Edgar H. Sibley
Panel Editor

ALGLIB—a library of procedures that perform analytic differentiation and other simple symbolic manipulations—has certain advantages over existing and more comprehensive packages. It can be implemented in a high-level language of the user's choice using a pseudocode available from the authors, and it is easily interfaced with the user's programs.

ALGLIB, A SIMPLE SYMBOL-MANIPULATION PACKAGE

J. M. SHEARER and M. A. WOLFE

In numerical mathematics there are many instances where it is necessary to differentiate a function $f: R^n \rightarrow R^1$ one or more times (see, e.g., [13]). If a package that performs analytical differentiation is not available, the user must either carry out the differentiation by hand, a tedious and error-prone process, or compute a numerical approximation to the derivatives. In many instances both alternatives are unacceptable.

Several packages for performing symbolic computation are currently available, MACSYMA [2] and REDUCE [6] being among the best known. However, these packages are to a large degree "isolated" in the sense that they do not readily interface with programs written in a language of the user's choice. An application in which a FORTRAN program has been interfaced with REDUCE 2 is described by Watanabe in [17].

This paper describes a library of procedures (ALGLIB) that perform analytic differentiation and other simple symbolic manipulations. The library of procedures may be implemented in many high-level languages. Ideas on how the procedures might be embedded in a compiler to create a new hybrid language are described, and several applications of ALGLIB are presented along with the results obtained using S-algol [4] and Triplex S-algol [1, 10] implementations.

COMPUTABLE FACTORABLE FUNCTIONS

The ALGLIB package manipulates the set of computable factorable functions—a subset of the set of factorable functions described by Sisser [14]—using the following definition.

Definition

Let X be a given set, the elements of which may be represented by a computer, and on which the binary operations $+ : X \times X \rightarrow X$, $- : X \times X \rightarrow X$, $* : X \times X \rightarrow X$, and $/ : X \times X \rightarrow X$ are defined. A function $f: X^n \rightarrow X$ is a computable factorable function if and only if it can be represented as the last in a finite sequence of functions $\{f_j\}$ that are such that, if $x = (x_1, \ldots, x_n) \in X^n$, then

$$f_j(x) = x_j \quad (j = 1, \ldots, n); \tag{1}$$

and, if $j > n$, then $f_j(x)$ has one of the forms

$$f_k(x) + f_l(x) \quad (k, l < j), \tag{2}$$

$$f_k(x) - f_l(x) \quad (k, l < j), \tag{3}$$

$$f_k(x) * f_l(x) \quad (k, l < j), \tag{4}$$

$$f_k(x)/f_l(x) \quad (k, l < j), \tag{5}$$

or

$$\mathrm{T}[f_k(x)] \quad (k < j), \tag{6}$$

where $\mathrm{T}[\cdot] \in \mathrm{F} = \{-(\cdot), \mathrm{sqrt}(\cdot), \exp(\cdot), \ln(\cdot), \cos(\cdot), \sin(\cdot), \tan^{-1}(\cdot), (\cdot)^m$, where m is an integer$\}$. □

The set F may be extended to include any other function from X to X that may be evaluated in a particular computing environment provided that its derivative is itself a computable factorable function. This definition is illustrated by the following example.

Example
The function $f\colon R^3 \rightarrow R^1$ defined by

$$f(x_1, x_2, x_3) = \cos(x_1 + x_2 * x_3)$$

is a computable factorable function since we may write

$$f_1(x) = x_1,$$

$$f_2(x) = x_2,$$

$$f_3(x) = x_3,$$

$$f_4(x) = x_2 * x_3 = f_2(x) * f_3(x),$$

$$f_5(x) = x_1 + x_2 * x_3 = f_1(x) + f_4(x),$$

$$f_6(x) = \cos(x_1 + x_2 * x_3) = \cos(f_5(x)).$$

Clearly f is equal to the last in a finite sequence of functions that satisfies the conditions of the definition. □

Owing to the nature of the differentiation operator, the partial derivative of a computable factorable function with respect to any of the variables is itself a computable factorable function. Much work has been done on computer-generated analytic derivatives of factorable functions; see, for example, the work by Rall [12, 13], Sisser [14–16], Pugh[11], and McCormick [8].

DATA STRUCTURES

Given an expression that defines a computable factorable function, ALGLIB generates the sequence of functions $\{f_j\}$ that make up its factorable form and then stores this sequence efficiently. The function, once stored in this way, may then be differentiated, evaluated, output as a string, or composed or combined with other functions that are similarly stored. This section describes the data structures used to store the finite sequence $\{f_j(x)\}$. Any term in the sequence is one of the following:

(1) a constant;
(2) a variable (i.e., of the form (1));
(3) a binary term (i.e., of one of the forms (2)–(5));
(4) a unary term (i.e., of the form (6)).

To store a constant or a variable, we need only store the name of the constant or the variable and its current value. Storing unary and binary terms is slightly more complicated. A unary term contains an argument and an operator, where the argument is another term in the sequence. We could therefore store a unary term in a data structure consisting of a string and a pointer; the string represents the unary operator, and the pointer points to the argument. Similarly, a binary term could be stored in a structure composed of the operator, in a string, and pointers to each of the subterms. This gives rise to a binary tree (or more correctly, an acyclic graph) where each node represents a term in the sequence $\{f_j\}$ and each leaf node is a constant or a variable. The head of the tree represents the last term in the sequence $\{f_j\}$. The function of the Example is represented by the tree structure shown in Figure 1.

To avoid storing several representations of the same object, which may occur as a result of generating several trees that contain the same term, we require a simple and efficient method for checking a new node against those that have already been created.

One approach is to link the constants together in one ordered linked list, the unary terms together in a second list, and the binary nodes in a third. Thus, when a new node representing a constant is about to be created, the constant is checked against the constants in the linked list. If a duplicate is found, the new node is not created, and a pointer to the old representation is used; otherwise the new node is created and is added to the linked list so as to preserve the ordering. A similar process is used for unary and binary terms. Although it is clear that approaches using a more efficient search procedure would be desirable, the alternatives tend to complicate the other processes of the package and will not be dealt with here. In order to maintain the linked lists, a linking field must be introduced into the data structures for constants, unary terms, and binary terms. An index field would also be included in these data structures, as would a data structure for variables to facilitate the ordering of the lists. Since structures to represent all the variables are created when the variables are defined, we should never require a new node to represent a variable.

To keep the linked lists and the variables accessible, we create an information block that is associated with each set of variables. This information block contains a vector of pointers to each of the variables, a pointer to the list of constants, a pointer to the list of unary terms, and a pointer to the list of binary terms. We introduce a

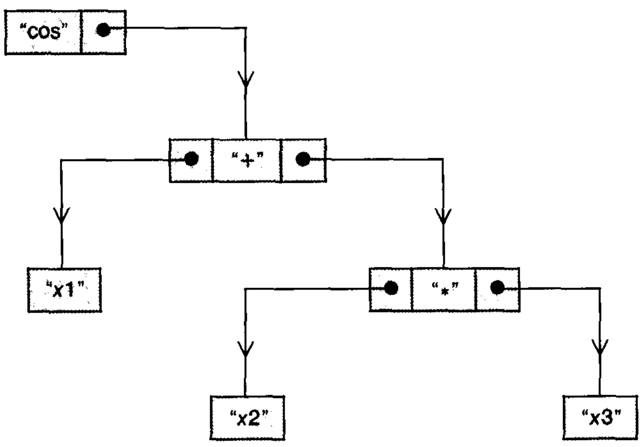

FIGURE 1. Tree Structure Representation of the Function of the Example

pointer into each structure definition that will be used to point to the information block corresponding to the variables on which the term corresponding to the structure is defined.

The preceding structures are adequate, but the package should be able to store the value of a term so that it may be reused automatically if the value of this term at the same point is required later. This involves introducing into the structure definitions for unary and binary terms a field to store the value of a term, and a flag to mark whether this value is up-to-date or not.

Similarly, if we create a tree structure to represent the derivative of a term with respect to one of the variables, it is then desirable to keep a pointer to this tree to avoid its being recomputed later. This involves introducing a vector of pointers into the structures for unary and binary terms. Each component of the vector points to the partial derivative of the term with respect to one of the variables. A nil pointer denotes that the derivative of this term with respect to this variable has not been computed.

The type declarations necessary in a Pascal implementation of the package for real-valued functions are given in Figure 2. In the type declarations presented in Figure 2, it is assumed that no more than 100 variables will be used and that all variable or constant names will be 20 characters or less. This, of course, could be altered to meet specific requirements.

```
type
  string = array [1..20] of char ;
  vecstring = array [1..100] of string ;
  termpointer = ↑term ;
  vectermpointer = array [1..100] of termpointer ;
  mattermpointer = array [1..100,1..100] of termpointer ;
  vecreal = array [1..100] of real ;
  matreal = array [1..100,1..100] of real ;
  blockpointer = ↑informationblock ;
  termclasses = (constant,variable,unary,binary) ;
  rootpointer = ↑root ;
  term =
      record
          index : integer ;
          value : real ;
          block : blockpointer ;
          case class : termclasses of
              constant :
                  ( constantname   : string ;
                    constantlink   : termpointer ) ;
              variable :
                  ( variablename   : string ) ;
              unary :
                  ( unaryarg    : termpointer ;
                    unaryop     : string ;
                    unarygrad   : vectermpointer ;
                    unaryknown  : boolean ;
                    unarylink   : termpointer ) ;
              binary :
                  ( binaryleft   : termpointer ;
                    binaryright  : termpointer ;
                    binaryop     : char ;
                    binarygrad   : vectermpointer ;
                    binaryknown  : boolean ;
                    binarylink   : termpointer )
      end ;
  informationblock =
      record
          constantterms  : termpointer ;
          unaryterms     : termpointer ;
          binaryterms    : termpointer ;
          variableterms  : vectermpointer ;
      end ;
```

FIGURE 2. Type Declarations Necessary in a Pascal Implementation

THE ALGLIB PACKAGE

The procedures that make up the ALGLIB package are described in this section. The procedure headings and an example call for each procedure are given for a pseudo-Pascal implementation involving real-valued functions. The type declarations given in the preceding section, and the appropriate type declarations for any variables, are assumed. It is also assumed that the factorable functions upon which ALGLIB operates map from R^n to R, but that R may be replaced with any other data type that has the properties of X in the Definition (p. 820). For example, R may be replaced with $I(R)$, the set of real intervals, if the implementation of Pascal supports interval arithmetic, as does Matrix Pascal [3].

The procedure `definevariables`, which has a heading of the form

```
procedure definevariables(variablenames:
vecstring) : vectermpointer;
```

takes the vector of strings that are intended to make up the variable names, creates a structure of type variable corresponding to each of these names, and returns a vector of pointers, each of whose components points to one of these structures. The names are checked for uniqueness and validity. To define a set of variables whose names are stored in the vector of strings, `names`, the statement

```
variables := definevariables(names);
```

is used, where `variables` has been defined as type `vectermpointer`.

The procedure `stringtofunction`, which has a heading of the form

```
procedure stringtofunction
(variables : vectermpointer; expression
: string) : termpointer;
```

takes as input the set of variables on which we wish to define the function, and a string representation of the function. The tree representation of the function is created, and a pointer to it is returned. Assuming we have defined a set of variables with names $x1$, $x2$, $x3$, as described above, and that the variable expression of type string contains the character string `cos(x1 + x2 * x3)`, the statement

```
f := stringtofunction
(variables, expression);
```

would create the tree representation of the Example (p. 821) and return a pointer to it.

The procedure `functionformat`, which has a heading of the form

```
procedure functionformat
(factorable: termpointer) : string;
```

reverses the process of `stringtofunction` and converts a function, held in its tree representation and pointed at by `factorable`, into a string ready for output. Extensive bracketing is used to avoid ambiguity.

Once the tree representation of a function has been created, it may be evaluated by the procedure `evaluate`, which has a heading of the form

```
procedure evaluate(factorable:
termpointer; point: vecreal) : real;
```

This procedure takes as input a pointer to the tree and the point at which the function is to be evaluated. The value of the function at the given point is returned. For example, if the vector of `reals`, `point`, contains the point at which the function `f` defined above is to be evaluated, then the statement

```
fofpoint := evaluate(f, point);
```

could be used.

The procedure `partial`, which has a heading of the form

```
procedure partial(factorable, variable:
termpointer) : termpointer;
```

takes a pointer to a function's tree representation and a pointer to a variable. It creates a tree representation of the derivative of the function with respect to the variable and returns a pointer to it. Thus the statement

```
dfbydx1 := partial(f, x[1]);
```

would create the tree structure representing $\partial f/\partial x_1$ and return a pointer to it.

The procedure `functionopfunction`, which has a heading of the form

```
procedure functionopfunction(a, b:
termpointer; op: string) : termpointer;
```

creates the tree representation of the two functions, `a` and `b`, combined by the given binary operator. Thus, if functions `f` and `g` had been created as described above, and the variable `operator` of type `string` contained the operator +, then the statement

```
h := functionopfunction(f, g, operator);
```

would create a tree representation of the function $h(\cdot) = f(\cdot) + g(\cdot)$.

The procedure `opfunction`, which has a heading of the form

```
procedure opfunction(arg: termpointer;
op: string) : termpointer;
```

creates the tree representation of the function that is obtained when the given operator acts on the given function. If the variable `operator` of type `string` contains the character string cos, then the statement

```
g := opfunction(f, operator);
```

would create the tree representation of the function $g(\cdot) = \cos(f(\cdot))$.

The procedure `compose`, which has a heading of the form

```
procedure compose(a: termpointer; b:
vectermpointer) : termpointer;
```

takes as input a pointer to the tree representation of a function from R^n to R^1 and a vector of pointers, each of whose components points to the tree representation of one of the components of a function from R^n to R^n. A tree representation of the function is obtained when the first function composed with the second is returned. For example, the statement

```
h := compose(f, g);
```

would create the tree representation of the function h: $R^n \rightarrow R^1$, where $h(\cdot) = f(g(\cdot))$.

The ALGLIB package also contains procedures that act on functions from R^n to R^n and on functions from R^n to $M(R^n)$, where $M(R^n)$ is the set of real matrices of order n.

IMPLEMENTATION

The ALGLIB package has been written in a pseudocode whose meaning should be clear to a programmer of most high-level languages. An annotated listing of the pseudocode may be obtained from the authors.

The package has been implemented for real-valued functions in S-algol [4] and for interval-valued functions in Triplex S-algol [1, 10], which is an extension of S-algol that supports interval arithmetic. Work on a Pascal implementation for real-valued functions is in progress. Full listings of the S-algol and Triplex S-algol implementations, together with a comprehensive manual describing the package, are available on request from the authors.

Instead of implementing the ALGLIB package as a library of procedures, the procedures could be embedded in a compiler, and a new data type, representing the tree form of a function, could be introduced. This would allow the incorporation of combination, composition, and evaluation of functions as integral parts of the language.

APPLICATIONS

The ALGLIB package is currently being used at the University of St. Andrews in research on the solution of systems of nonlinear algebraic equations and on unconstrained optimization. A standard by which new algorithms may be compared is provided by the packages for unconstrained optimization and nonlinear algebraic equations that have been given in pseudocode form by Dennis and Schnabel [5]. These packages have been implemented in S-algol, and the implementation has been interfaced with ALGLIB in such a way that the user, in addition to the options for function, gradient, and Jacobian and Hessian evaluation provided by Dennis and Schnabel, may also use ALGLIB. A listing of the code is obtainable from the authors.

Test Problems

Dennis and Schnabel [5] have given a set of test problems for use with their packages. The first three of these are used here to illustrate the S-algol implementation, with and without the use of ALGLIB.

Execution Times for the Packages of Dennis and Schnabel with and without ALGLIB

If ALGLIB is used, the data structures corresponding to the function and to the gradient and Hessian (or to the Jacobian) must be set up before numerical computation can begin. Let the CPU time that is required to set up the required data structures be t_s seconds. Let the CPU time that is required for the subsequent numerical computation be t_c seconds. Let the CPU time that is required for the computation of a solution using analytical expressions for the function, and for the Jacobian (or for the gradient and Hessian), be T seconds.

Nonlinear Algebraic Equations and Unconstrained Optimization

Table I contains results that are obtained from the package for solving systems of nonlinear algebraic equations. The column with heading T corresponds to Newton's method with analytical Jacobian.

Table II contains results that are obtained from the package for unconstrained optimization. The column with the heading T corresponds to the use of Newton's method with analytical gradient and Hessian.

In Tables I and II, the total CPU time required to solve a given problem is the sum of t_s and t_c if ALGLIB is used; this sum should be compared with the CPU time T, which is required if analytical expressions for partial derivatives are required. As may be expected, more CPU time is required if ALGLIB is used. One should, however, take into account the considerable, sometimes prohibitive, time required to calculate ana-

TABLE I. Solution of Nonlinear Algebraic Equations

Example	n	t_s	t_c	T
1	2	0.93	0.45	0.22
2	4	4.26	6.29	3.66
3	4	15.93	4.72	1.18

Description of Examples 1, 2, and 3: Corresponding to each test problem is a set of mappings f_i: $R^n \rightarrow R^1$ ($i = 1, \ldots, n$). A test problem for the unconstrained optimization package of Dennis and Schnabel consists of minimizing f: $R^n \rightarrow R^1$ defined by

$$f(x) = \sum_{i=1}^{n} \{f_i(x)\}^2,$$

and a test problem for the package for solving systems of nonlinear algebraic equations consists of solving $F(x) = 0$, where F: $R^n \rightarrow R^n$ is defined by

$$F_i(x) = f_i(x) \qquad (i = 1, \ldots, n).$$

For each test problem, an initial estimate of the solution is provided. Explicit formulas for the f_i, together with the initial estimates of the solutions, are given in [5, Appendix B].

TABLE II. Unconstrained Optimization

Example	n	t_s	t_c	T
1	2	2.03	5.40	2.13
2	4	11.85	7.15	4.87
3	4	77.36	22.72	2.91

*See legend for Table I.

lytical expressions for partial derivatives and to correct the algebraic mistakes, which nearly always arise.

Sisser's Minimization Algorithm

Sisser [14] has described how the Hessian of a twice differentiable factorable function f: $R^n \rightarrow R^1$ may be expressed in the form

$$H(x) = D(x) + \sum_{i=1}^{m} [u_i(x)c_i(x)v_i(x)^T + v_i(x)c_i(x)u_i(x)^T] \quad (7)$$

where m and the functions c_i: $R^n \rightarrow R^1$, u_i: $R^n \rightarrow R^n$, and v_i: $R^n \rightarrow R^n$ ($i = 1, \ldots, m$) depend on the function being considered; he has used eq. (7) in a modification of Newton's method for minimizing factorable functions [16].

ALGLIB has been used to generate the tree representations of the functions c_i, u_i, and v_i ($i = 1, \ldots, m$) and the integer m for several factorable functions, thereby making possible an S-algol implementation of Sisser's minimization algorithm.

Interval Arithmetic

Moore and Jones [9] and Jones [7] have described a search procedure for bounding isolate zeros of a function f: $R^n \rightarrow R^n$ using interval analysis, where the solutions lie in a given initial box. The Triplex S-algol implementation of ALGLIB has been interfaced with a Moore and Jones search procedure in such a way that either ALGLIB or analytic expressions for the function and Jacobian can be used.

The function f: $R^n \rightarrow R^n$ defined by

$$f(x) = Ax + Td(x) + c,$$

where $A \in M(R^n)$ is tridiagonal with diagonal elements equal to 2 and the leading off-diagonal elements equal to -1, $T \in M(R^n)$ is tridiagonal with diagonal elements equal to $\frac{10}{12}$ and leading off-diagonal elements equal to $\frac{1}{12}$, d: $R^n \rightarrow R^n$ is continuous diagonal and isotone with

$$d_i(x) = \frac{1}{(n+1)^2} \exp(x_i) \qquad (i = 1, \ldots, n),$$

and $c \in R^n$ is given by

$$c_i = \begin{cases} \frac{1}{12}(1/(n+1)^2) & (i = 1) \\ 0 & (i = 2, \ldots, n-1) \\ \frac{1}{12}(1/(n+1)^2) & (i = n), \end{cases}$$

has a unique zero. The search procedure of Moore and Jones has been used in a Triplex S-algol implementation to search for the unique zero of f in an n-cube centered at the origin of radius 10 with $n = 3$, both with and without ALGLIB. ALGLIB requires 11.24 seconds to set up its internal representations of the function and its F-derivative; and 20 bisections, 81 function evaluations, and 34 Jacobian evaluations are required both using and not using ALGLIB. The search requires 153 seconds of CPU time to isolate the zero in a box x^* such that

$$\|w(x^*)\| < 10^{-10}$$

when ALGLIB is used. When ALGLIB is not used, 147 seconds of CPU time are required. The slight difference in CPU time is more than offset by the convenience and reliability of using the ALGLIB package.

Acknowledgment. The authors are grateful to M. Monsi for obtaining the results that are given in Tables I and II.

REFERENCES

1. Bailey, P.J., Cole, A.J., and Morrison, R. Triplex user manual. CS/82/5, Dept. of Computational Science, Univ. of St. Andrews, N. Haugh, St. Andrews, Fife, Scotland, 1982.
2. Bogen, R.A., et al. MACSYMA reference manual. Version 6, Laboratory for Computer Science, M.I.T., Cambridge, Mass., 1977.
3. Bohlender, G., et al. Matrix Pascal. Res. Rep. RC9577 (42297), IBM Research Division, Yorktown Heights, N.Y., 1982.
4. Cole, A.J., and Morrison, R. *An Introduction to Programming with S-algol.* Cambridge University Press, New York, 1982.
5. Dennis, J.E., and Schnabel, R.B. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, Englewood Cliffs, N.J., 1983.
6. Hearn, A. REDUCE 2 user's manual. Univ. of Utah, Salt Lake City, 1973.
7. Jones, S.T. Searching for solutions of finite nonlinear systems—An interval approach. Ph.D. dissertation, Computer Science Dept., Univ. of Wisconsin-Madison, Madison, 1978.
8. McCormick, G.P. *Nonlinear Programming—Theory, Algorithms, and Applications.* Wiley, New York, 1983.
9. Moore, R.E., and Jones, S.T. Safe starting regions for iterative methods. *SIAM J. Numer. Anal. 14*, 6 (Dec. 1977), 1051–1065.
10. Morrison, R., Cole, A.J., Bailey, P.J., Wolfe, M.A., and Shearer, J.M. Experience in using a high level language which supports interval arithmetic. In *Proceedings of ARITH6, the 6th Symposium on Computer Arithmetic* (Aarhus, Denmark, June 20–22). IEEE Computer Society Technical Committee on Computer Architecture, 1983, pp. 74–78.
11. Pugh, R.E. A language for nonlinear programming problems. *Math. Program.* 2 (1972), 176–206.
12. Rall, L.B. *Computational Solution of Nonlinear Operator Equations.* Wiley, New York, 1969.
13. Rall, L.B. Applications of software for automatic differentiation in numerical computation. *Computing* Suppl. 2 (1980), 141–156.
14. Sisser, F.S. Computer-generated interval extensions of factorable functions and their derivatives. *Int. J. Comput. Math. 10* (1982), 327–336.
15. Sisser, F.S. Inverting an interval Hessian of a factorable function. *Computing 29* (1982), 63–72.
16. Sisser, F.S. A modified Newton's method for minimizing factorable functions. *J. Optim. Theory Appl. 38*, 4 (Dec. 1982), 461–482.
17. Watanabe, S. Hybrid manipulations for the solution of systems of nonlinear algebraic equations. *RIMS, Kyoto University 19* (1983), 367–395.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*software libraries*; E.2 [**Data Storage Representations**]: *composite structures*; I.1.1 [**Algebraic Manipulation**]: Expressions and their Representation

General Terms: Algorithms, Human Factors, Languages, Performances, Reliability

Additional Key Words and Phrases: computable factorable function, data structures, interval arithmetic, partial differentiation, symbolic computation

Authors' Present Address: J.M. Shearer and M.A. Wolfe, Dept. of Applied Mathematics, University of St. Andrews, St. Andrews, Fife, KY16 9SS, Scotland.