FORTDIFF: A Set of Subroutines for Fortran-to-Fortran Differentiation of Programs

R. Baker Kearfott*
and Shiying Ning
University of Southwestern Louisiana

Abstract

The algorithm contains a moderately-sized system of Fortran-90 subroutines, along with a driver program. The output of this system is a Fortran-77 program for evaluating the derivative of a user-specified function $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$, where m and n are arbitrary. The user defines f as a Fortran-90 subroutine, with certain syntax restrictions. The statements in the resulting Fortran-77 program are completely unrolled.

A program that produces a LaTeX file describing f and its derivatives is also included.

The system contains technology from a prototyping environment for nonlinear equations and nonlinear optimization algorithms. A symbolic form of automatic differentiation, implemented with operator overloading, is used.

Categories and Subject Descriptors: G.1.5[**Numerical Analysis**]: Roots of Nonlinear Equations, G.1.6: Optimization, G.4[**Mathematical Software**], D.3.4[**Programming Languages**]: Processors, I.1.2[**Algebraic Manipulation**]: Algebraic algorithms

General Terms: Tools for Scientific Computing

Additional Key Words and Phrases: Fortran 90, automatic differentiation, FORTRAN-77, symbolic computation

*This work was supported in part by National Science Foundation grant CCR-9203730.

1 Introduction

Computation of derivatives is an important auxiliary task in codes for numerical optimization, solution of nonlinear algebraic systems, and parameter fitting. Programming large Jacobian matrices by hand is costly and error-prone. For this reason, diverse techniques have been used to avoid this process. These include finite-difference approximations, quasi-Newton methods with secant updates as in [3] and elsewhere, symbolic computations and, more recently, automatic differentiation. Finite differences, though commonly used in the past, can suffer from truncation and roundoff errors and can involve many function evaluations. In quasi-Newton methods, not all derivatives are actually approximated, and convergence properties are somewhat less desirable. Modern commercial packages for symbolic manipulation, such as Reduce, Macsyma, or Mathematica, allow one to input a function in symbolic form, and to output its derivatives as a Fortran or C program. However, such symbolic manipulation can result in "expression swell," in which the output, not in lowest terms, can take many times the amount of space used to describe the original function; numerical properties of evaluation of such expressions are in doubt. Additionally, unless the symbolic package is integrated into a custom environment, it may be inconvenient to express the original function, given as a set of subroutines in C or Fortran, as an expression in the form required by the symbolic package. The third alternative, automatic differentiation, reviewed in [7], obtains derivatives without truncation or cancellation errors, and without disadvantages of traditional symbolic differentiation.

Here, we present an algorithm, based on automatic differentiation technology, that takes, as input, a Fortran 90 representation of a function and produces, as output, a FORTRAN 77 program that evaluates the function, its derivative, or both. The technology is based on the system described in [9]: Operator overloading is first used to produce an internal representation of the function. That internal representation can then be differentiated according to a list of simple rules, resulting in an internal representation of the same form. Such internal representations can be translated into TEX, Fortran, or another language, using a list of relatively simple rules. Alternately, generic routines can be used at runtime to interpret the internal representation, and thus obtain numerical values of the function or derivatives in any desired data format.

Other packages have dealt with differentiation of Fortran programs. For instance, ADIFOR [2] and [1], written in Fortran 77, is the result of an ambitious project to differentiate Fortran 77 programs. That package is much more sophisticated than this one, but is for the most part built on slightly different principles. The commercial package and associated language AMPL [5] also incorporates a similar underlying technology to this package. The main advantages of this package are its small size, simplicity and comprehensibility, and ease of installation.

2 Package Contents and Use

Production of a subroutine that evaluates derivatives proceeds according to the following steps.

1. Write a Fortran 90 program, using the syntax of §2.2.
2. Compile and run the program from step 1, using modules from the package, to produce an internal representation of the operations required to evaluate the function.
3. Run a program to differentiate the internal representation.
4. Run a program to either output the derivative as a FORTRAN 77 program or as a LaTeX file.

The package is structured according to these tasks.

2.1 Structure and components of the package

The system, supporting interval arithmetic, does so with the interval arithmetic package INTLIB [8], a TOMS algorithm. Not including INTLIB, the package consists of fifteen Fortran 90 files, totalling roughly 200K, along with a configuration file, example output, a user's guide and a make file. We describe these files according to task here.

The sample driver routine RUN_FORTRAN_DIFFERENTIATION runs the entire sequence of steps 2–4 for generating a FORTRAN 77 routine to evaluate derivatives. It calls the sample routine ORIGINAL_FUNCTION, that defines the function to be differentiated. In fact, the routine ORIGINAL_FUNCTION can be used as a template.

Figure 1 illustrates the module used for creation of the internal representation, termed the *code list*. Figure 2 illustrates the module used for generating the derivative from the internal representation. Additionally,

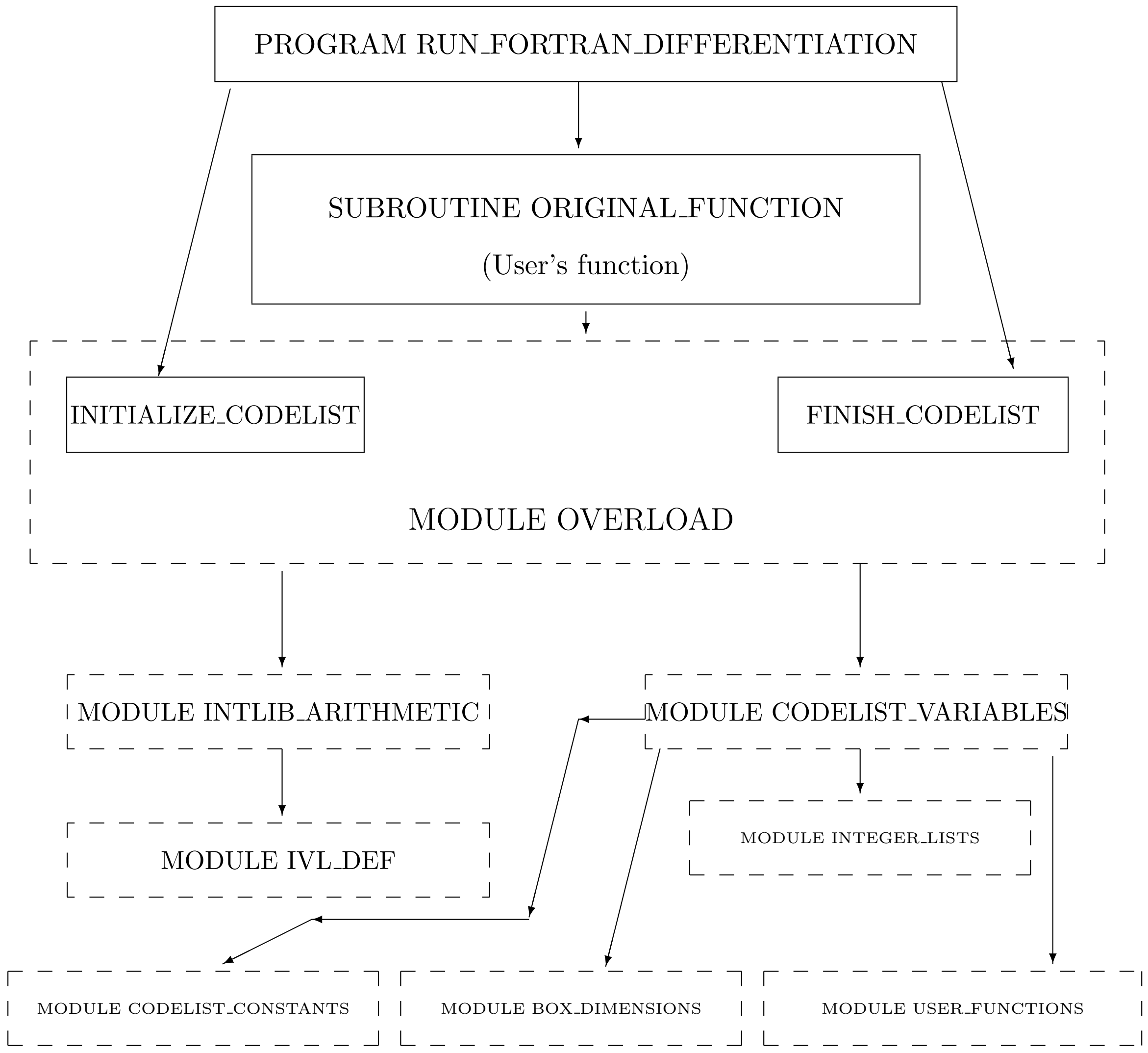

Figure 1: Components for generating the internal representation

PROGRAM PRINT_FUNCTION, using module GRADIENT_VARIABLES, is compiled separately to produce a LaTeX representation of either the function or the derivative. This and the programs and modules account for all of the Fortran 90 source files in the package. Finally, a configuration file OVERLOAD.CFG and a PostScript copy of the user's guide for the system in [9] are included.

The modules should be compiled according to the dependencies shown. A Unix makefile is supplied with the package.

2.2 Syntax and allowed operations

The package supports the elementary operations, including exponentiation between all admissible data types, as well as the standard functions SIN, COS, TAN, ASIN, ACOS, ATAN, COSH, SINH, EXP, LOG, and SQRT. Note that ABS, ATAN2, TANH and LOG10 are *not* presently supported. However, the user can define an arbitrary number of univariate functions, giving the functions in subroutine USR and derivatives in subroutine USRD.

The following rules are used for writing the Fortran 90 program defining the function components.

1. The independent variables and intermediate quantities depending on the input variables should be declared to be type CDLVAR ("CoDe List VARiable"). Such variables may occur on both sides of assignment statements and in arithmetic expressions obeying the rules of Fortran 90. They may occur as arguments to the elementary functions listed above.

2. Variables defining the function must be of the form F(1), . . . , F(N), and these variables must be declared as left-hand-side variables (type CDLLHS). Each of them may only occur once in the program, on the left side of an assignment statement. These variables are assumed to be assigned in the order F(1), F(2), . . . , F(n).

3. Variables used as independent variables must be "initialized" by passing them as arguments to the subroutine INITIALIZE_CODELIST at the beginning of the program. All independent variables must be in a single array, but the name of the array is arbitrary. The dimension of this array must be exactly the number of these variables.

4. Conditionals (e.g. IF-THEN-ELSE or CASE statements) may not be used, unless the conditions tested are constant. However, most non-

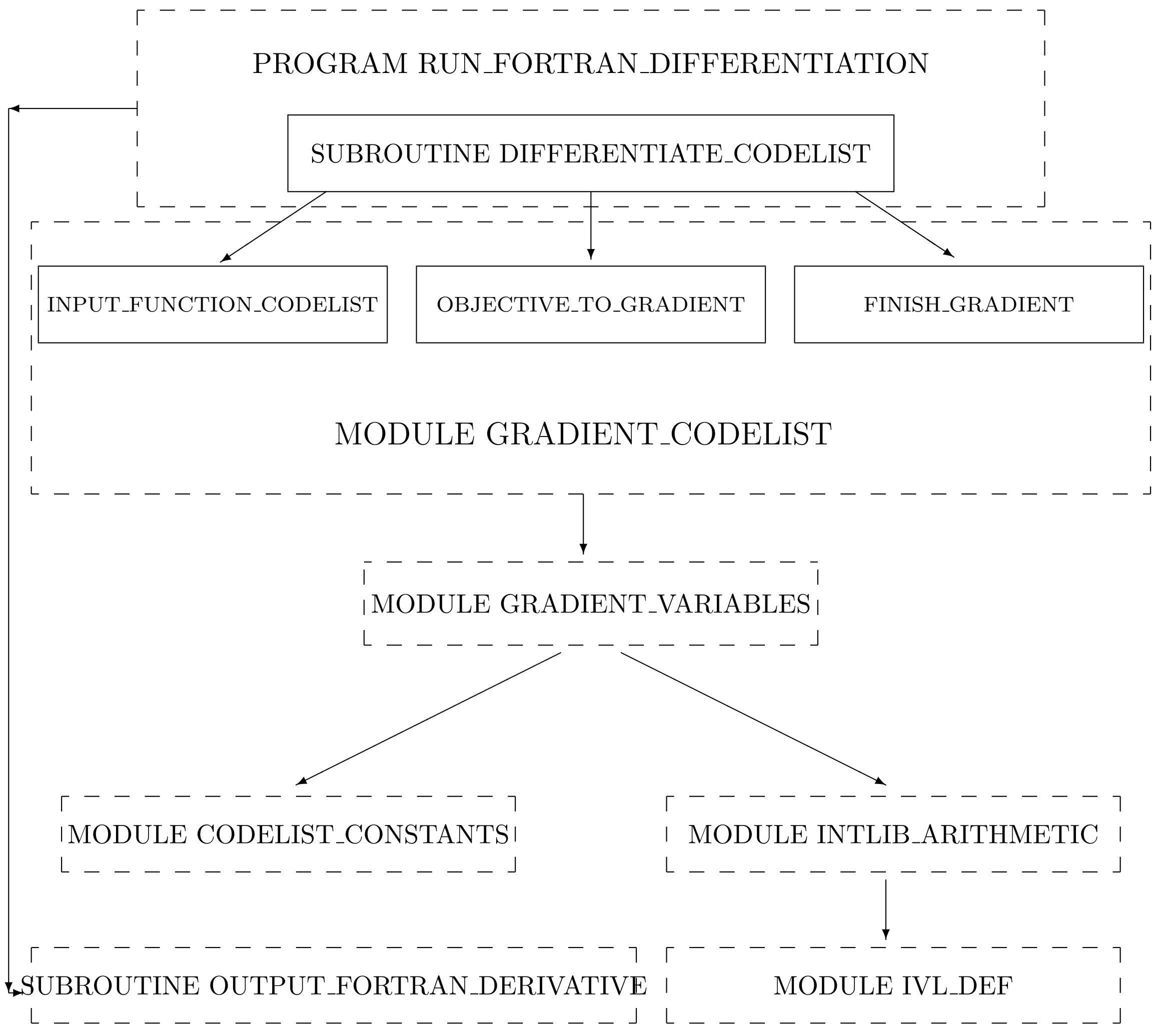
PROGRAM RUN_FORTRAN_DIFFERENTIATION
SUBROUTINE DIFFERENTIATE_CODELIST
INPUT_FUNCTION_CODELIST
OBJECTIVE_TO_GRADIENT
FINISH_GRADIENT
MODULE GRADIENT_CODELIST
MODULE GRADIENT_VARIABLES
MODULE CODELIST_CONSTANTS
MODULE INTLIB_ARITHMETIC
SUBROUTINE OUTPUT_FORTRAN_DERIVATIVE
MODULE IVL_DEF

Figure 2: Components for generating a derivative representation

conditional loops and other non-conditional constructs following Fortran 90 syntax are allowable.

5. Conditional branches may be programmed using the branch function χ, described in [9], with an example in §2.4 below.

6. Double precision, integer, and interval arithmetic may be used within this program, as well as mixed-mode operations with independent variables and variables of type CDLVAR (intermediate variables).

7. Use of IMPLICIT NONE is strongly recommended. Any variable that depends on the independent variables (as specified with INITIALIZE_CODELIST) should be declared to be of type CDLVAR.

8. The function may be defined through a heirarchy of subroutines, as long as each subroutine contains an appropriate USE OVERLOAD statement.

9. The last program statement should be a call to the subroutine FINISH_CODELIST.

Details are given in the user's guide for the system described in [9], distributed with this algorithm.

2.3 An example

An example of a properly constructed program appears in figure 3. In this figure, the symbol OUTPUT_FILE_NAME is intrinsic to the system, and defines the storage name of the internal representation. (The module OVERLOAD defines how the expressions are translated into the internal representation.) If this subroutine is called, followed by a call to OBJECTIVE_TO_GRADIENT as in the sample driver RUN_FORTRAN_DIFFERENTIATION, the output will be the FORTRAN 77 program pictured in figure 4. If the program PRINT_FUNCTION_GRADIENT is then run, the LaTeX file in figure 5 is produced. Running this file produces the contents of figure 6. In figure 6, the function components and their derivatives are indexed in the order

$$(f_1, \partial f_1/\partial x_1, \partial f_1/\partial x_2, f_2, \partial f_2/\partial x_1, \partial f_2/\partial x_2),$$

```
!  This is a test problem for Fortran differentiation.

SUBROUTINE ORIGINAL_FUNCTION(FILE_NAME)

  USE OVERLOAD

  CHARACTER (LEN=20):: FILE_NAME

      TYPE(CDLVAR), DIMENSION(2) :: X
      TYPE(CDLLHS), DIMENSION(2) :: F

      OUTPUT_FILE_NAME=FILE_NAME

      CALL INITIALIZE_CODELIST(X)

  F(1) = 4*X(1)**3 - 3*X(1) - X(2)
  F(2) = X(1)**2 - X(2)

      CALL FINISH_CODELIST

END SUBROUTINE ORIGINAL_FUNCTION
```

Figure 3: An example of Fortran 90 source for the function

the same order as their appearance in the internal representation. Figure 6 gives constants is as floating point numbers, although the internal representation is as intervals. A switch in the routines PRINT_FUNCTION and PRINT_FUNCTION_GRADIENT allows printouts in either format.

2.4 Example of programming conditional branches

Conditional branches, although necessary for representing many functions, cannot be implemented symbolically in a straightforward way when operator overloading is used for parsing, when interval arithmetic is used, or when we wish to differentiate with respect to one or more variables used in branching. For this reason, this package implements branching with a branch function

$$\mathrm{CHI}(x_s, x_q, x_r) = \begin{cases} x_q & \text{if } x_s < 0, \\ x_r & \text{if } x_s \geq 0. \end{cases}$$

```
      SUBROUTINE FG(X,F,G)

      DOUBLE PRECISION X( 2 )
      DOUBLE PRECISION F( 2 )
      DOUBLE PRECISION G( 2 , 2 )

      DOUBLE PRECISION Y( 17 )

     DO 10 I = 1,       2
         Y(I) = X(I)
  10 CONTINUE

      Y( 3 ) = Y( 1 ) ** 3
     Y(       4) =   0.40000000000000000000D+01* Y(       3)
     Y(       5) =   0.30000000000000000000D+01* Y(       1)
      Y( 6 ) = Y( 4 ) - Y( 5 )
      Y( 7 ) = Y( 6 ) - Y( 2 )
      Y( 8 ) = Y( 1 ) ** 2
      Y( 9 ) = Y( 8 ) - Y( 2 )
      Y( 10 ) = Y( 1 ) ** 2
     Y(      11) =   0.30000000000000000000D+01* Y(      10)
     Y(      12) =   0.40000000000000000000D+01* Y(      11)
     Y(      13) =  0.30000000000000000000D+01
      Y( 14 ) = Y( 12 ) - Y( 13 )
     Y(      15) =   0.20000000000000000000D+01* Y(       1)
     Y(      16) = -0.10000000000000000000D+01
     Y(      17) = -0.10000000000000000000D+01

      F( 1 ) = Y( 7 )
      F( 2 ) = Y( 9 )

      G( 1 , 1 ) = Y( 14 )
      G( 1 , 2 ) = Y( 16 )
      G( 2 , 1 ) = Y( 15 )
      G( 2 , 2 ) = Y( 17 )

      END
```

Figure 4: The FORTRAN 77 derivative program corresponding to figure 3

```
 %THE EXPRESSION CORRESPONDING TO CODE LIST FILE:FDTMPG.CDL

 %LaTeX:

 \documentstyle{article}
 \begin{document}
 \begin{eqnarray*}
 f_{     1} &=&\left(\left( ( 4.0    )
\left(x_{ 1}^ 3\right)\right)
         - \left( ( 3.0    )
x_{ 1}\right)\right) - x_{ 2}\\
 \frac{\partial f_{     1}}{\partial x_{     1}} &=&\left( ( 4.0    )
\left( ( 3.0    )
\left(x_{ 1}
        ^ 2\right)\right)\right) -  ( 3.0    )
\\
 \frac{\partial f_{     1}}{\partial x_{     2}} &=& (-1.0    )
\\
 f_{     4} &=&\left(x_ 1^2\right) - x_{ 2}\\
 \frac{\partial f_{     4}}{\partial x_{     1}} &=& ( 2.0    )
x_{ 1}\\
 \frac{\partial f_{     4}}{\partial x_{     2}} &=& (-1.0    )
\\
 \end{eqnarray*}
 \end{document}
```

Figure 5: The LaTeX file for the derivatives, corresponding to figure 3

$$\begin{aligned}
f_1 &= \left(\left((4.0)\left(x_1^3\right)\right) - ((3.0)x_1)\right) - x_2 \\
\frac{\partial f_1}{\partial x_1} &= \left((4.0)\left((3.0)\left(x_1^2\right)\right)\right) - (3.0) \\
\frac{\partial f_1}{\partial x_2} &= (-1.0) \\
f_4 &= \left(x_1^2\right) - x_2 \\
\frac{\partial f_4}{\partial x_1} &= (2.0)x_1 \\
\frac{\partial f_4}{\partial x_2} &= (-1.0)
\end{aligned}$$

Figure 6: The typeset function and derivatives, corresponding to figure 5

Rules for differentiating the function CHI appear in [9] and in the user's guide. Figure 7 illustrates how the simple function

$$f(x_1, x_2) = \begin{cases} 2x_1 & \text{if } x_2 < 0, \\ x_1^2 & \text{if } x_2 \geq 0. \end{cases} \tag{1}$$

would be programmed. The corresponding FORTRAN 77 program for the derivatives appears in figure 8. The typeset output from the LaTeX file generated for this function appears in figure 9.

Observe from figure 8 that *each* part of the branch is *always* evaluated, but that only the proper branch is returned as a function or derivative value. This scheme can be desirable in some circumstances, such as when the branch variable is an interval that contains zero. However, for some functions, this situation may generate an excessive amount of computation. Furthermore, arithmetic errors may occur if some of the branches are not defined everywhere. This is not serious if the error handler can be set to continue execution after such errors (such as can be done with IEEE arithmetic, with the result set to NaN).

2.5 User-defined extensions

The user can define univariate functions and their derivatives. The syntax in the Fortran 90 program is: Y = U_(K, X, A, B) where U_ can appear anywhere in an arithmetic expression, K is an index, X is the argument (an independent or intermediate variable), and A and B are parameters. The differentiator translates this to a call to the function USRP(K,X,A,B), and its derivative with respect to X to a call to function USRPD(K,X,A,B). Fortran 90 examples of USRP and USRPD appear in the module USER_FUNCTIONS, supplied with the package. FORTRAN 77 equivalents can also be supplied.

3 On Higher-Order Derivatives and Modifications

The internal representation of the derivative of the function is in exactly the same form as the internal representation of the function itself. Furthermore, the set of allowable operations and standard functions, with the exception of the user-defined function U_, is closed under repeated differentiation. Thus, if the program does not contain user-defined functions[1],

[1]This restriction can be removed with some additional development, provided the user programs derivatives of the appropriate orders.

```
!  This test problem illustrates programming conditional branches.

SUBROUTINE ORIGINAL_FUNCTION(FILE_NAME)

  USE OVERLOAD

  CHARACTER (LEN=20):: FILE_NAME

      TYPE(CDLVAR), DIMENSION(2) :: X
      TYPE(CDLLHS), DIMENSION(1) :: F

!  Caution! Do not change the following statement --
!
      OUTPUT_FILE_NAME=FILE_NAME

      CALL INITIALIZE_CODELIST(X)

  F(1) = CHI(X(2),2*X(1),X(1)**2)

      CALL FINISH_CODELIST

END SUBROUTINE ORIGINAL_FUNCTION
```

Figure 7: Fortran 90 source defining a simple branch

```
      SUBROUTINE FG(X,F,G)

      DOUBLE PRECISION X( 2 )
      DOUBLE PRECISION F( 1 )
      DOUBLE PRECISION G( 1 , 2 )

      DOUBLE PRECISION Y( 9 )

     DO 10 I = 1,        2
         Y(I) = X(I)
  10 CONTINUE

     Y(        3) =   0.20000000000000000000D+01* Y(        1)
      Y( 4 ) = Y( 1 ) ** 2
      IF ( Y( 2 ) .LT.0 ) THEN
         Y( 5 ) = Y( 3 )
      ELSE
         Y( 5 ) = Y( 4 )
      END IF
     Y(        6) =  0.20000000000000000000D+01
     Y(        7) =   0.20000000000000000000D+01* Y(        1)
      IF ( Y( 2 ) .LT.0 ) THEN
         Y( 8 ) = Y( 6 )
      ELSE
         Y( 8 ) = Y( 7 )
      END IF
      Y( 9 ) = 0D0

      F( 1 ) = Y( 5 )

      G( 1 , 1 ) = Y( 8 )
      G( 1 , 2 ) = Y( 9 )

      END
```

Figure 8: The FORTRAN 77 derivative program corresponding to figure 7

$$\begin{aligned} f_1 &= \chi(x_2, [2.0, 2.0]x_1, x_1^2) \\ f_2 &= \chi(x_2, [2.0, 2.0], [2.0, 2.0]x_1) \\ f_3 &= 0 \end{aligned}$$

Figure 9: The typeset function and derivatives, corresponding to figure 7

OBJECTIVE_TO_GRADIENT can be repeatedly called to compute higher order derivative tensors.

Figure 10 shows the typeset output from the LaTeX file produced from the second derivative representation corresponding to the function of figure 3, printed as the original function with the routine PRINT_FUNCTION. Observe that the second derivatives are listed very redundantly in the order:

#	derivative
1	f_1
2	$\partial f_1/\partial x_1$
3	$\partial f_1/\partial x_2$
4	$\partial f_1/\partial x_1$
5	$\partial^2 f_1/\partial x_1^2$
6	$\partial^2 f_1/\partial x_1 \partial x_2$
7	$\partial f_1/\partial x_2$
8	$\partial^2 f_1/\partial x_2 \partial x_1$
9	$\partial^2 f_1/\partial x_2^2$
10	f_2
11	$\partial f_2/\partial x_1$
12	$\partial f_2/\partial x_2$
13	$\partial f_2/\partial x_1$
14	$\partial^2 f_2/\partial x_1^2$
15	$\partial^2 f_2/\partial x_1 \partial x_2$
16	$\partial f_2/\partial x_2$
17	$\partial^2 f_2/\partial x_2 \partial x_1$
18	$\partial^2 f_2/\partial x_2^2$

Although this present scheme is convenient and simple, it is clear that the redundancy is prohibitive for large numbers of variables or high-order derivatives. We have developed an alternate internal representation, without redundant representation, but additional work is necessary for full incorporation into the system. The present scheme is reasonable for first-order derivatives, while the FORTRAN 77 code can be modified manually for second-order derivatives.

Alternately, it is possible to write relatively simple drivers for derivatives of particular fixed order (such as two). Such drivers would take the (redundant) internal representation and output appropriate Fortran and LaTeX files that do not have redundancy. The user can do this by modifying the routines OUTPUT_FORTRAN_DERIVATIVE and PRINT_FUNCTION_GRADIENT.

Finally, the user may note that a different principle is used in producing the FORTRAN 77 file than the LaTeX file. The technique from the LaTeX file production can be used to produce an alternate FORTRAN 77 file in which

$$\begin{aligned}
f_1 &= ([4.0,4.0]x_1^3 - [3.0,3.0]x_1) - x_2\\
f_2 &= [4.0,4.0][3.0,3.0]x_1^2 - [3.0,3.0]\\
f_3 &= [-1.0,-1.0]\\
f_4 &= [4.0,4.0][3.0,3.0]x_1^2 - [3.0,3.0]\\
f_5 &= [4.0,4.0][3.0,3.0][2.0,2.0]x_1 - 0\\
f_6 &= -0\\
f_7 &= [-1.0,-1.0]\\
f_8 &= 0\\
f_9 &= 0\\
f_{10} &= x_1^2 - x_2\\
f_{11} &= [2.0,2.0]x_1\\
f_{12} &= [-1.0,-1.0]\\
f_{13} &= [2.0,2.0]x_1\\
f_{14} &= [2.0,2.0]\\
f_{15} &= 0\\
f_{16} &= [-1.0,-1.0]\\
f_{17} &= 0\\
f_{18} &= 0
\end{aligned}$$

Figure 10: The typeset second derivative tensor corresponding to figure 3

intermediate results of the computation are not stored, but parenthetical expressions appear.

4 Summary and Disclaimer

A relatively simple set of routines for a type of symbolic differentiation of programs is provided. This set of routines illustrates use of operator overloading technology and automatic differentiation ideas for such purposes. The routines are relatively easy to understand and modify. The technology is also available in other packages, such as the commercial optimization package and language AMPL [5], the C++ package ADOLC [6], and other packages mentioned in [7] and later. This package gives Fortran access, with source code in an easily understandable form.

The output FORTRAN 77 program contains a single operation per program statement, i.e. it is completely unrolled, and all intermediate quantities are stored. This can have certain advantages, but may be impractical for functions with large numbers of operations. For such problems, a sophisticated package such as ADIFOR may be appropriate.

Also, the LaTeX file, although giving correct expressions, is generated in a simple way. It may not be esthetically optimal, or fit within page boundaries, for complicated functions. Again, for such functions it is possible that commercial symbolic manipulation packages would provide appropriate solutions.

References

[1] Averick, B. M., Moré, J. J., Bischof, C. H., Carle, A., and Griewank, A., *Computing Large Sparse Jacobian Matrices using Automatic Differentiation*, SIAM J. Sci. Comput. **15** (2), pp. 285–294, 1994.

[2] Bischof, C. and Griewank, A., *ADIFOR: A Fortran System for Portable Automatic Differentiation*, technical report no. MCS-P317-0792, 1992.

[3] Dennis, J. E., and Schnabel, R. B., *Numerical Methods for Unconstrained Optimization and Nonlinear Least Squares*, Prentice-Hall, Englewood Cliffs, NJ, 1983.

[4] Fourer, R., Gay, D. M., Kernighan, B. W., *A Modeling Language for Mathematical Programming*, Manage. Sci. **36** (5), pp. 519–554, 1990.

[5] Gay, D. M., *AMPL Syntax Summary*, preprint, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1990.

[6] Griewank, A., Juedes, D., and Srinivasan, J., *User's Guide for ADOL-C – Version 1.0 August, 1990*, preprint, 1990.

[7] Griewank, A. and Corliss, G. F., ed., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991.

[8] Kearfott, R. B., Dawande, M., Du K.-S. and Hu, C.-Y., *INTLIB: A Portable FORTRAN 77 Interval Standard Function Library*, accepted for publication in ACM Trans. Math. Software.

[9] Kearfott, R. B., *A Fortran 90 Environment for Research and Prototyping of Enclosure Algorithms for Constrained and Unconstrained Nonlinear Equations*, accepted for publication in ACM Trans. Math. Software.