort.50
>symg
>mv fort.7 test_sg.f
```

The data set created during precompilation is the enhanced source program. The FORTRAN 77 compiler and link editor used with the code prior to enhancement are also used to compile and link the enhanced code. The object module for the enhanced code must be linked with the GRESS run-time library. The following commands can be used to compile and link TEST_SG.FOR on a VAX/VMS computer with the GRESS run-time library (EXLIB.OLB).

```
$FOR TEST_SG
$LINK TEST_SG,EXLIB/LIB
```

The following command can be used to compile and link te _sg.f on a UNIX system with the GRESS run-time library (sglib.o and clib.o).

```
>f77 -o test_sg test_sg.f sglib.o clib.o
```

For this example three numbers (2.0, 3.0, 5.0) were entered into a file named fort.5 with a 3E15.5 format. To execute the enhanced version of TEST.FOR (i.e., TEST_SG) on a VAX/VMS computer and report the derivatives and sensitivities of R and S with respect to A, B, and C

```
$ASSIGN/USER_MODE   fort.5 for005
$RUN TEST_SG
```

To execute test_sg on a UNIX system simply enter test_sg at the UNIX prompt.

```
>test_sg
```

The output from the PRNTXX is a report of derivatives and sensitivities written to logical unit 6. The sensitivity report for the test program includes the parameter name but not the response name.

```
GRADIENTS FOR VARIABLE       9428
  7.000000E+00  0.000000E+00  1.700000E+01
SENSITIVITIES FOR VARIABLE       9428
  1.891892E-01  0.000000E+00  1.148649E+00
GRADIENTS FOR VARIABLE       9432
  2.320000E+02  0.000000E+00  5.800000E+02
SENSITIVITIES FOR VARIABLE       9432
  5.517241E-01  0.000000E+00  3.448276E+00
```

The GRADIENT is the first derivative. Gradients and sensitivities are reported when PRNTXX is called. It is the responsibility of the user to keep track of the order in which PRNTXX is called. Many users prefer to use the GETGXX or GETNXX routines to retrieve derivatives so that they can generate their own report, tailored to meet their needs. PRNTXX is useful for a quick look at the derivative and sensitivity values or for debugging.

This example shows how to do a simple program using the CHAIN option. Library routines and precompiler directives can be used to adapt the enhanced code for a more advanced application. The interested user should review the sections on GRESS library routines and directives to the precompiler for helpful suggestions.

APPENDIX C - ADGEN SAMPLE PROBLEM

ADGEN was developed as a GRESS option that provides the capability of automated implementation of the adjoint sensitivity methods into existing FORTRAN 77 models. The flow chart shown in Fig. 2.1 illustrates the processing steps for a GRESS application. A FORTRAN 77 program is input to the GRESS precompiler (SYMG). SYMG creates a new FORTRAN program that when compiled and linked with the GRESS run-time library is capable of calculating derivatives for each floating point assignment statement along with the normally calculated result. For an ADGEN application, these derivatives are either stored in memory or written to a computer disk in a structure that can easily be solved by back substitution. GRESS run-time library routines (i.e., BSOLXX or FBSOLXX) or utility programs (BSOLVE and BREDUCE) are then used to solve the matrix for user selected results of interest. A report of sensitivities and derivatives for selected results with respect to all or part of the input data is generated. Input data is identified either automatically as any data that is entered via a FORTRAN read statement or manually by the user through the insertion of subroutine calls to the GRESS run-time library.

A simple FORTRAN program is presented as an example. The processing steps, data sets created, output information, and utility programs necessary to perform a complete application are shown. There are four steps in performing an ADGEN application: 1) precompile with SYMG; 2) compile and link with GRESS run-time library; 3) execute the enhanced code; and 4) solve the matrix. The following program, named TEST.FOR (or test.f on unix systems) is used for demonstration.

```
C GRESS/ADGEN test program
      READ(5,*)B,C,D
      X = B + D
      Y = D**2 + B**2
      R = 7.0*X + D**2
      S = Y**2
      END
```

Note specifically that FORTRAN variables B, C, and D are input via a READ statement which means they will automatically be treated as independent parameters. Let's assume that we would like FORTRAN variables R and S to be chosen as results of interest. That is, we would like the first derivatives and sensitivities of variables R and S with respect to B, C, and D to be calculated and reported.

To prepare the code for an ADGEN application, subroutine calls must be included to identify the purpose of the run (SETRXX), to select results of interest (POTRXX), and to either solve the matrix (BSOLXX or FBSOLXX), or to clear the matrix buffers (CLEARXX). A description of how to use these routines and others is included Chapter 3.

To further control the application the user may choose to insert directives to the precompiler. For a small code such as TEST.FOR, the directives are unnecessary; however, some have been included in this example to aid the user in understanding how they are used.

```
*COMMENTS ON
*DERIV DXDY
      CALL SETRXX('ADJOINT')
C GRESS/ADGEN test program
      READ(5,*)B,C,D
      X = B + D
      Y = D**2 + B**2
      R = 7.0*X + D**2
      CALL POTRXX(R,' R ')
      S = Y**2
      CALL POTRXX(S,' S ')
      CALL BSOLXX(-1,45,0.001)
      END
```

The first two lines are directives to the precompiler. The *COMMENTS ON directive causes the precompiler to pass any comments beginning with a 'C' or 'c' in column one to the enhanced code. The default is to not include comments in the enhanced code. The *DERIV directive changes the variable used to store the derivative for an assignment statement from the default, DX, to the specified user specified string, DXDY.

Once the code is prepared for precompilation the source code must be associated with logical unit 50. On VAX/VMS systems this is done with an assign statement. On UNIX systems the association can be made by copying the source code to fort.50 or linking (i.e., ln) the source code to fort.50. SYMG writes the enhanced source code to logical unit 7.

The following commands will make logical unit assignments and execute SYMG to enhance TEST.FOR on a VAX/VMS computer.

```
$ASSIGN TEST.FOR for050
$ASSIGN TEST_SG.FOR for007
$RUN SYMG
```

The following commands can be used to execute symg on a UNIX system to enhance the sample program named test.f.

```
>cp test.f fort.50
>symg
>mv fort.7 test_sg.f
```

The data set created during precompilation is the enhanced source program. The FORTRAN 77 compiler and link editor used with the code prior to enhancement are also used to compile and link the enhanced code. The object module for the enhanced code must be linked with the GRESS run-time library. The following commands can be used to compile and link TEST_SG.FOR on a VAX/VMS computer with the GRESS run-time library (SLIB.OLB).

```
$FOR TEST_SG
$LINK TEST_SG,SLIB/LIB
```

The following command can be used to compile and link test_sg.f on a UNIX system with the GRESS run-time library (sglib.o and clib.o).

```
>f77 -o test_sg test_sg.f sglib.o clib.o
```

For this example three numbers (2.0, 3.0, 5.0) were entered into a file named fort.5 with a 3E15.5 format. To execute the enhanced version of TEST.FOR (i.e., TEST_SG) on a VAX/VMS computer and report the derivatives and sensitivities of R and S with respect to B, C, and D

```
$ASSIGN/USER_MODE   fort.5 for005
$RUN TEST_SG
```

To execute test_sg on a UNIX system simply enter test_sg at the UNIX prompt.

```
>test_sg
```

As an alternative to solving the adjoint matrix with BSOLXX (or FBSOLXX) the user can choose to output the adjoint matrix to disk and solve it with the BSOLVE program. To output the adjoint matrix to disk, replace the call to BSOLXX in the sample program with a call to CLEARXX. CLEARXX has no arguments. When the enhanced program is executed, an adjoint matrix will be created.

The two utility programs used to solve the adjoint matrix are BREDUCE and BSOLVE. BREDUCE implements the back reduction algorithm discussed in Reference 1 to create a "reduced" form of the adjoint matrix. The BREDUCE step can only be executed one time and is only needed when working with large models. Since TEST.FOR is a small program we will skip the BREDUCE step and proceed directly to BSOLVE. To solve the matrix enter the following.

```
$RUN BSOLVE
```

On UNIX systems simply enter bsolve.

```
>bsolve
```

The BSOLVE program will request a value for the smallest sensitivity to report (i.e., CUTOFF). In practice sensitivities that are less than 0.01 are of little interest. Entering a value of 0.0 for CUTOFF will result in all sensitivities being reported.

The output from the BSOLVE program is a report of derivatives and sensitivities written to logical unit 45. The sensitivity report for the test program includes both response and parameter names.

```
Row number for response = 6  Number of parameters  =  3

RESPONSE   1      R =    7.400000D+01

   ROW  NAME  DERIVATIVE       SENSITIVITY
    1    B    7.00000E+00      1.89189E-01
    2    C    0.00000E+00      0.00000E+00
    3    D    1.70000E+01      1.14865E+00

Row number for response =  7  Number of parameters  =  3

RESPONSE   2      S  =  8.410000D+02


   ROW  NAME  DERIVATIVE       SENSITIVITY
    1    B    2.32000E+02      5.51724E-01
    2    C    0.00000E+00      0.00000E+00
    3    D    5.80000E+02      3.44828E+00
```

This example shows how to do a simple program using the ADGEN option. Library routines and precompiler directives can be used to tailor the enhanced code for a more advanced application. The interested user should review the sections on GRESS library routines and directives to the precompiler for helpful suggestions.

APPENDIX D - THE GENSUB OPTION

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 (responses) with respect to independent variables (parameters). GENSUB allows the processing of program units as small as a do loop or as large as an entire program for derivative calculation. GENSUB will use either forward or reverse chaining depending on which is most efficient for the given problem.

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 with one of the parameter declaration routines GENPXX, GENAPXX, or GENDPXX. Parameters 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 parameters on entry to the subroutine.

Run-time routine GENRESXX is used to identify responses. Responses 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 sub-section of the program. At the end of the sub-section (e.g., function or subroutine) being processed with the GENSUB option, the user should insert a call to subroutine CHAINGG with the result array as an argument. CHAINGG 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. CHAINGG returns the amount of memory used processing the derivatives in the second and third arguments. The third argument is the amount of memory required in four byte words that was automatically set to zero. The second argument is the amount of memory in bytes that was not automatically set to zero.

The ALLOCGG routine can be used to pre-allocate memory for the section of code being processed with GENSUB. Depending on the code being processed, 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 request using ALLOCGG. Since the amount of memory used to process a section of code can vary, the user must be careful when using ALLOCGG. However, on some operating systems even over estimating the amount of memory to allocate with ALLOCGG is more efficient than not pre-allocating memory.

Upon return from CHAINGG the memory allocated for storing and propagating derivatives is released.

GENSUB can be used to process all or part of a program; however, the first step in a GENSUB application is to separate the section of code to be processed from the rest of the program. The first example shows the calculation of derivatives for the output from subroutine SUB1 (i.e., B) with respect to the input (i.e., X(1), X(2), X(3), and X(4)). To prepare the subroutine SUB1 for precompilation, use the *GENSUB directive. Subroutine calls to declare parameters and responses can be inserted before or after enhancement. Also, call to CHAINGG can be inserted before or after enhancement. In the example, derivatives of responses with respect to parameters are output with a print statement. Note that the array RESULT was dimensioned one-by three to hold the derivatives of one result with respect to three declared parameters.

```
*gensub
      SUBROUTINE SUB1(AA,BB,CC)
      DIMENSION X(4),RESULT(1,3)
      EQUIVALENCE(FX,RX)
C
C Declare AA, BB, and CC to be parameters
C
      call genpxx(AA)
      call genpxx(BB)
      call genpxx(DD)
C
      CC=3.
      FX=AA/BB  + CC*DD
      RANN = 1.0
      A = RANN * fX
      B = ATAN(ABS(A/(A+3.1))) - SIN(SQRT(A/DD))
      C = ALOG(ABS(B)) + EXP(B/(B+2.5))
      B = A*B/C + ALOG10(ABS((C+A)/B)) + COS(ABS(C)/C**2)
      B = A**2/ABS(B)**1.02 * B
      call genresxx(B)
      call chaingg(result,imem,izero)


      print*,' memory allocated (imem) =',imem
      print*,' memory allocated pre-set to zero (izero) =',izero
      print*,(result(1,ij),ij=1,3)
      RETURN
      END
```

Subroutine SUB1 is ready to be enhanced with the GRESS precompiler. The following commands will make logical unit assignments and execute SYMG to enhance SUB1.FOR on a VAX/VMS computer.

```
$ASSIGN SUB1.FOR for050
$ASSIGN SUB1_SG.FOR for007
$RUN SYMG
```

The following commands can be used to execute symg on a UNIX system to enhance the sample program named sub1.f.

```
>cp sub1.f fort.50
>symg
>mv fort.7 sub1_sg.f
```

The data set created during precompilation is the enhanced subroutine, SUB1_SG.FOR (or sub1_sg.f). The following main program (MAIN.FOR or main.f) will be used to call subroutine SUB1.

```
      PROGRAM MAIN
C
      AA = 1.0
      BB = 2.0
      DD = 4.0
      CALL SUB1(AA,BB,DD)
      END
```

The FORTRAN 77 compiler and link editor used with the code prior to enhancement are also used to compile and link the enhanced code. The object module for the enhanced code must be linked with the GRESS GENSUB library. The following commands can be used to compile and link SUB1_SG.FOR on a VAX/VMS computer with the GRESS GENSUB library (genlib.obj).

```
$FOR SUB1_SG
$FOR MAIN
$LINK MAIN,SUB1_SG,GENSUB
```

The following command can be used to compile and link test_sg.f on a UNIX system with the GRESS GENSUB library (genlib.o).

```
>f77 -o sub1_sg main.f sub1_sg.f genlib.o
```

The RUN command is used to execute the enhanced version of SUB1 on a VAX/VMS system.

```
$RUN SUB1_SG
```

On unix systems, simply enter the program name at the system prompt.

```
>sub1_sg
```

With the GENSUB option the output is determined by the user. In this example the amount of memory used during the execution of SUB1 and the derivatives of B with respect to the declared parameters are output with print statements.

```
memory allocated (imem) =  288
memory allocated pre-set to zero (izero) =  64
12.13453484     -6.067267418     71.32409668
```

On subsequent executions of the program, the user may choose to use the values printed out for imem and izero to pre-allocate the memory required for subroutine sub1_sg. Memory can be pre-allocated using the ALLOCGG library routine. The following example shows how ALLOCGG could be used with subroutine SUB1_SG. The call to ALLOCGG can be inserted before or after enhancement.

```
*gensub
      SUBROUTINE SUB1(AA,BB,DD)
      DIMENSION X(4),RESULT(1,3)
      EQUIVALENCE(FX,RX)
C  Pre-allocate memory
      call allocgg(288,64)
C
C Declare AA, BB, and CC to be parameters
C
      call genpxx(AA)
      call genpxx(BB)
      call genpxx(DD)
C
      CC=3.
      FX=AA/BB  + CC*DD
      RANN = 1.0
      A = RANN * fX
      B = ATAN(ABS(A/(A+3.1))) - SIN(SQRT(A/DD))
      C = ALOG(ABS(B)) + EXP(B/(B+2.5))
      B = A*B/C + ALOG10(ABS((C+A)/B)) + COS(ABS(C)/C**2)
      B = A**2/ABS(B)**1.02 * B
      call genresxx(B)
      call chaingg(result,imem,izero)
      print*,' memory allocated (imem) =',imem
```

```
print*,' memory allocated pre-set to zero (izero) =',izero
print*,(result(1,ij),ij=1,3)
RETURN
END
```

When using ALLOCGG the user must be careful to ensure that enough memory is pre-allocated. For small subroutines on most systems ALLOCGG is not necessary. For large segments of code the user may see significant savings in execution time and memory utilization; however, the impact of ALLOCGG is machine and problem dependent. I would recommend using ALLOCGG whenever you know the amount of memory required. Don't use ALLOCGG the first time you run a new routine or problem.

GENSUB can also be used to process an entire program. In most cases CHAIN or ADGEN would be more efficient; however, if a code is iterative and you can solve for derivatives between iterations, GENSUB with its dynamic allocation may have some benefits. The following program is set up for a GENSUB application.

```
*gensub
*comments on
C           GRESS/GENSUB SAMPLE PROBLEM 2
      PROGRAM JMA
      DIMENSION X(4),F(4,4,4),RS(1000)
      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)
      PRINT*,'X(I),I=1,4)',X
C
C  Define X to be an array of parameters
C
      call genapxx(x,4)
      LOOP1 = 4
      LSUMO = 100
      D = 0.0
      DO 1 I = 1,LSUMO
      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
      FS = FS + F(L,K,J)
2     CONTINUE
```

```
      BFS = B + FS
      RS(I) = BFS
      D = D + BFS
C
C DECLARE POTENTIAL RESPONSES
C
1     CONTINUE
      WRITE(6,9) D
9     FORMAT(1H ,'D',1PE16.8)
8     FORMAT(1H ,'A,B'/(1H ,I3,1P,4E16.8))
C  D is declared a response
      call genresxx(D)
C  The CHAINREV routine implements chain rule
C  in reverse mode with derivatives returned in DX.
C
      call chainrev(dx,imalloc,icalloc)
C
C  IMALLOC is memory allocated
C  ICALLOC is memory allocated and preset to zero
      print*,' malloc =',imalloc,' calloc =',icalloc
      write(6,*)(dx(j),j=1,4)
      STOP
      END
      SUBROUTINE SUB1(I,A,B,X)
      DIMENSION X(4)
      FX = X(1)/X(2) + X(3)*X(4)
      RANN = 0.0
      CALL GETRAN(RANN)
      A = RANN * FX
      B = ATAN(ABS(A/(A+3.1))) - SIN(SQRT(A/X(4)))
      C = ALOG(ABS(B)) + EXP(B/(B+2.5))
      B = A*B/C + ALOG10(ABS((C+A)/B)) + COS(ABS(C)/C**2)
      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))
      FXR = FXR*X(4) + MAX(X(1),X(2),X(3),X(4))
      FXR = FXR - MIN(X(1),X(2),X(3),X(4))
      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
```

The output from this sample problem includes the derivatives of D with respect to the elements in array X, as well as the amount of memory required.

```
total malloc = 2548816      total calloc = 1237
1067380.   -19961.63    401086.7    149175.0
```

Though not shown, it would be possible to solve for derivatives between iterations in the previous example, thus reducing overall memory requirements. The X array and D are the only variables that need be retained between iterations.

APPENDIX E - IMPLEMENTATION NOTES

My experiences in implementing GRESS on the various computers and operating systems is provided in this section. Though computer scientists are working very hard at defining standards for languages such as C and FORTRAN, the interfacing between those languages is far from standardized. Most of the differences between the various implementations of GRESS are related to the interfacing between C and FORTRAN. Provided is information on how to compile and link the GRESS precompiler and run-time libraries.

E.1. VAX/VMS

GRESS was developed in the VAX/VMS environment. Most of the major program options have been tested by application to programs used in the nuclear industry and elsewhere. The following instructions will create SYMG.EXE.

```
$FOR SYMG
$LIN SYMG
```

The GRESS Run-Time Library can easily be created as an object library.

```
$FOR SGLIB
$CC CLIB
$LIB/CREATE SLIB SGLIB,CLIB
```

An enhanced program named TEST_SG.FOR should be compiled and then linked with the object library.

```
$FOR TEST_SG
$DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCRTL.OLB*
$LINK TEST_SG,SLIB/LIB
```

The executable image, TEST_SG.EXE, is now ready for execution.

The GRESS GENSUB option requires only GENLIB.C. To compile GENLIB.C enter the following.

```
$CC GENLIB.C
```

If TEST_SG.FOR is precompiled with the *GENSUB directive, then the following commands will compile and link TEST_SG.FOR and create an executable program, TEST_SG.EXE.

*Define lnk$library only needs to be done once during a session.

```
$FOR TEST_SG
$DEFINE LNK$LIBRARY SYS$LIBRARY:VAXCRTL.OLB
$LINK TEST_SG,GENLIB
```

The VAX/VMS G_FLOAT option does not work under all circumstances with GRESS. ADGEN and CHAIN will work unless the value of a derivative falls outside the range E-38 to E+38, approximately. The run-time libraries must be compiled with the G_float option. Since GRESS propagates derivatives in single precision and G_FLOAT allows double precision exponents in the range E-300 to E+300 it is possible for a derivative to be too big or too small for GRESS. In most circumstances this does not occur.

E.2. IBM/AIX

GRESS was implemented on an IBM/6000 RISC Work Station. Most major program options have been tested. The following instructions were used to create a precompiler named symg, and to compile the run-time libraries.

```
$xlf -o symg symg.f
$xlf -c sglib.f
$cc -c clib.c
$cc -c genlib.c
```

For a CHAIN or ADGEN application, an enhanced program named test_sg.f would be compiled and linked with sglib.o and clib.o to create an executable file named sg.

```
$xlf -o sg test_sg.f sglib.o clib.o
```

For a GENSUB application, test_sg would be compiled and linked with genlib.o to create an executable file named sg.

```
$xlf -o sg test_sg.f genlib.o
```

E.3. VAX/ULTRIX

GRESS was implemented on a VAX with ULTRIX operating system. ULTRIX is a UNIX operating system. This version has undergone very limited testing. Most major program options were tested on small sample problems. The following instructions were used to create a precompiler named symg, and to compile the run-time libraries.

```
$f77 -o symg symg.f
$f77 -c sglib.f
$cc -c clib.c
$cc -c genlib.c
```

For a CHAIN or ADGEN application, an enhanced program named test_sg.f would be compiled and linked with sglib.o and clib.o to create an executable file named sg.

```
$f77 -o sg test_sg.f sglib.o clib.o
```

For a GENSUB application, test_sg would be compiled and linked with genlib.o to create an executable file named sg.

```
$f77 -o sg test_sg.f genlib.o
```

Due to system limitations the amount of memory required by the enhanced code was severely restricted. Following the instructions in Chapter 4, parameters IWORKSZ, LSTOT, LTOT, MAXR, and ITAB were substantially reduced. A significant reduction in those parameters creates a version of GRESS that is suitable for small problems (as measured in execution time and memory requirements). CHAIN option should work for larger problems; however, ADGEN is severely limited.

E.4. SUN

GRESS was implemented on a SUN SPARC Station 1+. This version has undergone very limited testing. Most major program options were tested on small sample problems. The following instructions were used to create a precompiler named symg, and to compile the run-time libraries.

```
$f77 -o symg symg.f
$f77 -c sglib.f
$f77 -c flib.f
$cc -c clib.c
$cc -c genlib.c
```

For a CHAIN or ADGEN application, an enhanced program named test_sg.f would be compiled and linked with sglib.o and clib.o to create an executable file named sg.

```
$f77 -o sg test_sg.f sglib.o clib.o
```

For a GENSUB application, test_sg would be compiled and linked with genlib.o to create an executable file named sg.

```
$f77 -o sg test_sg.f flib.o genlib.o
```

Due to system limitations the amount of memory required by the enhanced code was severely restricted. Following the instructions in Chapter 4, parameters IWORKSZ, LSTOT, LTOT, MAXR, and ITAB were substantially reduced. A significant reduction in those parameters creates a version of GRESS that is suitable for small problems (as measured in execution time and memory requirements). CHAIN option should work for larger problems; however, ADGEN is severely limited.

E.5. HEWLETT-PACKARD

GRESS was implemented on a HP 9000 Work Station. This version has undergone very limited testing. Most major program options were tested on small sample problems. The following instructions were used to create a precompiler named symg, and to compile the run-time libraries.

```
$f77 -o symg symg.f
$f77 -c sglib.f
$cc -c clib.c
$cc -c genlib.c
```

For a CHAIN or ADGEN application, an enhanced program named test_sg.f would be compiled and linked with sglib.o and clib.o to create an executable file named sg.

```
$f77 -o sg test_sg.f sglib.o clib.o
```

For a GENSUB application, test_sg would be compiled and linked with genlib.o to create an executable file named sg.

```
$f77 -o sg test_sg.f flib.o genlib.o
```

On the HP, SYMG writes the enhanced code to logical unit 70.

INTERNAL DISTRIBUTION

1.	B. R. Appleton	21.	M. D. Morris
2.	S. A. Bartell	22-26.	E. M. Oblow
3.	J. M. Bownds	27.	F. G. Pin
4.	R. W. Brockett (Consultant)	28.	S. F. Railsback
5.	J. J. Dongarra	29.	R. J. Raridon
6.	J. J. Dorning (Consultant)	30.	C. C. Travis
7.	M. B. Emmett	31.	R C. Ward
8.	D. M. Hetrick	32.	R. M. Westfall
9-13.	J. E. Horwedel	33-37.	B. A. Worley
14.	D. Ingersoll	38-42.	R. Q. Wright
15.	H. I. Jager	43.	M. Yambert
16.	J. E. Leiss (Consultant)	44-45.	Laboratory Records Dept.
17.	P. Kanciruk	46.	Laboratory Records, ORNL-RC
18.	R. McLean	47.	Document Reference Section
19.	T. Mitchell	48.	Central Research Library
20.	N. Moray (Consultant)	49.	ORNL Patent Office

EXTERNAL DISTRIBUTION

50. Office of the Assistant Manager for Energy Research and Development, DOE Field Office, Oak Ridge, P.O. Box 2008, Oak Ridge, TN 37831
51. Rod Bain, Department of Chemical Engineering, University of Wisconsin, Madison, WI 53706
52. Jean-Francois M. Barthelemy, Interdisciplinary Research Office, Structural Dynamics Division, M/S 246, NASA/Langley Research Center, Hampton, VA 23665-5225
53. Laura McDowell-Boyer, Grand Junction Office, P.O. Box 2567, Grand Junction, CO 81502
54. J. R. Cook, Interim Waste Technology Division, Savannah River Laboratory, P.O. Box 616, Aiken, SC 29802
55. Dr. Richard F. Deckro, Portland State University, Engineering Management Program, P.O. Box 751, Portland, OR 97207-0751
56. J. E. Dennis, Jr., Rice University, Department of Mathematical Sciences, P.O. Box 1892, Houston, TX 77251
57. G. F. Corliss, Marquette University, Department of Mathematics, Milwaukee, WI 53233
58. E. G. Dupnick, ECON, Inc., 3020 Hamakeer CT B 105, Fairfax, VA 22031
59. David M. Gay, 35 Livingston Ave., New Providence, NJ 07974-2219
60. Jean Charles Gilbert, INRIA, Domaine de Voluceau, B P 105 78153, Le Chesnay, France
61. Andreas Griewank, Argonne National Laboratory, Division of Mathematics & Computer Science, 9700 S. Cass Ave., Argonne, IL 60439
62. Laura Hall, Interdisciplinary Research Office, Structural Dynamics Division, M/S 246, NASA/Langley Research Center, Hampton, VA 23665-5225
63. W. V. Harper, Resource International, 281 Enterprise Dr., Westerville, Ohio 43081
64. David Juedes, Iowa State University, Department of Computer Science, 4317 Lincoln Swing #34, Ames, IA 50010
65. John M. Kallfelz, Paul Scherrer Institut, CH-5232 Villigen PSI, Switzerland

66. Koichi Kubota, Keio University, Department of Admin. Engineering, 3-14-1 Kohoku-ku Hiyoshi, Yokohama 223, Japan
67. Charles L. Lawson, 1340 Marianna Rd., Pasadena, CA 91105
68. J. Dan Layne, P.O. Box 1036, Littleton, CO 80160
69. Weldon A. Lodwick, Department of Mathematics, Computational Mathematics Group, Campus Box 170, P.O. Box 173364, Denver, Colorado, 80217-3364
70. D. W. Muir, IAEA Nuclear Data Section, P.O. Box 200, A-1400 Vienna, Austria
71. Ionel M. Navon, Florida State University, Department of Mathematics, Love Bldg., Rm. 111, Tallahassee, FL 32306-3027
72. T. A. Parish, Nuclear Engineering Department, Zachry Building, Texas A&M University, College Station, TX 77843
73. L. B. Rall, University of Wisconsin, Department of Mathematics, 480 Lincoln Drive, Madison, WI 53706
74. Marcela L. Rosenblun, Rice University, Dept. of Mathematical Sciences, P.O. Box 1892, Houston, TX 77251
75. G. R. Shubin, Numerical Analysis, Org. G-6412,M/S 7L-21, Boeing Computer Services, P.O. Box 24346, Seattle, WA 98124-0346
76. Edgar Souli, Inst. de Res. Tech. & Indus. Dev., Division d'Etudes de Separation, Cen. d'Nucleaires de Saclay, F-91191 Gif-Sur-Yvette Cedex, France
77. Leigh Tesfatsion, Iowa State University, Department of Economics and Mathematics, Heady Hall, Ames, IA 50010-1070
78. W. C. Thacker, NOAA/AOML, 4301 Rickenbacker Causeway, Miami, FL 33149
79. J. Thames, Digital Calculus Corp., 5406 Via Del Valle, Torrance, CA 90505
80. A. L. Tits, Electrical Engineering Department, University of Maryland, College Park, MD 20742
81. A. D. Yu, Interim Waste Technology Division, Savannah River Laboratory, P.O. Box 616, Aiken, SC 2980
82-91. Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831 (10)

END

DATE
FILMED
2/03/92