A Fortran 90 Environment for Research and Prototyping of Enclosure Algorithms for Nonlinear Equations and Global Optimization

R. Baker Kearfott*
University of Southwestern Louisiana

Abstract

An environment for general research into and prototyping of algorithms for reliable constrained and unconstrained global nonlinear optimization and reliable enclosure of all roots of nonlinear systems of equations, with or without inequality constraints, is being developed. This environment should be portable, easy to learn, use, and maintain, and sufficiently fast for some production work. The motivation, design principles, uses, and capabilities for this environment are outlined. The environment includes an interval data type, a symbolic form of automatic differentiation to obtain an internal representation for functions, a special technique to allow conditional branches with operator overloading and interval computations, and generic routines to give interval and non-interval function and derivative information. Some of these generic routines use a special version of the backward mode of automatic differentiation. The package also includes dynamic data structures for exhaustive search algorithms.

Categories and Subject Descriptors: G.1.5[**Numerical Analysis**]: Roots of Nonlinear Equations, G.1.6: Optimization, G.4[**Mathematical Software**], D.3.3[**Programming Languages**]: Language Constructs and Features

General Terms: Programming environments

Additional Key Words and Phrases: Fortran 90, automatic differentiation, nonlinear algebraic systems, global optimization, symbolic computation

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

1 Introduction, Background, and Motivation

Numerous applications benefit from enclosure methods for numerical nonlinear analysis. Such methods include *rigorous* global optimization, both constrained and unconstrained, and *rigorous* location of all roots to nonlinear systems of equations, with or without side inequality constraints.

Global optimization is important in engineering, biological and economic modelling, and other applications; see [7] and [8]. Furthermore, enclosure (i.e. interval) methods, when applicable, not only can provide solutions *with certainty,* but can also be more efficient than other methods; see [14] or [37].

Enclosure methods for unconstrained and constrained solution of nonlinear systems are useful in robotics in sensor data analysis and collision detection ([12] and [13]), generally for reliability in computer graphics (as in [29], [36] and elsewhere), etc.

Ad hoc algorithms for each of the above applications are sometimes the most efficient. However, all such enclosure methods contain common sub-tasks, such as computing interval residuals or interval function values. Furthermore, a substantial theory has been developed (see [32]), and there are many computational tools that can be incorporated in these methods in different ways; see, for example [14], or, for recent tools of ours, [20], [22], or [23]. It is still unclear what the scope of applicability of these tools is. Furthermore, efficient prototyping of these tools requires a programming environment in which they are easily accessible in a uniform way.

General research on such methods can proceed within a language such as Fortran 77. For example, with elements of the Fortran 77 package INTBIS ([21]) we have investigated acceleration at singular roots (in [19]) and bound-constrained global optimization (in [24]). However, the speed of such investigations is constrained by lack of

an interval data type Giving tasks to graduate students, we have personally observed much higher productivity with an interval data type (in ACRITH-XSC [39]) than with accessing interval arithmetic through subroutine calls. Routines are much smaller, algorithms are more easily understood by reading the actual routines, and routines are more easily maintained[1].

a universal representation of functions Even traditional approximate nonlinear equation solvers and optimizers require routines to evaluate

[1]This fact has been recognized for decades, and was the motivation behind [5] and [41].

residuals or objective functions, gradients, and Jacobians or Hessian matrices for Newton steps, etc. A major burden has been the necessity to code both functions and derivatives. This is even more so for a general investigation of enclosure methods, since function information is used in more contexts. For example, not only may interval enclosures for an objective function, gradient, and Hessian matrix be required, but also floating point values, as well as relationships among the intermediate quantities produced during evaluation; there are possibly ten or more separate routines to return information on a single mathematical function. For this reason, *generic interpreters* for each of these functions (interval function, interval gradient, etc.), as well as a *globally available internal representation* of specific functions are useful. Means of automatically generating such representations from simple programs are required.

dynamic lists Branch and bound techniques are common to both deterministic global optimization and reliable root isolation in nonlinear systems. In such techniques, two sub-regions are produced from a single parent region. One of the child regions is placed, in order, in a list, while the other is kept for further processing. However, the order of the list and its management differ from algorithm to algorithm, and insertion into the list may be required in different, separated, subroutines. Experimentation with new algorithms may demand moving the places where the lists are required. Furthermore, the size of each list element depends on the dimension of the problem, and the maximum required number of elements of a list is not known a priori. Thus, dynamic, generic list operations, with encapsulated details, are advantageous.

encapsulation of certain operations Besides list operations, other common operations include computation of preconditioners of various types, computation of a step of an interval Newton method, and others. Encapsulation of these would greatly ease high-level algorithmic research and development. This encapsulation should be in subroutines, functions, or operators with a natural interface. In particular, only arguments that are logically required should appear; arguments such as array bounds and workspace vectors slow research and prototyping.

A computer language to implement an environment with the above attributes *must* have

- user-defined data types and operator overloading,

- a pointer data type,
- dynamic memory allocation, and
- good facilities for defining and accessing global data.

Furthermore, for portability the language should be standardized and widely available. This is particularly important when hardware is rapidly improving and changing, when remote access to diverse hardware is easily available, and as obsolete equipment is rapidly retired. Under similar conditions, the language should also admit a style for easy maintenance and modification.

We have found Fortran 90 to have the necessary attributes. Knowing Fortran 77, we had no difficulties learning Fortran 90, and our Fortran 90 code for the package seems particularly natural. Finally, necessary non-interval auxiliary routines, such as a Fortran 77 sparse linear system solver, are immediately seamlessly accessible, without *any* necessary modification.

In the remainder of the paper, we briefly describe key aspects of our environment under development. In §2, we explain our interval data type. We briefly describe the internal representation of our functions and its generation in §3. We single out a particular aspect, representation of conditional branches, in §4. We outline our generic routines for interpreting this representation in §5, while key properties of our list-processing capabilities appear in §7. We mention what can be viewed as a special type symbolic differentiation capability, available in the package, in §6.

We assume some familiarity with interval arithmetic in §2, §4, and §5. Introductions to the subject can be found in [1], [14], [30], or [32].

A description of package operations such as preconditioner computation and interval Gauss–Seidel steps will appear elsewhere.

2 The Interval Data Type

We have recently produced INTLIB ([27]) as a portable Fortran 77 library to support basic interval arithmetic functions and interval arithmetic evaluation of the elementary functions. However, direct use of INTLIB requires writing a subroutine call for each elementary operation, such as an addition or multiplication. As mentioned in §1, this leaves the developer or maintainer at a substantial disadvantage. However, since the Fortran 90 standard contains Fortran 77, we may directly use INTLIB as a supporting package when defining an interval data type. We have created a Fortran 90 module INTLIB_ARITHMETIC precisely for this purpose.

```
PROGRAM TEST_INTERVAL_ARITHMETIC

!  This routine tests the Fortran 90 interface to the elementary
!  interval arithmetic portion of INTLIB.

   USE INTLIB_ARITHMETIC

      TYPE(INTERVAL) X

      CALL SIMINI        ! Initialize machine constants and interval
                         ! constants used in the elementary functions
      X = IVL(1,2)

      WRITE(6,*) X**2 + 3*X + 2

END PROGRAM TEST_INTERVAL_ARITHMETIC
```

Figure 1: Example – interval arithmetic using Fortran 90 overloading.

An example of the use of this module appears in Figure 1. With a certain Fortran 90 compiler on a DOS-based PC, this program produces the output

```
5.9999999999999947  12.0000000000000107
```

The same computation, accessing INTLIB in Fortran 77 appears in Figure 2. The output was exactly the same as above.

Previously, interval data types have been available with products such as the Augment precompiler [5] with [41], Pascal-SC [34], Fortran-SC or ACRITH-XSC [2], etc. See also [25] for additional references. However, only recently has it become possible to provide portable packages accessible to most of the scientific and engineering community.

W. Walter is developing an alternate Fortran 90 package for interval arithmetic [38]. This package, termed FORTRAN-XSC, is, to a certain extent, a portable version of ACRITH-XSC. Developed upon a portable accurate dot product, it has substantial support for linear algebra operations.

In contrast, our interval arithmetic module, with INTLIB as supporting package, does not have an accurate dot product, but presently has the most common elementary functions. Its arithmetic, rigorous but not optimally accurate, should be adequate in many situations, and may be somewhat faster than the arithmetic in FORTRAN-XSC.

Our interval arithmetic module has also been designed to be compatible with ACRITH-XSC in the sense that the names of supported operations and

```
C  This standard Fortran-77 routine uses INTLIB directly.

      DOUBLE PRECISION X(2)

      DOUBLE PRECISION TMP1(2), TMP2(2)

      CALL SIMINI        ! Initialize machine constants and interval
                         ! constants used in the elementary functions
      X(1) = 1D0
      X(2) = 2D0
      CALL RNDOUT(X,.TRUE.,.TRUE.)

      CALL POWER(X,2,TMP1)
      CALL SCLMLT(3D0,X,TMP2)
      CALL ADD(TMP1,TMP2,TMP1)
      CALL SCLADD(2D0,TMP1,TMP1)

      WRITE(6,*) TMP1(1), TMP1(2)

      END
```

Figure 2: Example – interval arithmetic directly in Fortran 77.

elementary functions match. This should facilitate conversion of ACRITH-XSC programs.

3 Representing Functions – Code Lists

To solve systems of nonlinear equations or to find optima, researchers and practitioners must represent functions and derivatives in a computer-usable form. Thought has been given to this, beginning with the first use of digital computers for such problems. The most straightforward method is to program the function and any required derivatives as separate subroutines or functions. However, computation and coding of the derivatives is error-prone. Some symbolic manipulation packages can produce programs for the derivatives, given programs for the functions. However, the resulting expressions are sometimes so complex (not in lowest terms) that they are unusable. Finite difference approximations are sometimes used, but these usually contain significant amounts of both roundoff and trunction errors; furthermore, their use is illogical in most contexts where interval arithmetic is applied.

A third alternative is automatic differentiation. See, for example [9],

[10], [15], or [33]. In the forward mode, as in [33], the arithmetic operations and elementary functions can be overloaded, for a data type that simultaneously contains function and derivative values. In the backward mode, intermediate quantities obtained during evaluation of the function are stored, then later combined to produce derivatives. The backward mode can produce gradients for a scalar-valued function in a time proportional to the number of operations necessary to evaluate the function, but also has storage requirements proportional to the number of operations necessary to evaluate the function. Also, representation of branching (IF-THEN-ELSE) in evaluation of the function poses difficulties when the backward mode is used. See [15] and the papers in [11].

Our function representation scheme is related to the backward mode of automatic differentiation. In particular, we use operator overloading for a *code list data type,* that we define in a Fortran 90 module OVERLOAD. For example, the actual sequence of computer operations for addition of two code list variables consists of writing a numeric code identifying the operation as addition, as well as the addresses of the operands, to a file. The complete result after execution of a program defining a function is a file containing the sequence of operations for that function. This information is termed a "linear representation," since no loop constructs explicitly appear[2].

An example of a program to generate a code list appears in Figure 3, while the result of running this program appears in Figure 4. The first row in Figure 4 gives dimension information, such as the number of dependent variables, number of independent variables, number of intermediate quantities produced during computation, numbers constants of two types, and number of conditional branches[3]. The next four rows identify operations, operand addresses, and result addresses. For example, operation 22 is multiplication by a constant, operation 5 is taking the square, operation 20 is addition, and operation 23 is addition of a constant. Row 6 identifies intermediate quantity 5 as a dependent variable. The last two rows contain the values of the constants and the rows of the code list with which they are associated.

We will give complete details of the definition of the code list, as well as a complete description of syntax and capabilities for the code list variable types, in a separate user's guide.

We may think of the code list as defining a sequence of relations

$$x_{p_j} = \phi_j(x_{q_j}, x_{r_j}), \quad 1 \le j \le N_{\mathrm{OPS}} \tag{1}$$

[2]It is also termed a "Wengert list," from R. E. Wengert [40].

[3]See §4 for an explanation of conditional branches.

```
!  This program illustrates use of module OVERLOAD to generate a
!  code list.

PROGRAM TEST_FUNCTION

  USE OVERLOAD

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

      OUTPUT_FILE_NAME='FUNCTEST.CDL'

      CALL INITIALIZE_CODELIST(X)

  F(1) = X(1)**2 + 3*X(1) + 2

      CALL FINISH_CODELIST

END PROGRAM TEST_FUNCTION
```

Figure 3: Simple program to generate a code list.

```
        1         1         5         5         0         1         1         0
  22   2   1   0
   5   3   1   0
  20   4   3   2
  23   5   4   0
  18   5   0   0
   1     0.29999999999999991100E+01     0.30000000000000008900E+01
   4     0.19999999999999995600E+01     0.20000000000000004400E+01
```

Figure 4: The code list file for the function in Figure 3.

where N_{OPS} is the total number of operations to evaluate the function. Each ϕ represents an elementary arithmetic operation ($+$, $-$, $*$, or $/$) or a standard function, such as sin, exp, or power $**$.

Our code list differs from that in some automatic differentiation applications, since *values* are not initially stored, but only symbolic information for the sequence of operations defining the function. Thus, the code list can be placed in global storage, made available in a module and, in principle, used in *any* routine requiring a specific function as data. These routines may include subroutines to compute residuals, gradients or Jacobi matrices of either the original system or the *expanded system* formed by assigning variables to the intermediate quantities, or to solve for one variable in terms of another [4].

In further contrast to some schemes for code lists, we presently make no attempt to identify "active" and "dead" variables, i.e. to restructure the code list to reduce the number of rows by identifying intermediate quantities that are no longer needed during the course of function evaluation. This is because, in some contexts, we must think of the code list as defining relationships among the intermediate quantities, and not merely as a prescription to obtain the final dependent variable values. In particular, we can use the relationships between intermediate quantities produced during evaluation of the function to reduce overestimation in the final results; see [16], [22] and [28].

4 Representing Conditional Branches

As mentioned in §3, conditional branches pose a problem when generating code lists through operator overloading, since straightforward execution of a program containing conditional branches results in different code lists, depending on current values of the branch variables. Since we think of (and indeed, in our nonlinear equations and global optimization algorithms use) the intermediate quantities as variables, the number of variables in our system would dynamically change.

However, conditional branches are important in various applications involving nonlinear systems and global optimization. For example, in graphics, computing the intersection of B-spline surfaces would require working with functions specified through conditional branches. For this reason, we have designed a special *branch function* **CHI**. The resulting code list contains the intermediate variables for *both* conditional branches, as well as information concerning when a particular branch is valid. Function and derivative routines can then interpret this code list appropriately, depending on the values of the branch variables. Also, to obtain interval inclusions of functions and derivatives, evaluation of both branches is necessary when the decision variable takes on interval values; thus, if we desire a uniform environment for both interval and standard floating point arithmetic, storage of

[4]It is possible to solve for one variable in terms of another since inverses are known for the elementary operations such as $x_p = \sin(x_q)$; we get tight bounds on the ranges of these inverses, without linearization. The inverses may consist of more than one point or interval; in those cases, we intersect with the original value of the intermediate variable and return the list of resulting intervals.

```
IF (X(1).LT.0) THEN
   F(1) = 2*X(1)
ELSE
   F(1) = X(1)**2
END IF
```

Figure 5: An ordinary conditional branch described by CHI in Figure 6.

both branches of the symbolic representation is not wasteful.

The function

$$x_p = \mathrm{CHI}(x_s, x_q, x_r) \tag{2}$$

will return x_q if $x_s < 0$, x_r if $x_s \geq 0$, and $x_q \cup x_r$ otherwise[5]. For example, suppose we wish to program the conditional branch in Figure 5. If this represents the complete function definition, then we may write the simple program in Figure 6. The code list produced from this program appears in Figure 7. There, operation 27, in line 4 of the code list, represents the branch function of equation 2. We note that *both* branches of the conditional statement appear in the code list[6]. The last row in Figure 7 gives the address of the quantity x_s used in the branch decision.

With interval arithmetic, we have devised natural rules for evaluating and differentating χ within the functions described in §5. For example, the derivatives of CHI are defined as follows.

[5]In this case, the result may consist of two intervals.

[6]The multiplication occurs in the second row, and the power occurs in the third row of Figure 7.

```
!  This is a simple test of the characteristic function for handling
!  if-then-else.

PROGRAM FCHITS

  USE OVERLOAD

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

      OUTPUT_FILE_NAME='FCHITS.CDL'

      CALL INITIALIZE_CODELIST(X)

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

      CALL FINISH_CODELIST

END PROGRAM FCHITS
```

Figure 6: Simple illustration of the branch function.

```
        1         1         4         4         0         1         0         1
    5   2   1   0
   22   3   1   0
   27   4   3   2
   18   4   0   0
    2     0.19999999999999995600E+01     0.20000000000000004400E+01
    3   1
```

Figure 7: The code list produced by the program in Figure 6.

If $x_s \geq 0$ then

$$\partial\chi/\partial x_q = 1$$
$$\partial\chi/\partial x_r = 0$$

If $x_s < 0$ then

$$\partial\chi/\partial x_q = 0$$
$$\partial\chi/\partial x_r = 1$$

Otherwise :

$$\partial\chi/\partial x_q = [0,1]$$
$$\partial\chi/\partial x_r = [0,1]$$

The basic formulas for symbolic differentiation of code lists containing CHI are

$$\begin{aligned} \frac{\partial\chi(x_s,x_q,x_r)}{\partial x_q} &= \chi(x_s,1,\frac{\partial x_r}{\partial x_q}) \\ \frac{\partial\chi(x_s,x_q,x_r)}{\partial x_r} &= \chi(x_s,\frac{\partial x_q}{\partial x_r},1) \end{aligned}$$

Differentiation with respect to the decision variable is more problematical. For numerical differentiation, we may use

$$\frac{\partial\chi(x_s,x_q,x_r)}{\partial x_s} = \begin{cases} 0 & \text{if } \; x_s \geq 0 \text{ or } x_s < 0 \\ 0 & \text{if } \; x_q = x_r \text{ and } x_r \text{ is a point,} \\ \text{ELSE:} & \\ [0,\infty) & \text{if } \; x_r > x_q, \\ (-\infty,0] & \text{if } \; x_r < x_q, \\ (-\infty,\infty) & \quad\text{otherwise} \end{cases}$$

for rigor in the computations. In actual branch-and-bound algorithms, however, the domain can be subdivided into regions where x_s is of constant sign. Symbolic differentiation cannot be done unless we assume $x_q = x_r$ where $x_s = 0$, so we may set $\frac{\partial\chi(x_s,x_q,x_r)}{\partial x_s} = 0$. Such a case occurs, for example, when χ is describing a continuous spline.

Though we arrived at this structure independently, this handling of conditional branches is similar to the treatment of logical variables described in [36].

5 Using the Code List – Obtaining Function and Derivative Values

Our package will include *interpretive* routines for evaluating functions and derivatives for both interval and floating point data. The capabilities presently under development are

1. computation of the dependent variables, given the independent variables for both interval and floating point data types;
2. computation of the Jacobi matrix for the dependent variables with respect to the independent variables, given the independent variables.
3. computation of the Jacobi matrix for the dependent variables with respect to the independent variables, given the intermediate quantities in evaluation of the function.
4. computation of all of the intermediate quantities from the independent variables, through a "forward sweep" of the code list.
5. computation of the Jacobi matrix for the intermediate quantities with respect to the intermediate quantities.
6. solution for one intermediate quantity in terms of the others, using a specified relation from the code list.
7. computation of the Hessian matrix (or tensor) of the dependent variables with respect to the independent variables, given the independent variables.

The code list, along with all dimension information, is accessible to each of the routines through a Fortran 90 module. All workspace is allocated dynamically. The main programming device is the Fortran 90 CASE statement, with a case corresponding to each operation defined for the code list[7]. Depending on the compiler implementation, this construct will interpret the code list efficiently. Also, use of the internal representation and module allows a particularly simple user interface. For example, interval values for *any* function can be obtained with the statement

[7]This technique is the natural way to interpret code lists. It was first suggested to the author in a private conversation with Arnold Neumaier.

```
CALL F(X,FVAL)
```

The programmer's only obligation is to assure that the independent variable values are in the array X and that the array FVAL has enough storage to hold the dependent variable values[8].

As is seen from equation 1, with the exception of the branch statement of equation 2, each elementary operation has at most two operands and one result. Thus, if we think of each intermediate quantity and each dependent variable definition as equations, and each independent variable and intermediate variable as new independent variables, then the Jacobi matrix of the resulting *expanded system* has at most three non-zero entries in each row. Assuming that such matrices must be stored, it is efficient to store them in a two-dimensional array whose row bound is 3. Information about the columns in which the non-zero elements occur in a given row is already available in the code list. We will refer to Jacobi matrices stored with this data structure as *expanded Jacobi matrices*[9].

We use a backward substitution process to compute the Jacobi matrices of the original dependent variables with respect to the original independent variables from expanded Jacobi matrices. In this technique, known by researchers in automatic differentiation, we eliminate the entries in columns of the expanded Jacobi matrix corresponding to intermediate variables from the rows corresponding to dependent variables. The Jacobi matrix of the dependent with respect to the independent variables then occurs in the columns corresponding to the independent variables. Experiments in [16] hint that this method can achieve tighter interval enclosures for the original Jacobi matrix than straightforward evaluation. This is particularly true when the bounds on the intermediate quantities obtained during function evaluation are tightened using the relationships among the intermediate quantities. The overall complexity of our routine to do this is

$$\mathcal{O}\left(N_{\mathrm{OPS}} + N_{\mathrm{EQ}}\right) + \mathcal{O}\left(N_{\mathrm{EQ}} N_{\mathrm{OPS}}\right),$$

where N_{OPS} is the number of operations required to evaluate the function and N_{EQ} is the number of components of the function (i.e. the number of dependent variables). This complexity is not better than that of the forward

[8]Unfortunately, this example also illustrates a minor inflexibility. It is complicated to devise a routine in Fortran 90 that works efficiently with the syntax FVAL = F(X), where FVAL is an array or user data type without a preset size. This is due to technical aspects of the standard related to lack of automatic "garbage collection".

[9]This technique was proposed by Xiaofa Shi in a private conversation.

mode, when the matrix has at least as many rows as columns, but it is easier to use independently tightened interval bounds on the intermediate quantities in the backward mode.

6 Symbolic Differentiation of Code Lists

Since there is an elementary relationship between each elementary operation ϕ in Equation 1 and its derivative with respect to the operands, we may produce the code list for a derivative tensor, given the code list for the original dependent variables. We designed the set of operations for our code lists to be closed under such differentiation. This differentiation can be viewed as *symbolic*, since its input is a defining representation for a function independent of argument value, and its output is a similar representation. However, it differs from traditional symbolic differentiation and manipulation, since algebraic expressions are not stored, but only elemental relationships defining a particular representation of the function.

We have a routine that outputs a derivative code list of exactly the same form as the function code list. The routines mentioned in §5 can thus be applied to these derivative code lists to obtain numerical (interval or floating point) values of higher-order derivatives. For example, an alternative to using the Jacobi matrix routine would be to symbolically differentiate the code list, then use the function routine. The differentiation routine can also, in principle, be applied an indefinite number of times to its own output, for derivatives of arbitrary order.

Several issues arise with this technique. First, higher-order derivative tensors of functions of many independent variables exhibit ever-higher symmetries. A scheme meant for high-order derivatives should thus take account of such symmetry to reduce redundant storage and operations; for example, it may be possible to improve our scheme using ideas in [31]. Second, many components of higher-order tensors typically are zero; we can possibly use ideas in [6]. In general, derivative code lists should contain structure information for storage economization.

A third issue is the extent of applicability of derivative code lists. The size of a derivative code list is larger than the original list, but how much larger depends on the implementation of the differentiation scheme and on the function. An alternative is to interpret the original function code list in a routine that directly computes numerical values for the higher-order derivatives.

We will give details of our derivative code list elsewhere. An interesting related treatment of the backward mode of automatic differentiation to obtain higher-order derivatives appears in [4].

7 Boxes and Lists for Nonlinear Equations and Optimization

Interval methods for nonlinear algebraic systems and global optimization involve exhaustive searches of the domain. See the algorithms in [14, §9.11], [26], [30, pp. 77–78], [35, p. 111], etc. In such exhaustive searches, the algorithm recursively subdivides an initial box $\mathbf{B}$, producing a list $\mathcal{L}$ of sub-boxes. As each box in $\mathcal{L}$ is processed, boxes are removed from and inserted into $\mathcal{L}$. Particularly in global optimization algorithms, elements of such lists are ordered by e.g. the lower bounds on the function value over the corresponding box.

We recapitulate the considerations in the introduction for software employing such lists:

1. Information other than the coordinates of a box $\mathbf{B}$, such as a corresponding approximate root or critical point, whether a computational existence test has proven existence of a root or critical point in $\mathbf{B}$, and bounds on the range of the objective function in the case of optimization, should be stored with the box. This information may change with the algorithm.

2. It should be possible to insert boxes into the list in logically separate parts of the algorithm. For example, generalized bisection may produce two boxes, one of which should be inserted into the list for later consideration; see [18], etc. Alternately, in the substitution-iteration process we first advocated in [22], computation of both branches of a square root of a strictly positive interval would also lead to two boxes. Furthermore, division by a zero-containing interval in an interval Gauss–Seidel step, with subsequent intersection of the extended intervals with the original interval, could lead to two intervals.

3. The maximum number of boxes in $\mathcal{L}$ is not known *a priori.* Furthermore, general software (even research codes) should be able to handle arbitrary dimensions efficiently with respect to ease of use, storage, and computation time.

4. It should be easy to modify research codes, easy to maintain production codes, and easy to read both types of code.

Such facts lead to the following conclusions.

- Information associated with the box **B** should be encapsulated in a derived data type. Simple support for creation and destruction of elements of this data type should be provided.

- Storage for the list $\mathcal{L}$ should be allocated and freed dynamically, and list operations should be encapsulated and made generic. This can be done with a dynamic linked list of the form described in [3, §8.2], with data of the type of **B** associated with each node.

We have created two box data types (one for optimization and one for general nonlinear equations), as well as list data types for integers and intervals. Operations on such lists include insertion, removal of the first element, a "purge" operation (for global optimization), printing, checking whether a list is empty, and creation. The list operations are *generic*, so that the same routine name can be used with a box of any type. For example, the statement

```
        CALL INSERT(L,B)
```

inserts item B into list L, where L is a list of any type, and B is an item of corresponding type.

Storage is created for a list element as the element is inserted, and is reclaimed as the element is removed. For boxes associated with interval vectors, the list routines read the dimension of the interval vector from the code list, and allocate precisely the amount of storage necessary.

The operations on lists make heavy use of Fortran 90 pointer variables.

Depending on hardware and compiler implementations, operations involving such dynamic allocation can involve large numbers of machine operations. However, operations on such lists within enclosure method codes typically occur infrequently compared to other computations, and are thus appropriate. In some contexts, such as when dealing with integer lists of known maximum size (e.g. variable indices), use of simple arrays may be more appropriate.

8 Summary, Conclusions, and Future Work

We have used new Fortran-90 capabilities to design a system for developing numerical nonlinear equation and global optimization codes. Features of this system include an interval data type and a special symbolic implementation of automatic differentiation that can be iterated. We have defined a special characteristic function to allow consideration of conditional branches. We have also supplied various routines to obtain numerical values from the symbolic lists produced from the automatic differentiation. Finally, we have supplied dynamic data structures for exhaustive search algorithms.

The new system should eliminate much of the programming burden when developing and testing both interval and non-interval nonlinear equation and optimization codes.

Future work can include improvement of the storage structures for higher derivatives, as mentioned in §6 above, and development of additional functions ϕ allowable in the code list. For example, we may provide for user-defined elementary functions. We may also allow linear forms like $\sum_{j=1}^{n_p} a_j x_{i_j}$, where the x_{i_j} are independent variables, since evaluation of such expressions leads to exact ranges.

9 Acknowledgement

I wish to acknowledge the referees and the editor John Reid for their quick but careful reading and for their useful suggestions.

References

[1] Alefeld, G. and Herzberger, J. *Introduction to Interval Computations*, Academic Press, New York, etc., 1983.

[2] Bleher, J. H., Rump, S. M., Kulisch, U., Metzger, M., Ullrich, C., and Walter, W. Fortran-SC — A Study of a Fortran Extension for Engineering Scientific Computation with Access to ACRITH. *Computing 39*, 2 (1987), 93–110.

[3] Brainerd, W. S., Goldberg, C. H., and Adams, J. C. *Programmer's Guide to Fortran 90*. Mc-Graw-Hill, New York, 1990.

[4] Christianson, B. Reverse Accumulation and Accurate Rounding Error Estimates for Taylor Series Coefficients. *Optimization Methods and Software 1*, 1 (1992), 81–94.

[5] Crary, F. The AUGMENT Precompiler. Technical report no. 1470, Mathematics Research Center, University of Wisconsin, Madison, 1976.

[6] Dixon, L. C. W., Maany, Z. and Mohseninia, M. Automatic Differentiation of Large Sparse Systems. *Journal of Economic Dynamics and Control 14*, (1990), 299–311.

[7] Floudas, C. A. and Pardalos, P. M. *A Collection of Test Problems for Constrained Global Optimization Algorithms.* Springer-Verlag, New York, 1990.

[8] Floudas, C. A. and Pardalos, Eds. *Recent Advances in Global Optimization*. Princeton Univ. Press, Princeton, N. J., 1992.

[9] Griewank, A. The Chain Rule Revisited in Scientific Computing. *SIAM News 24*, 3 (1991), 20–21.

[10] Griewank, A., The Chain Rule Revisited in Scientific Computing. *SIAM News 24*, 4 (1991), 8–24.

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

[12] Hager, G. Interval-Based Techniques for Sensor Data Fusion. Preprint, GRASP Lab – Room 301C, Univ. of Pennsylvania, 3401 Walnut St., Philadelphia, PA 19104-6228, 1990.

[13] Hager, G. D. Solving Large Systems of Nonlinear Constraints with Application to Data Modeling. *Interval Computations*, in press.

[14] Hansen, E. R. *Global Optimization using Interval Analysis*. Marcel Dekker, Inc., New York, 1992.

[15] Iri, M. and Kubota, K. Methods of Fast Automatic Differentiation and Applications. Technical report no. RMI 87-02, University of Tokyo, Department of Mathematical Engineering and Instrumentation Physics, 1987.

[16] Jan, C.-H. *Expression Parsing and Rigorous Computation of Bounds on All Solutions to Practical Nonlinear Systems.* Ph.D. dissertation, University of Southwestern Louisiana, 1992.

[17] Jansson, C. and Knüppel, O. A Global Minimization Method: The Multi-Dimensional Case. Preprint, T. U. Hamburg, 1992.

[18] Kearfott, R. B. Abstract Generalized Bisection and a Cost Bound. *Math. Comp. 49*, 179 (1987), 187–202.

[19] Kearfott, R. B. Interval Newton / Generalized Bisection When There are Singularities near Roots. *Annals of Operations Research 25*, (1990), 181–196.

[20] Kearfott, R. B. Preconditioners for the Interval Gauss–Seidel Method. *SIAM J. Numer. Anal. 27*, 3 (1990), 804–822.

[21] Kearfott, R. B., and Novoa, M. INTBIS, A Portable Interval Newton/Bisection Package (Algorithm 681). *ACM Trans. Math. Software 16*, 2 (1990), 152–157.

[22] Kearfott, R. B. Decomposition of Arithmetic Expressions to Improve the Behavior of Interval Iteration for Nonlinear Systems. *Computing 47*, (1991) 169–191.

[23] Kearfott, R. B., Hu, C. Y., Novoa, M. III. A Review of Preconditioners for the Interval Gauss–Seidel Method. *Interval Computations 1*, 1 (1991), 59–85.

[24] Kearfott, R. B. An Interval Branch and Bound Algorithm for Bound Constrained Optimization Problems. *Journal of Global Optimization 2*, (1992), 259–280.

[25] Kearfott, R. B., Dawande, M., Du K.-S. and Hu, C.-Y. INTLIB: A Portable Fortran-77 Elementary Function Library. Preprint, Department of Mathematics, University of Southwestern Louisiana, 1992.

[26] Kearfott, R. B. and Du, K. The Cluster Problem in Multivariate Global Optimization. Preprint, Department of Mathematics, University of Southwestern Louisiana, 1992.

[27] Kearfott, R. B., Dawande, M., Du K.-S. and Hu, C.-Y. INTLIB: A Reasonably Portable Interval Elementary Function Library. Preprint, Department of Mathematics, University of Southwestern Louisiana, 1992.

[28] Kearfott, R. B. and Shi, X. A Preconditioner Selection Heuristic for Efficient Iteration with Decomposition of Arithmetic Expressions for Nonlinear Systems. *Interval Computations*, in press.

[29] Kearfott, R. B. and Xing. Z. Rigorous Computation of Surface Patch Intersection Curves. Submitted to *Comput.-Aided Geom. Des.*

[30] Moore, R. E. *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, 1979.

[31] Neidinger, R. D. An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order. *ACM Trans. Math. Software 18*, 2 (1992), 159–173.

[32] Neumaier, A. *Interval Methods for Systems of Equations.* Cambridge University Press, Cambridge, England, 1990.

[33] Rall, L. B. *Automatic Differentiation: Techniques and Applications*. Springer, Berlin, New York, etc., 1981.

[34] Rall, L. B. An Introduction to the Scientific Computing Language Pascal-SC. *Computers and Mathematics with Applications 14*, 1 (1987), 53–59.

[35] Ratschek, H., and Rokne, J. *New Computer Methods for Global Optimization*. Wiley, New York, 1988.

[36] Snyder, J. M. Interval Analysis for Computer Graphics. *Computer Graphics 26*, 2 (1992), 121–130.

[37] Walster, G. W., Hansen, E. R. and Sengupta, S. Test Results for a Global Optimization Algorithm. In *Numerical Optimization 1984*, P. T. Boggs, R. H. Byrd and R. B. Schnabel, Eds., SIAM, 1985, 272–287.

[38] Walter, W. V. FORTRAN-XSC: A Portable Fortran 90 Module Library for Accurate and Reliable Scientific Computing. In *Computing Suppl. 9 (Validation Numerics)*, R. Albrecht, G. Alefeld and H. J. Stetter, Eds., 1993, 265–286.

[39] Walter, W. V. ACRITH-XSC: A Fortran-Like Language for Verified Scientific Computing. In *Scientific Computing with Automatic Result Verification*, E. Adams and U. Kulish, Eds., Academic Press, New York, etc., 1993.

[40] Wengert, R. E. A simple Automatic Derivative Evaluation Program. *Comm. ACM 7*, 8 (1964), 463–464.

[41] Yohe, J. M. Software for Interval Arithmetic: A Reasonably Portable Package. *ACM Trans. Math. Software 5*, 1 (1979), 50–53.