Interval Arithmetic in Forte™ Fortran

Technical White Paper
## Contents

1. **Introduction** ................................................................. 1
2. **Interval Arithmetic Fundamentals and Compiler Technology** ...... 3
   Introduction .................................................................. 3
   Containment ............................................................. 4
   Dependence ................................................................ 5
3. **Extended Intervals** .......................................................... 7
   Notation ..................................................................... 9
   The Containment Set .................................................... 14
   Interval Expression Evaluation ..................................... 16
   The Containment-Set Closure Identity ......................... 16
   The Fundamental Theorem of Interval Arithmetic .......... 19
   Expression Evaluation ................................................. 22
   Containment-Set-Equivalent Expressions ..................... 22
   Interpreting Empty Results ......................................... 23
   Continuity .................................................................. 26
   Summary ................................................................. 30
4. **Ease-of-Use** ................................................................. 31
   Context-Dependent Interval Constants ....................... 31
   Mixed-Mode Interval Expression Processing ................ 32
   Strict-Expression Processing ....................................... 32
Forwards

This Sun Microsystem Technical White Paper introduces powerful, intrinsic language support for intervals, based on the containment-set closure identity proved in *Extended Real Intervals and the Topological Closure of Extended Real Relations*, by G. W. Walster, E. R. Hansen and J. D. Pryce [1].

In this new system, no undefined events (called exceptions), are logically possible. This is an important new result. The implementation of this new system by Sun represents a major advance in machine computation.

Using a two-point compactification of the real line, and the new concept of containment sets (proven to be equivalent to closures), a logically infallible approach to computer implementation of interval arithmetic on the extended real line is developed.

The Interval Arithmetic system in Forte™ Developer Fortran has many features giving it flexibility for a wide variety of uses, including:

- exception-free expression processing using the closed interval system,
- context-dependent interpretation of empty intervals,
- automating the proof of expression continuity (forthcoming),
- mixed-mode expression evaluation,
- single number input/output, and
- certainly-, possibly-, and set-order relations.

This technical white paper introduces the above innovations to members of the interval research and algorithm developer communities.
With this work, Sun is leading the way to new directions in computing technology. Using this new compiler, we can move forward in such important areas as globally convergent methods for solving nonlinear optimization problems on networks, using a distributed processing environment. Building on the decades of previous mathematical groundwork and this new compiler, all the tools are now at hand for moving beyond what is possible with ordinary floating-point arithmetic.

Ramon Moore
Interval Consultant
Scientific Computing has become a major tool for all sciences. The speed of modern computers enables users to deal with more and more complicated problems. Today, even problems that were unsolvable 15 years ago can be solved within a few seconds. In all cases, however, the computed solution necessarily deviates more or less from the correct solution. A simultaneous or an a posteriori computation of the error, therefore, is highly desirable. Interval arithmetic is indispensable to reach this goal, and provides mathematical rigour in numerics.

It certainly is a tremendous service to all sciences that now a leading vendor provides interval arithmetic within its Fortran programming environment. Without the necessary programming comfort, interval arithmetic is hardly usable.

Conventional methods of numerical analysis often have an elegant counterpart or extension in interval analysis. An example is Newton’s method. It is the key method for non-linear problems. The method converges quadratically to the solution if the initial value of the iteration is already close enough. However, it may fail in finite as well as in infinite precision arithmetic even in the case of only one single solution in a given interval. In contrast to this, the interval version of Newton’s method is globally convergent. It never fails — not even in floating-point arithmetic.

In 1969, the interval version of Newton’s method was extended in such a way that it computes enclosures of all (single) zeros in a given interval (T33, ZAMM 1970). With this extension, Newton’s method reaches its final strength. It is a major tool for many fascinating applications. The key operation to achieve this is division by an interval which contains zero. It is most pleasant that Sun’s Forte Developer Fortran even provides this extremely useful operation. The system arithmetic on the completed set of real numbers is defined without any exceptions.

Interval analysis has been developed to a high standard during the last decades. It is not a trivial subject. Its properties have to be studied before it can be successfully applied. This is a new challenge for computer users. Teaching will ease this task.
With Sun’s Forte Developer Fortran interval arithmetic is now conveniently available on computers that are widespread. This is a great step ahead — a step which was long overdue. The system provides considerable and outstanding support for scientific computing in general, and for numerical analysis in particular. With it a major breakthrough in different corners of the sciences can very well be expected.

Ulrich Kulisch
Professor of Mathematics
Universitaet Karlsruhe
Introduction

Background

Since R. E. Moore established the mathematical foundation and utility of interval arithmetic nearly a half-century ago, stunning advances in computing have occurred. In particular, high-speed computing resources are now readily available, and floating-point software is more portable due to the near universal acceptance of the IEEE-754 floating-point standard. Despite these and other advances, the dominant floating-point computing paradigm remains unchanged.

While the fundamental character of non-interval, floating-point algorithms has not changed since IBM introduced floating-point numbers in 1954, today interval algorithms do more than bound rounding errors in floating-point computations — they solve a variety of nonlinear problems. With object-oriented languages to support interval C++ classes and Fortran modules, interval arithmetic is poised to become core computing technology. However, commercial interval applications are required if interval arithmetic is to become mainstream.

Interval applications require the stability and features provided by intrinsic data types. Because intrinsic interval support is missing from today’s Fortran, C, C++, and Java™ development environments, few commercial interval applications have been developed. Once compilers support intervals, commercial applications and hardware will follow, enabling mainstream users to reap the inherent benefits that only intervals offer.
New Interval Features in Forte Developer Fortran

By taking the lead to introduce intrinsic language support for intervals in Forte Developer Fortran (f95), Sun Microsystems has taken the opportunity to advance the quality of language support for interval data types. This document describes the new interval support features implemented in f95. While all innovations are designed to increase the ease with which developers can write interval code, introducing mathematically closed interval systems has implications both for interval and real analysis. The underlying closed set-based system and the resulting closed interval systems enjoy properties that neither real nor extended real number systems possess.

Sun’s Commitment

Sun has been instrumental in developing state-of-the-art technical computing systems for more than 15 years. Toward that end, Sun has maintained an industry leadership position in many technology areas. Today, Sun continues this tradition with its unbridled support for interval arithmetic. While the benefits of interval arithmetic are clear, knowledge of its function and impact is lacking in much of the engineering, scientific, and consumer populations. Sun knows interval arithmetic is important and is engaged in partnerships with universities and other institutions to establish it as a widely used and appreciated core computing technology.
Introduction

By computing rigorous numerical bounds, interval arithmetic is able to unambiguously answer questions that are otherwise difficult, if not impossible to numerically answer. Arguably the two most important consequences of intervals and interval arithmetic are:

- rigorous bounds on errors from all sources and their interaction, including input data and rounding errors; and
- numerical solutions to nonlinear problems, including nonlinear systems of equations, nonlinear programming, and ordinary differential equations.

These consequences logically follow from the fact that computed intervals include the totality of all possible outcomes of any given computation. This interval design requirement is referred to as the containment constraint — the one and only requirement that a valid interval computing system must satisfy. Failure to satisfy the containment constraint precludes the possibility of computing rigorous numerical bounds, and is therefore a fatal interval system error.

Given the requirement to satisfy the containment constraint, compilers supporting interval data types can be distinguished by the quality of different implementation components, such as:

- transparent language syntax and semantics,
- speed,
- interval width,
• reliability, and
• interval-specific development tools.

Quality of implementation in an interval context is any feature, including speed, interval width, and ease-of-use, but not containment. Containment is a constant, not a goal. Quality of interval implementation features documented in the remainder of this white paper include:

• extensions to the foundation of intervals (following a very brief description of classical interval arithmetic, as introduced by Moore); and
• various compiler ease-of-use features, including:
  • context-dependent interval constants,
  • mixed-mode interval expression processing,
  • interval-specific relational operators, and
  • single-number interval input/output.

Although not yet an interval support feature in f95, a section is devoted to verifying that an expression is continuous over the subset of its domain defined by argument intervals.

Containment

In Interval Analysis [4], Moore defined interval arithmetic for finite rational functions composed of the four basic arithmetic operations (BAOs) on finite real numbers. Analogous to the situation with real numbers, division by intervals containing zero was not defined. Originally, neither was raising an interval to an integer power. The sets of values that finite interval operations must contain are defined in classical interval arithmetic as introduced by Moore and briefly described below. Extensions are described in Chapter 3 of this document.

Let \([a, b]\) denote a real, closed, compact interval constant. That is, for example:

\[
[a, b] = \{ x \in \mathbb{R} \mid a \leq x \leq b \}
\]

where \(\mathbb{R}\) denotes the set of finite real numbers, \([x \mid -\infty < x < +\infty]\). If \(op \in \{+, -, \times, \div\}\) denotes one of the four basic arithmetic operators, then arithmetic operations on a pair of intervals, \([a, b]\) and \([c, d]\) must produce a new interval, say \([e, f]\), such that:

\[
[e, f] \supseteq \{ x \ op \ y \mid x \in [a, b] \text{ and } y \in [c, d]\}
\] (1)
where \( d < 0 \) or \( 0 < c \) in the case of division, because division by zero is not defined for real numbers. The right-hand side of Equation (1) defines the sets of values that interval arithmetic operations on finite real intervals must contain. While computing narrow intervals is desirable, the only requirement is containment. The right-hand side of equation (1) is the set of values that the operation, \( x \text{ op } y \) must contain. The term containment set is used to describe the set of values that an interval result must contain.

Arithmetic operation monotonicity makes possible the following rules to compute finite interval arithmetic operation endpoints:

\[
[a, b] + [c, d] = [a + c, b + d] ,
\]

\[
[a, b] - [c, d] = [a - c, b - d] ,
\]

\[
[a, b] \times [c, d] = [\min(a \times c, a \times d, b \times c, b \times d), \max(a \times c, a \times d, b \times c, b \times d)] ,
\]

and

\[
[a, b] \div [c, d] = [\min(a \div c, a \div d, b \div c, b \div d), \max(a \div c, a \div d, b \div c, b \div d)].
\]

where, to exclude division by intervals containing zero, \( d < 0 \) or \( 0 < c \). Directed rounding is used to guarantee containment when interval operations are implemented using IEEE-754 floating-point operations.

**Dependence**

Within a single expression, interval arithmetic treats each occurrence of a given interval variable as if it were either an interval constant or a new independent variable. Multiple occurrences of the same variable are therefore not recognized by interval arithmetic. This is known as the dependence problem of interval arithmetic, because the failure to use dependence information can cause interval results to be unnecessarily wide.

The interval evaluation of a finite rational expression, in which each variable appears only once, was first proved by Moore [4] to produce an exact bound on the set of possible results. Any expression in which each variable appears only once is called a single-use expression, or SUE. When computing an expression using interval arithmetic, the form of the expression — if it is not a SUE — can dramatically impact the result’s width. For example, in the two expressions:

\[
(a + b)x
\]

and

\[
ax + bx
\]
replacing the points, \( a, b, \) and \( x \), by the intervals, \( A = X, B = [-10, -10] \), and \( X = [10, 20] \), results in the following values of the corresponding interval expressions:

\[
(A + B)X = [0, 0] \times [10, 20] = [0, 0]
\]  

and

\[
AX + BX = [100, 200] - [100, 200] = [-100, 100].
\]  

Evaluating expression (2) produces an interval result that is as narrow as possible — that is, *sharp*. Expression (3) does not always produce sharp results because it contains multiple occurrences of the interval variable, \( X \) that interval arithmetic treats as independent. If expression (3) were instead:

\[
(4)
\]

with \( A, B, \) and \( X \) as in (3), and with \( Y = [10, 20] \), the result, \([-100, 100]\) is sharp.

The *dependence problem of interval arithmetic* is the name given to the fact that interval arithmetic treats an expression with multiple occurrences of the same variable, such as expression (3), as if it were a SUE, like expression (4), instead of a SUE, like expression (2).

Interval code developers quickly learn to use SUEs whenever they exist. In other cases, there can be more than one SUE form from which to choose. If a SUE contains a division, it can be problematic if the denominator interval contains zero. Another difficulty is that most expressions have no SUE form. Division by intervals containing zero and other indeterminate forms is conveniently handled using the closed interval system described in the next section. Without loss of containment, containment-set equivalence is a fail-safe test for expression substitution. Methods exist for computing narrow bounds on the range of non-SUE expressions over large boxes. These include:

- the interval version of Newton's algorithm,
- interval nonlinear global optimization, and
- high order Taylor multinomials.

However, work is needed to develop more general methods that work for highly coupled expressions and scale up to the problem sizes that are sometimes seen in commercial applications.
Extended Intervals

In *A More Complete Interval Arithmetic* [3], Kahan was the first to recognize that division by an interval containing zero is possible if interval endpoints are extended real numbers, including the ideal values \(-\infty\) and \(+\infty\). The interval:

\[
[-\infty, \infty] = \{x | -\infty \leq x \leq \infty\}
\]

\[= \mathbb{R}^*\]

is the set of all possible extended real numbers. Therefore, \(\mathbb{R}^*\) contains the result of division by zero or any operation on extended real values. Returning the extended interval \(\mathbb{R}^*\) also bounds the result of interval operations that include indeterminate forms, such as \(\infty - \infty\), \(0 \times \infty\), \(\infty \div \infty\), or \(0 \div 0\). Using \(\mathbb{R}^*\) is required to construct interval systems that are mathematically closed. Any system that always produces defined outcomes contained in the system is said to be closed. In a closed interval system, a valid interval result is produced by any operation or expression using any interval arguments. The term *expression* refers to any code list that can be implemented on a computer, including both arithmetic operations and library functions. In a closed interval system, defensive code to avoid undefined outcomes, referred to as *exceptions*, is unnecessary. Therefore, compiler support for interval data types in a closed interval system does not require interval exception handling.

While returning \(\mathbb{R}^*\) can never be a containment failure, \(\mathbb{R}^*\) may not be a sharp result. To know whether it is possible to return a narrower interval than \(\mathbb{R}^*\), the containment set of division by zero and indeterminate forms must be defined.
An indeterminate form occurs when a function is evaluated at a point that is on the border of the function’s open domain. For example, the natural domain of division, \( x / y \) is the set:

\[
\{(x, y) \mid x \in \mathbb{R}, y \in \mathbb{R} - \{0\}\}
\]  

(5)

Because of rounding errors and dependence, it is common for interval results to be outside the domain of a subsequent operation or function. For example, it is not an error if the result of Equation (3) is the argument of the square root function. If every result that is partially outside the finite domain of a subsequent operation or function is treated as an error, defensive code is required to avoid false-alarm exceptions.

Even if \( \mathbb{R}^* \) is returned when an indeterminate form is encountered, sharp results can be computed. For example, the expression:

\[
\frac{x}{x + y}
\]

(6)

can be rewritten as the SUE:

\[
\frac{1}{1 + \frac{y}{x}}
\]

(7)

Because the variables \( x \) and \( y \) appear only once in Equation (7), interval evaluation produces a sharp result. The interval evaluation of (6) may not. However, the domains of these two expressions are not the same. Expression (6) is not defined if \( x + y = 0 \), whereas expression (7) is not defined if either \( x = 0 \) or \( x = -y \neq 0 \).

The interval versions of (6) and (7) are:

\[
\frac{X}{X + Y}
\]

(8)

and

\[
\frac{1}{1 + \frac{Y}{X}}
\]

(9)

with the restriction in (8) that \( 0 \not\in X + Y \) and in (9) that \( 0 \not\in 1 + \frac{Y}{X} \) and \( 0 \not\in X \).

While (9) produces a sharp result if it is defined, it is not defined, for example, if \( X = [-1, 1] \), and \( Y = [2, 3] \). Yet in this case (8) produces:
but this result still is not sharp. The sharp result, in this case \([-1, \frac{1}{3}]\), is produced by the SUE:

\[
1 - \frac{1}{1 + \frac{Y}{X}}.
\]  

(11)

For expression (11) the domain restrictions are that \(0 \notin 1 + \frac{Y}{X}\) and \(0 \notin Y\).

In a closed interval system, even if \(\mathbb{R}^+\) is returned whenever division by zero occurs, expressions (9) and (11) can be combined:

\[
\left(\frac{1}{1 + \frac{Y}{X}}\right) \cap \left(1 - \frac{1}{1 + \frac{X}{Y}}\right).
\]  

(12)

Unless both (9) and (11) return \(\mathbb{R}^+\), expression (12) is sharp.

Even when both (9) and (11) return \(\mathbb{R}^+\), expression (12) produces the single containing interval with the narrowest possible width. Among all possible single intervals, this result is sharp. In some cases, multiple intervals are required to exactly represent the containment set of expression (12).

This example raises a more general question: When is it legitimate to substitute one interval expression for another? Is substitution possible when the enclosed expressions have different domains? The analysis of this and other questions is facilitated by introducing the closed \(\mathbb{R}^+\)-system in which the containment set of an interval expression is always defined.

**Notation**

The analysis of containment sets includes many opportunities for notation ambiguity. The following notation is adopted and strictly enforced to clearly distinguish between points, sets and intervals, and between variables and their values. To facilitate the exposition, most results are stated without proof. Mathematical details and proofs are contained in [1].

1. A function is a particular kind of relation. Evaluating a function at a point is a special case of evaluating a relation over a subset of its source space, producing a subset of its target space. The basic arithmetic operations in the extended system are relations. Evaluating an expression is always on set-
valued arguments producing a set-valued result. Normal function evaluation occurs when evaluation at a singleton-set \( \{x_0\} \) produces a singleton \( \{y_0\} \). Then, and only then, is the notation \( y_0 = f(x_0) \) used.

2. The extended real number system is a compact topological space \( \mathbb{R}^* \) containing \( \mathbb{R} \). Each extended basic arithmetic operation is the topological closure in \( (\mathbb{R}^*)^n \) of (the graph of) the corresponding operation regarded as a subset of \( \mathbb{R}^3 \). The \( \mathbb{R}^* \)-system is proved in [1] to be consistent by deploying standard results on evaluation, composition, etc. of closed relations over compact spaces.

3. The containment set of a relation evaluated at a point can be disconnected compact sets, not necessarily a single interval. The defining properties and the definition of the containment set of an expression are contained in the next section.

4. The subscript, 0, is used to denote a specific value of a variable. For example, \( x_0 = y_0 \) denotes the fact that the specific values, \( x_0 \) and \( y_0 \), of the variables \( x \) and \( y \) are the same. For example, if \( x_0 = 2 \) and \( y_0 = 2 \), then \( x_0 = y_0 \). The fact that \( x_0 = y_0 \) does not necessarily imply that the variables \( x \) and \( y \), or the expressions they represent, are identical. Consequently, different occurrences of the variables \( x \) and \( y \) cannot be interchanged only because \( x_0 = y_0 \). However, the equation \( x = y \) implies the variables themselves are interchangeable. The consequence of the distinction between value and variable or expression equality is illustrated in the following three examples.

a. Given two variables, \( x \) and \( y \), with given values \( x_0 = 2 \) and \( y_0 = 1 \), then
\[
\frac{x_0}{y_0} = \frac{2}{1} = 2.
\] However, if \( x_0 = 0 \) and \( y_0 = 0 \), then \( \frac{x_0}{y_0} \) is undefined.

Nevertheless, if \( x = 2y \), then even though \( x_0 = 0 \) and \( y_0 = 0 \), \( \frac{x_0}{y_0} = \frac{2y_0}{y_0} = 2 \). In this case, \( x \) and \( y \) are dependent.

b. If \( x_0 \) and \( y_0 \) are the limits of sequences \( (x_i) \) and \( (y_i) \) and if \( x_i = 2y_i \), then
\[
\lim_{i \to \infty} \frac{x_i}{y_i} = 2.
\] If the two sequences are unrelated, then it will be shown below that the containment set of the expression \( \frac{y_0}{y_0} = \frac{0}{0} \) is \( (-\infty, +\infty) \).

c. Given intervals \( X_0 = [a, b] \) and \( Y_0 = [c, d] \), then without information regarding the relationship between the interval variables \( X \) and \( Y \),
\[
X_0 - Y_0 = [a-b, b-a].
\] However, if \( X = Y \), then \( X - Y = X - X = 0 \).
5. Customary notation in both the mathematical and interval literature uses upper case letters to denote sets and intervals, respectively. When working with sets and intervals together, they must be distinguished. Because this paper is primarily concerned with intervals rather than sets, all sets are enclosed in braces to distinguish them from intervals. Therefore, in this paper $X$ is the closed interval,

$$X = [\underline{X}, \overline{X}]$$

(13)

$$= \{ z \mid \underline{X} \leq z \leq \overline{X} \}$$

(14)

The set $\{X\}$ is not necessarily an interval, although it can be one. Singleton sets are denoted using lower case letters and are also enclosed in braces to distinguish them from points and degenerate intervals. Degenerate intervals are enclosed in brackets. The singleton set, $\{x\}$, and the degenerate interval $[x]$ each have only one element, the point, $x$. In summary,

- $X$ is an interval,
- $[x] = [x, x]$ is a degenerate interval,
- $\{X\}$ is a compact set, and
- $\{x\}$ is a singleton set.

6. Bold letters are used to represent vectors of points, sets, and intervals. In particular,

$$\mathbf{x} = (x_1, \ldots, x_n) ,$$

(15)

$$\{\mathbf{x}\} = (\{x_1\}, \ldots, \{x_n\}) ,$$

(16)

$$\{\mathbf{X}\} = (\{X_1\}, \ldots, \{X_n\}) ,$$

(17)

$$[\mathbf{x}] = ([x_1], \ldots, [x_n]) , \text{ and}$$

(18)

$$\mathbf{X} = (X_1, \ldots, X_n) .$$

(19)

are respectively:

a. a point in $n$-dimensional Euclidean space, $\mathbb{R}^n$;

b. a singleton set, the only element of which is a point in extended $n$-dimensional Euclidean space $(\mathbb{R}^*)^n$;

c. the Cartesian product, $\{X_1\} \otimes \ldots \otimes \{X_n\} \in (\mathbb{R}^*)^n$, of the sets, $\{X_i\}$;

d. a degenerate interval, the only element of which is a point in extended $n$-dimensional Euclidean space; and

e. an $n$-dimensional box, $X_1 \otimes \ldots \otimes X_n \in (\mathbb{R}^*)^n$, of the intervals, $X_i$.  

May 2000

11
7. An expression is any computation defined by the execution of a code list. The following are important to distinguish:

a. a segment of computer code,

b. the expression defined when this code segment is executed, and

c. the relation or function defined by this expression.

Irrespective of whether the code includes branches, loops and subprogram calls, array references, or overwriting of a variable’s value by a new value, any particular execution is a finite code list of operations. For instance, if the input is \((x_1, x_2)\), and the code list is:

\[
\begin{align*}
  x_3 &= x_1 + x_2 \\
  x_4 &= x_2 / x_3 \\
  x_5 &= x_4 + x_3
\end{align*}
\]

where the output is \(x_5\), then:

\[
f(x_1, x_2) = \frac{x_2}{x_1 + x_2} + (x_1 + x_2).
\]

The function value \(f(x_0)\) exists at the point \(x_0 = (x_{01}, \ldots, x_{0n})\) in \(\mathbb{R}^n\) iff, for each compute step in the code list, the arguments to each basic operation lie in the operation’s domain. The set \(D\) of such \(x_i\) is called \(f\)'s natural domain provided operators and intrinsic function’s domains are used to define the domain of \(f\) in \(\mathbb{R}^n\). If only the four basic arithmetic operations (BAOs) are used, then every \(f\) defines a rational function of the inputs. In the case of rational functions, \(D\) is precisely those points \(x\) for which divide-by-zero and infinite operations do not occur while evaluating \(f\) at \(x_0\).

8. Brackets and braces establish context for the interpretation of symbols used to represent expressions defined by code lists. For example, when evaluated at the point \(x_i\), the singleton set \(\{x_i\}\), or the degenerate interval \([x_i]\), the expression \(f\) is represented by:

- \(f(x_i)\), the function, \(f\), evaluated at the point, \(x_i \in D\); the domain of \(f\);
- \(f(\{x_i\})\), the relation, \(f\), evaluated at the singleton set \(\{x_i \in \mathbb{R}^n\}\); and
- \(f([x_i])\) (or equivalently \(f([x_i], \ldots, x_i])\), the interval evaluation of the expression, \(f\), at the degenerate interval \([x_i \in \mathbb{R}^n]\).
Because a code list evaluation can produce either a single value or multiple values, the neutral term expression rather than function is used to refer to the object of a code list evaluation. The development in [1] extends the mathematical foundation under interval arithmetic by using the set-theoretic properties of intervals to derive the containment set of an expression that can be either a function or a relation.

9. The closure of the compact set \( \{X\} \), is \( \overline{X} \), and includes all possible accumulation points formed from any subsequence of any sequence, \((x_j)\), the terms of which are elements of the set \( \{X\} \). That is:

\[
\overline{X} = \left\{ z \mid \begin{align*}
z &= \lim_{j \to \infty} y_j \\
x_j &\in \{X\}
\end{align*} \right\}
\]

where \( \lim_{j \to \infty} x_j \) is the limit of any subsequence of terms in the sequence \((x_j)\). Because the set \( \{X\} \) is compact, it is guaranteed to contain at least one accumulation point. The narrowest interval, \( X \), that contains all the elements in the set, \( \{X\} \), is the interval hull, denoted:

\[
hull(\{X\}) = [\inf(\{X\}), \sup(\{X\})].
\]

10. The closure of the expression \( f \) with compact domain, \( D_f \), evaluated at the point \( x_0 \) is denoted \( \overline{f}(\{x_0\}) \), and is defined: if \( x_0 \in \overline{D}_f \), then

\[
\overline{f}(x_0) = \left\{ z \mid \begin{align*}
z &= \lim_{j \to \infty} y_j \\
y_j &\in f(\{x_j\}) \\
x_j &\in D_f \\
&= \lim_{j \to \infty} x_j \\
x_j &\in x_0
\end{align*} \right\}
\]

Otherwise, if \( x_0 \notin \overline{D}_f \), then

\[
\overline{f}(\{x_0\}) = \emptyset.
\]

The closure of \( \overline{f} \) is always defined, but may be the empty set. The domain of \( \overline{f} \) is the set of argument values for which \( \overline{f}(\{x_0\}) \neq \emptyset \), or equivalently the closure of the domain of \( f \):

\[
\overline{D}_f = \overline{D}_f = \left\{ x_0 \mid \begin{align*}
x_0 &= \lim_{j \to \infty} x_j \\
x_j &\in D_f
\end{align*} \right\}.
\]
Given the conditions on the right-hand side of (26) are satisfied, the closure of \( f \) at the point \( x_0 \) is the set of all possible accumulation points in the subsequences whose members are elements of the sets, \( f(x_j) \). If \( x_0 \in D_f \), all \((x)\)-sequences have the common accumulation point, \( x_0 \). No restrictions are imposed on the point \( x_0 \).

11. As with points (see item 6 on page 11), expressions of point-, interval-, and set-vector arguments mean different things. For example, \( f(x_0), f(x_0), \) and \( f([x_0]) \), are respectively:
   a. the function, \( f \), evaluated at the point \( x_0 \in D_f \subseteq \mathbb{R}^n \),
   b. the relation, \( f \), evaluated at the singleton set \( \{x_0\} \), given the point, \( x_0 \in (\mathbb{R}^n)^n \), and
   c. the interval evaluation of the expression, \( f \), at the degenerate interval \([x_0]\), given the point, \( x_0 \in (\mathbb{R}^n)^n \).

As discussed, in Section 1, the definition of \( f([x]) \) in the interval literature is normally limited to single-valued functions with domain \( D_f \subseteq (\mathbb{R}^n)^n \). Although complex intervals are considered in the interval literature, complex variables and intervals are not considered here or in [1].

12. Because \( \{X\} \) denotes a set of points in \( n \)-dimensional space, the notation \( f(\{X\}) \) denotes the relation, \( f \), evaluated over all singleton sets, \( \{x\} \in \{X\} \) that is:

\[
f(\{X\}) = \left\{ z \mid z \in f(\{x\}), x \in \{X\} \right\}.
\]  

(29)

13. Because \( X \) is a box in \( n \)-dimensional space, if \( X \subseteq D_f \), the notation \( f(X) \) denotes an interval that must be an enclosure of

\[
\left\{ z \mid z \in f([x]), x \in X \right\}.
\]

(30)

The question to be answered is: What is the containment set of values that \( f(X) \) must enclose, if \( X \not\subseteq D_f \)?

The Containment Set

Let \( h([x]) \) be an expression of \( n \)-variables. When the expression, \( h \), is evaluated at the point \( x \), the result, \( h([x]) \), is a set of values that may or may not be a singleton set. When these values are used as arguments in other expressions, additional sets of values are produced. For example, in:
Consider some set, \( \{Y\} \not\subseteq h((x_0)) \). That is, \( \{Y\} \) does not contain all the elements in the set \( h((x_0)) \). As a consequence, it is possible that for some \( g \),

\[
(32) 
\]

If there exists an expression, \( g \) such that (32) is true, then the set, \( \{Y\} \), has caused a containment failure. Why not simply identify the containment failure of \( \{Y\} \) with the relation:

\[
(33) 
\]

If \( h((x_0)) \) is defined, (33) is indeed a containment failure. However, if \( h \) is not defined at the point, \( x_0 \), neither are the values that must be included in the containment set. The problem is to define the containment set of the expression, \( h \) at points, \( x_0 \) where \( h((x_0)) \) is undefined. This is done by defining the containment set of \( h \) to be the smallest set, \( Y \), that can never cause a subsequent containment failure. Begin by examining the composite expression:

\[
(34) 
\]

Given \( h \), consider the set of all possible compositions of the form in (34). Denote the containment set of the expression, \( f \), evaluated at \( x_0 \): \( \text{cset}(f,(x_0)) \). The set of values in \( \{Y\} \) causes a containment failure to occur if there exist both a composition having the form in (34) and a point, \( x_0 \), for which

\[
(35) 
\]

Equation (35) is a containment failure because \( \text{cset}(g((Y),(x_0))) \) fails to contain all the elements in \( \text{cset}(f,(x_0)) \). This conception of a containment failure is motivated by the following considerations:

- Equation (35) is the essential event that the constraint on containment sets is designed to prevent. That is, when used as an argument of any subsequent expression, a containment set must not cause a subsequent containment failure.

- Equation (35) is the basis for defining the containment constraint that containment sets must satisfy.
Because it begs the question of what containment sets are, equation 35 does not constrain containment sets from including arguments at singular points and indeterminate forms.

**Definition 1.** The containment constraint on the set of values \( Y_0 \), of the expression \( h \) of \( n \)-variables, evaluated at the point, \( x_0 \), and denoted \( cset(h, \{ x_0 \}) \), is that:

\[
\text{cset}(f, \{ x_0 \}) \subseteq \left\{ z \mid z \in g\{ (y_0, x_0) \} \right\}
\]

using any possible composition of the form,

\[
f(x) = g\{ (y, x) | y \in h(x) \}
\]

and any \( x \in (\mathbb{R}^n)^n \), for which

\[
cset(f, \{ x_0 \}) \neq \emptyset.
\]

A trivial way to satisfy the containment constraint and therefore avoid containment failures is to let \( Y_0 \) in Definition 1, and therefore \( cset(h, \{ x_0 \}) \), be the entire set of extended real numbers \( \mathbb{R}^n \). Because unnecessary members of \( cset(h, \{ x_0 \}) \) are not wanted, the containment set of \( h \) at \( x_0 \) must be the smallest set that satisfies the containment constraint in Definition 1.

The interval evaluation of an expression \( f \) at \( x_0 \), denoted \( f([x_0]) \), must satisfy:

\[
f([x_0]) \supseteq \text{hull}(cset(f, \{ x_0 \}))
\]

where \( \text{hull}(\cdot) \) is the interval hull of the set of elements in its argument. The value of \( f([x_0]) \) must be an enclosure of the containment set of \( f \) to preclude subsequent containment failures.

**The Containment-Set Closure Identity**

The containment-set closure identity establishes that containment sets are simply closures. The identity is proved in [1].

**Theorem 2.** Given any expression \( f([x]) \) of \( n \)-variables and the point, \( x_0 \), then:

\[
cset(f, \{ x_0 \}) = \tilde{f}(\{ x_0 \}).
\]

The identity of containment sets and closures uniquely defines the set of values that the interval evaluation of an expression must contain. That is, (39) becomes

\[
f([x_0]) \supseteq \tilde{f}(\{ x_0 \}) = cset(f, \{ x_0 \}).
\]
### Table 3-1: Containment set for addition: $cset(x+y, \{(x_0, y_0)\})$

<table>
<thead>
<tr>
<th></th>
<th>${-\infty}$</th>
<th>${\text{real: } y_0}$</th>
<th>${+\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${-\infty}$</td>
<td>${-\infty}$</td>
<td>${-\infty}$</td>
<td>$\mathbb{R}^*$</td>
</tr>
<tr>
<td>${\text{real: } x_0}$</td>
<td>${-\infty}$</td>
<td>${x_0 + y_0}$</td>
<td>${+\infty}$</td>
</tr>
<tr>
<td>${+\infty}$</td>
<td>$\mathbb{R}^*$</td>
<td>${+\infty}$</td>
<td>${+\infty}$</td>
</tr>
</tbody>
</table>

### Table 3-2: Containment set for subtraction: $cset(x-y, \{(x_0, y_0)\})$

<table>
<thead>
<tr>
<th></th>
<th>${-\infty}$</th>
<th>${\text{real: } y_0}$</th>
<th>${+\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${-\infty}$</td>
<td>$\mathbb{R}^*$</td>
<td>${-\infty}$</td>
<td>${-\infty}$</td>
</tr>
<tr>
<td>${\text{real: } x_0}$</td>
<td>${+\infty}$</td>
<td>${x_0 - y_0}$</td>
<td>${-\infty}$</td>
</tr>
<tr>
<td>${+\infty}$</td>
<td>${+\infty}$</td>
<td>${+\infty}$</td>
<td>$\mathbb{R}^*$</td>
</tr>
</tbody>
</table>

### Table 3-3: Containment set for multiplication: $cset(x \times y, \{(x_0, y_0)\})$

<table>
<thead>
<tr>
<th></th>
<th>${-\infty}$</th>
<th>${\text{real: } y_0} &lt; 0$</th>
<th>${0}$</th>
<th>${\text{real: } y_0} &gt; 0$</th>
<th>${+\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${-\infty}$</td>
<td>${+\infty}$</td>
<td>${+\infty}$</td>
<td>$\mathbb{R}^*$</td>
<td>${-\infty}$</td>
<td>${-\infty}$</td>
</tr>
<tr>
<td>${\text{real: } x_0 &lt; 0}$</td>
<td>${+\infty}$</td>
<td>${x \times y}$</td>
<td>${0}$</td>
<td>${x \times y}$</td>
<td>${-\infty}$</td>
</tr>
<tr>
<td>${0}$</td>
<td>$\mathbb{R}^*$</td>
<td>${0}$</td>
<td>${0}$</td>
<td>${0}$</td>
<td>$\mathbb{R}^*$</td>
</tr>
<tr>
<td>${\text{real: } x_0 &gt; 0}$</td>
<td>${-\infty}$</td>
<td>${x \times y}$</td>
<td>${0}$</td>
<td>${x \times y}$</td>
<td>${+\infty}$</td>
</tr>
<tr>
<td>${+\infty}$</td>
<td>${-\infty}$</td>
<td>${-\infty}$</td>
<td>$\mathbb{R}^*$</td>
<td>${+\infty}$</td>
<td>${+\infty}$</td>
</tr>
</tbody>
</table>

### Table 3-4: Containment set for division: $cset(x \div y, \{(x_0, y_0)\})$

<table>
<thead>
<tr>
<th></th>
<th>${-\infty}$</th>
<th>${\text{real: } y_0} &lt; 0$</th>
<th>${0}$</th>
<th>${\text{real: } y_0} &gt; 0$</th>
<th>${+\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${-\infty}$</td>
<td>${0, +\infty}$</td>
<td>${+\infty}$</td>
<td>${-\infty, +\infty}$</td>
<td>${-\infty}$</td>
<td>${-\infty, 0}$</td>
</tr>
<tr>
<td>${\text{real: } x_0 \neq 0}$</td>
<td>${0}$</td>
<td>${x \div y}$</td>
<td>${-\infty, +\infty}$</td>
<td>${x \div y}$</td>
<td>${0}$</td>
</tr>
<tr>
<td>${0}$</td>
<td>${0}$</td>
<td>${0}$</td>
<td>$\mathbb{R}^*$</td>
<td>${0}$</td>
<td>${0}$</td>
</tr>
<tr>
<td>${+\infty}$</td>
<td>${-\infty, 0}$</td>
<td>${-\infty}$</td>
<td>${-\infty, +\infty}$</td>
<td>${+\infty}$</td>
<td>${0, +\infty}$</td>
</tr>
</tbody>
</table>
In the original interval arithmetic formulation, the fact that intervals are sets is not used to increase the domain of interval expressions to include extended real intervals. To permit interval domain extension, the containment constraint is defined without regard to the limitations of either the real or extended real number systems. The containment set of an expression is the smallest set of values that an expression’s interval evaluation must contain. The interval evaluation of any expression must be an enclosure of the expression’s containment set.

Tables 3-1 through 3-4 display containment sets for the four basic arithmetic operations (BAOs). These and other containment sets are derived in [1]. Some explanations are needed:

- First, all arguments are shown as singleton sets. Results are either sets or intervals. To avoid ambiguity, the following customary notation for arithmetic on points is not used:

\[
(-\infty) + (-\infty) = -\infty,
\]
\[
(-\infty) + y = -\infty, \quad \text{and}
\]
\[
(-\infty) + (+\infty) = \mathbb{R}^*.
\]

- Second, tables show results only for singleton-set inputs to each operation. Results for general set inputs are simply the union of the results of the single-point results as they range over the input sets.

The containment set of a relation evaluated at a point can be disconnected compact sets, not necessarily a single interval. To make an implementable system, the easily described family of extended closed intervals, \( \mathbb{I}^* \), is used. As it must be, the whole of \( \mathbb{R}^* \) is in \( \mathbb{I}^* \). Whenever an expression is evaluated, the resulting containment set is replaced by the latter’s interval-hull, resulting in an unsharp, but more easily manipulated enclosure.

A machine implementable interval arithmetic is obtained if \( \mathbb{I}^* \) comprises all closed intervals with IEEE floating-point-representable endpoints (including \( \pm\infty \)).

For division, when the containment set is not an interval, namely \( \{-\infty, +\infty\} \), the interval hull, which is \( \mathbb{R}^* \), is returned. While not the containment set, \( \mathbb{R}^* \) is a sharp interval that does not violate the containment constraint. With an interval system that supports exterior intervals (the union of two semi-infinite intervals), the containment set of division by zero can be represented sharply. In Fortran, only interior intervals are supported. Nevertheless, the simple interval system
implemented in f95 is closed. Consequently, valid enclosures are produced in all cases, including division by zero and indeterminate forms involving zero and infinities.

The closure of expressions can be applied in general, not just to rational functions involving the four arithmetic operations. This result is used to extend the fundamental theorem of interval arithmetic to any expression over closed compact intervals.

**The Fundamental Theorem of Interval Arithmetic**

The fundamental theorem of interval arithmetic provides a simple method of constructing enclosures of rational functions. The theorem was so named by Rall in [6], see [2]. It was originally proved by Moore in [4]. While the theorem is powerful and important, it can be made more general. Even in its more general form, the theorem does not justify that enclosures of composite expressions are composites of sub-expression enclosures. Extensions to the original fundamental theorem are made in Theorems 7 and 9, first proved in [8] and [9], respectively.

First the original fundamental theorem is stated. Then the extensions are added.

**Definition 3, Interval extension.** An expression, $f$, evaluated at the degenerate interval $[x_0]$ is an interval extension of a real function, $f$, evaluated at the point, $x_0$, iff $f([x_0]) = f(x_0)$ for all $x_0 \in D_f$, see [5], page 21.

**Definition 4, Inclusion isotonicity.** An expression, $f$, is inclusion isotonic if for every pair of intervals, $X_0 \subseteq Y_0$, then $f(X_0) \subseteq f(Y_0)$, see inclusion monotonicity in [4].

**Theorem 5, (the original Fundamental Theorem).** Let $f(X)$ be an inclusion isotonic interval extension of the real function $f(x)$. Then $f(X_0)$ contains the range of $f(x_0)$ for all $x_0 \in X_0 \subseteq D_f$, where $D_f$ is the domain of $f$.

Because the four interval arithmetic operators are inclusion isotonic interval extensions, interval arithmetic operations are enclosures of point arithmetic operations. Repeated application of Theorem 5 yields enclosures of rational functions. However, for Theorem 5 to hold, an interval expression must be an interval extension. As a consequence, four important cases are not covered by the original fundamental theorem.
1. If a point expression is not a single-valued real function, Theorem 5 cannot be used to prove that an interval expression is an enclosure of the point expression. For example, Theorem 5 cannot be invoked to construct an enclosure of a point expression at either a singular point or an indeterminate form, such as \( f(x, y) = x / y \), either when \( y = 0 \), or when both \( x = 0 \) and \( y = 0 \).

2. Theorem 5 cannot be used to prove an interval expression is an interval enclosure if the expression is not an interval extension. Suppose an enclosure is the interval evaluation of an approximation of some function to which an interval bound on the approximation error is added. For example, let an enclosure of \( f \) at the point \( x_0 \) be \( f([x_0]) = g([x_0]) + \varepsilon [-1, 1] \) for \( \varepsilon > 0 \). Because \( f([x_0]) \neq f(x_0) \), \( f([x_0]) \) is not an interval extension of \( f(x_0) \). In this case, Theorem 5 does not apply.

3. Theorem 5 does not define how to construct an enclosure when an interval argument is partially or completely outside the domain of the enclosed expression. For example, suppose \( f(x) = \ln(x) \). What is the set of values that must be contained by an enclosure of \( f(x) \) over the interval \( X = [-1, 1] \)? This situation can arise not because of an analysis mistake or coding error, but simply as a consequence of dependence, as illustrated in equation (3).

4. Theorem 5 does not define how to construct an enclosure of a composition from enclosures of component expressions if the component expressions are not inclusion isotonic. For example, given enclosures of subexpressions \( g \) and \( h \), what are sufficient conditions for \( g(h([x_0]), [x_0]) \) to be an enclosure of \( f([x_0]) = g(h([x_0]), [x_0])) \)?

Cases 1 through 3 are covered by simply replacing the interval extension (Definition 3) in Theorem 5 by any interval expression that is a containment-set enclosure.

**Definition 6, Containment-set enclosure.** An expression, \( f \), evaluated at the degenerate interval \([x_0]\), is a containment-set enclosure of a real expression, \( f([x_0]) \), if \( f([x_0]) \supseteq \tilde{f}([x_0]) \).

**Theorem 7.** Let the expression \( f \) have an inclusion isotonic containment-set enclosure, \( f([x_0]) \) of the expression \( f([x_0]) \) at every point, \( x_0 \in X_0 \). Then \( f(X_0) \) contains the range set, \( \overline{R_0} \), of \( \tilde{f}([x_0]) \) for all \( x_0 \in X_0 \). That is,

\[
f(X_0) \supseteq \text{hull} \left( \{ \overline{R_0} \} \right),
\]

where

\[
\{ \overline{R_0} \} = \{ z \mid z \in \tilde{f}([x_0]), x_0 \in X_0 \}.
\]
The proof parallels the proof of the original fundamental theorem.

**Proof.** Assume \( x_0 \in X_0 \). By inclusion isotonicity, \( f(X_0) \) contains \( f([x_0]) \), which in turn contains \( f([x_0]) \), because \( f([x_0]) \) is a containment-set enclosure of \( f([x_0]) \). Since this is true for all \( x_0 \in X_0 \), \( f(X_0) \) contains the range set of \( f \) over \( X_0 \). ■

Theorem 7 guarantees that extended real interval arithmetic operations contain the range of values produced by the corresponding point operation over all elements of the extended interval operands. This is true even for combinations of operations and operands for which the corresponding point operation is undefined. The evaluation of any rational expression is a finite sequence of arithmetic operations. Therefore, the range of any rational expression must be contained in the corresponding enclosure defined by the same set of extended interval operations.

While the consequences of the simple change from Theorem 5 to 7 are sweeping, neither theorem covers case 4 listed above. Even with Theorem 7, it remains unclear how to construct enclosures of composite expressions from sub-expression enclosures that are not inclusion isotonic. In [9], Walster and Hansen cover this case by extending the fundamental theorem to include compositions. The first step is to extend the definition of a containment-set enclosure to expressions evaluated over non-degenerate intervals:

**Definition 8, Containment-set enclosure:** An expression, \( f \), evaluated over an interval, \( X_0 \), is a **containment-set enclosure** of the expression, \( f \), if \( f(X_0) \supseteq \tilde{f}([X_0]) \).

**Theorem 9.** Given real expressions, \( f(x) \) and \( g(y, x) \), the composite expression \( f(x) = g(h(x), x) \), and containment-set enclosures, \( Y_0 = h(X_0) \) and \( g(Y_0, X_0) \), then \( g(Y_0, X_0) \) is a containment-set enclosure of \( f(X_0) \).

**Proof.** From the definition of the containment set of \( f \) and the containment-set closure identity

\[
cset(f, X_0) = \tilde{f}(X_0) \supseteq g(h([X_0]), \{X_0\}). \tag{47}
\]

Because \( g \) and \( h \) are containment-set enclosures, \( h([X_0]) \subseteq h(X_0) = Y_0 \) and

\[
\tilde{g}(h([X_0]), \{X_0\}) \subseteq g(h([X_0]), \{X_0\}), \tag{49}
\]

the required result. ■
Remark 1. The containment-set enclosures on which Theorem 9 depends can be created from the definition of closures or by prior application of Theorem 9. Inclusion isotonicity is not required.

Expression Evaluation

Repeated application of the fact that:

\[ \tilde{f}(\{X_0\}) \subseteq \tilde{g}(\{(y_0, x_0) \mid y_0 \in \tilde{h}(\{X_0\}, x_0 \in X_0)\}) \]  

is a way, in principle, to compute a bound on the containment set of an arbitrary expression. See [1] for the proof of this result. However, repeated application of the right-hand side of (51) will, in general, be impractical because it is the union of a large number of disjoint sets. Nevertheless, this result can be symbolically represented as:

\[ \text{eval}(\tilde{f}, \{X_0\}) \supseteq \text{cset}(f, \{X_0\}) . \]  

Alternatively, intervals provide a practical way to compute bounds on the containment set of an expression using repeated application of Theorem 9. This guarantees that the containment set of any expression defined by a finite sequence of compositions is contained in the same composition of containment-set enclosures:

\[ \text{cset}(f, X_0) \subseteq \text{eval}(f, X_0) , \]

where \( \text{eval}(f, X_0) \) represents the interval evaluation of an arbitrary expression.

Containment-Set-Equivalent Expressions

Because domains of basic arithmetic operations are open sets, interval arithmetic has inherited all the restrictions on arithmetic using real variables. Two real functions, \( f \) and \( g \), are mathematically identical iff \( f(x) = g(x) \) for all \( x \in D_f = D_g \). Only in this circumstance can \( f \) and \( g \) be exchanged, because in this case they are everywhere identical. The general question of when expression substitution is permitted was posed at the end of Section 2.

In interval analysis, two expressions can be exchanged if they are containment-set equivalent. Two expressions are containment-set equivalent if they have identical containment sets for all possible values of their arguments. The interval evaluation of containment-set-equivalent expressions produces enclosures of their common containment set. Therefore, expression exchange of containment-set-equivalent expressions cannot cause a containment failure. This result can be
used to choose the best containment-set-equivalent expression for a particular situation, depending, for example, on whether speed or interval width is more important.

Without loss of containment, expression $g$ can replace expression $f$ in any expression, if for all $\{x\} \in (\mathbb{R}^n)$:

$$\text{cset}(f, \{x_0\}) \subseteq \text{cset}(g, \{x_0\}).$$

(54)

**Definition 11.** Two expressions, $f$ and $g$, are containment–set equivalent iff

$$\text{cset}(f, \{x_0\}) = \text{cset}(g, \{x_0\}) \text{ for all } x_0 \in (\mathbb{R}^n)^n.$$

For example, expressions (8), (9), and (11) on pages 8 and 9 are containment-set equivalent, because their containment sets are the same everywhere in $(\mathbb{R}^n)^n$. As a consequence, their enclosures can be exchanged without loss of containment.

For the non-degenerate interval argument, $X_0$,

$$\text{cset}(f, \{X_0\}) = \left\{ z \mid \begin{array}{l} z = \text{cset}(f, \{x_0\}) \\ x_0 \in X_0 \end{array} \right\}$$

(55)

$$= \left\{ z \mid \begin{array}{l} z = \bar{f}(\{x_0\}) \\ x_0 \in X_0 \end{array} \right\}$$

(56)

$$\subseteq \text{eval}(\bar{f}, \{X_0\})$$

(57)

$$\subseteq \text{eval}(f, X_0)$$

(58)

Although different expressions can be containment-set equivalent, their enclosures may have different widths. Because different containment-set equivalent expressions are interchangeable, whichever one returns the narrowest result can be used. Containment-set equivalence expands the set of expressions for which substitutions are possible.

**Interpreting Empty Interval Results**

To close interval systems, provision is made in $\mathbf{f95}$ input/output to represent the empty interval using the character string [empty]. The empty interval is the same as the empty set, $\emptyset$, in mathematics. Interpreting empty interval results requires some care, because depending on the context, an empty result may be an error indicator.
Intervals have a dual numeric and set interpretation. In a set context, empty results and operations on empty intervals are natural. For example, an empty interval in a set context is a valid outcome of the intersection (.IX. in £95) of two disjoint intervals. The interval hull (.IH. in £95) is the smallest interval that contains the union of two intervals. Therefore, the interval hull of an empty interval and any interval, say $X$, is simply $X$.

Intrinsic functions with a finite domain can also produce empty intervals. When they do, it may or may not indicate a programming or algorithm error has occurred. For example, it is an error if a computed interval that must be non-negative turns out to be strictly negative. This is an error because it is a containment failure. In £95, this error can be detected in a number of ways. For example, an explicit test can be made using the certainly-less-than relational operator (.CLT. in £95):

$$\text{IF}(X \cdot \text{CLT} \cdot 0)$$

See the Relational Operators section on page 34 for additional interval-specific relational operators.

Alternatively, if an intrinsic function with a non-negative domain is given a strictly negative interval argument, an empty interval result may be an indication of an error. For example, if $X$ is the complete result of a calculation that must be non-negative, $X$ may contain some negative values, because of rounding and/or dependence. See the example in equation (3) on page 6. However, $X$ must contain some non-negative values. Therefore, if $X$ is strictly negative, $\sqrt{X}$ returning the empty interval is an indication of an error. However, if $X$ is only partially negative and $X$ is partitioned into two sub-intervals, $X_1$ and $X_2$, with $X_1$ strictly negative, $\sqrt{X_1}$ returning the empty interval is not an error. If $X_1$ is strictly negative, the empty interval is the required result for the following expression to return a sharp interval result for all values of $X_2$: $\sqrt{X_1}.\text{IH}.\sqrt{X_2}$.

Context is necessary to definitively determine whether an empty result is an error. Whenever an arithmetic result is unexpectedly empty, an error may have occurred.

Most empty interval results propagate through arithmetic operations, and therefore will not be lost. However, there are four ways an erroneous empty intermediate result can be hidden:

- An empty result is an operand of a hull operation with a non-empty operand. This can be prevented by checking for interval hull operands that may be unexpectedly empty.
• An empty result is an operand of an intersection operation, the result of which legitimately can be empty. To prevent losing an erroneous empty result, check to make sure arguments of the intersection operator are not empty if they cannot be empty. The Continuity Examples section on page 28 illustrate this situation.

• Dead code. (There is no path to dead code.) To prevent losing an empty result, check for, and eliminate, dead code.

• An empty result is part of an incomplete set of relational tests. Complete sequences of relational tests can be used to prevent the loss of an empty interval. For example, the code in Figure 3-1 can hide the fact that \( B \) is empty. If \( A \) is empty, \( X \) must be also. However, if \( B \) is empty, \( X \) is set to \( A^{**}2 \) in Figure 3-1. If \( B \) is erroneously empty and never used in subsequent code, the error that caused \( B \) to be empty may go undetected.

```
IF (A .CLT. B) THEN
  X = A
ELSE
  X = A**2
ENDIF
```

*Figure 3-1* Incomplete sequence of relational tests

Affirmative relational tests (any order relation other than certainly- or possibly-not-equal) are false if either operand is empty. To prevent empty results from hiding in incomplete sequences of relational tests, either check if \( A \) or \( B \) is empty before executing the code in Figure 3-1, or use a complete set of affirmative relational tests, as shown in Figure 3-2.

```
IF (A .CLT. B) THEN
  X = A
ELSEIF (A .PGE. B) THEN
  X = A**2
ELSE
  ! Code to deal with A or B being empty
ENDIF
```

*Figure 3-2* Affirmative relational tests
Coding complete sequences of relational tests is facilitated with the full set of certainly-, possibly-, and set-relational operators. (See the Relational Operators section on page 34.)

Compiler support can be provided to alert programmers when each of the four empty-hiding mechanisms exists. For example, a compiler can:

- flag any interval hull operator containing an operand that results from an arithmetic operation,
- flag any incomplete relational test sequence,
- flag any intersection operator containing an operand that results from an arithmetic operation, and
- identify dead code.

These mechanisms are currently not implemented in f95, and therefore remain quality of implementation opportunities.

Continuity

Some interval algorithms work by proving that solutions cannot exist. Interval algorithms progress by deleting places that are proved to not contain a solution. Perhaps the interval algorithm for finding the global minimum of a nonlinear function is the simplest example of an interval algorithm that deletes where solutions do not exist. A version of this algorithm can be based on nothing more than the fundamental theorem of interval arithmetic. If the objective function over some interval or box is strictly greater than the current least upper bound on the objective function, then the box in question can be deleted, because it cannot contain the global minimum. No assumptions are necessary.

To find stationary points and roots of nonlinear equations, two different theorems are commonly used as the basis for interval algorithms to prove existence and/or non-existence of a solution. They are Brouwer’s fixed-point theorem and the mean-value theorem. The mean-value theorem is the basis for the interval version of Newton’s algorithm to find the roots of nonlinear equations. Both theorems depend on the assumption of function continuity over the interval in question. The Brouwer fixed-point theorem and the interval Newton algorithm are examples of an interval contraction mapping. Successive iterations of a contraction mapping can result in successively narrower intervals, ultimately limited by wordlength.
Because traditional interval arithmetic inherits the limitations of point arithmetic on the open domains of real functions, little consideration has ever been given to questions of function continuity. Continuity has just been assumed, because whenever an argument is partially outside an operator or function’s domain, an exception is raised. However, if branching or interval versions of discontinuous intrinsic functions, such as the Fortran\texttt{ANINT} function, are used, continuity may not exist and no exception will be raised.

In the closed interval system, there are no exceptional events and no provision for verifying continuity. Closed interval systems have only one requirement, containment. Proving continuity is not a requirement for an interval implementation. Nevertheless, it is a quality of implementation opportunity to automate function continuity verification to guarantee correct application of interval algorithms that depend on the continuity assumption.

A simple way to prove continuity of an expression is to verify that every operation and function used to compute an expression is continuous over argument intervals and that no branching has occurred. This is a sufficient, but not necessary condition for continuity. Implementing this mechanism is similar to the exception testing process used in real point arithmetic, but also requires that branching be precluded. A second important difference is that continuity verification is only required in special circumstances, such as application of Brouwer’s fixed-point theorem or the interval Newton algorithm. Normally, continuity verification is an unnecessary barrier to performance.

The \texttt{f95} compiler does not contain a continuity verification test. The syntax and semantics are under design. Within a specified block of code, three different mechanisms to signal continuity are being considered:

- Set a logical variable to be \texttt{true} or \texttt{false} depending on whether any operation has been performed that is inconsistent with the continuity hypothesis.
- Transfer control to a specified label as soon as the continuity hypothesis cannot be maintained.
- Terminate execution to identify the first location where the continuity hypothesis cannot be sustained.

Depending on the structure of user’s code, one or the other mechanism will be preferred. No appreciable run-time performance penalty will result if this mechanism is made part of more complete interval instruction hardware. In the meantime, the same syntax and most of the semantics can be provided with a module, but at the cost of runtime performance.
Continuity Examples

The following two examples illustrate how failure of the continuity assumption in Brouwer’s fixed-point theorem and the interval Newton algorithm can lead to erroneous conclusions.

**Theorem 14.** Let $f$ be a continuous function and $X$ be a closed compact interval. If $f(X) \subseteq X$, then there is at least one fixed-point, $x^* \in X$, for which $f(x^*) = x^*$.

The converse result can also be used. That is, if $f(X) \cap X = \emptyset$, then there can be no fixed-point of $f$ in the interval, $X$. These two results form the basis for a simple interval algorithm to find fixed-points. However, the assumption of continuity must not be overlooked, or incorrect conclusions can result. For example, consider the simple function:

$$f(x) = \sqrt{x} - 1. \tag{59}$$

Performing the interval evaluation of $f$ over the interval $[-4, 4]$, results in the returned value of $[0, 1]$, which, according to Brouwer’s fixed-point theorem, implies that there is a fixed-point of $f$ in the interval $[0, 1]$. There is not, as can be readily seen in Figure 3-3. Subsequent iterations in this case confirm that there is no fixed point, but damage to an algorithm may have been done.

![Figure 3-3](image)

**Figure 3-3** $f(x) = \sqrt{x} - 1$. 

Interval Arithmetic in Forte Fortran
For an interval Newton example, consider the function:

\[ f(x) = \frac{1}{2} + x^2 + \sqrt{x} \quad (60) \]

over the starting interval, \( X_0 = [-1, 3] \). Each iteration in the interval Newton algorithm takes three steps:

\[ x_n = m(X_n) \quad (61) \]

\[ N(x_n, X_n) = x_n - \frac{f([x_n])}{f'(X_n)} \quad (62) \]

\[ X_{n+1} = X_n \cap N(x_n, X_n) \quad (63) \]

where \( m(X) = (x + \hat{x})/2 \), the midpoint of the interval, \( X; f([x_n]) \) is the interval evaluation of \( f \) at the point \( x_n \); and \( f'(X) \) is the first derivative of \( f \), evaluated over the interval, \( X \). That is, \( f'(X) \) is a bound on the range of the derivative of \( f \) over the interval, \( X \). In the present case, \( m([-1, 3]) = 1 \) and \( f([1]) = [2.5] \). The range of the first derivative of \( f \) over the interval \([-1, 3]\) is \([1.5, +\infty)\). So, the value of \( N(1, [-1, 3]) \) is:

\[ 1 - \frac{[2.5]}{[1.5, +\infty]} = \left[ \frac{2}{3}, 1 \right] \quad (64) \]

\( \subset [-1, 3] \quad (65) \)

Therefore, the false conclusion is drawn that a root of \( f \) exists in the interval \( \left[ \frac{2}{3}, 1 \right] \), as can be seen by examining Figure 3-4.

This figure depicts the interval bound on \( f \) that is used to construct the interval Newton iteration. The bound on \( f \) is computed from the interval bound on the first term of the Taylor series expansion of \( f \) about the point, \( x = 1 \). The interval Newton step is the intersection of the Taylor series bound with the equation \( y = 0 \). In the next iteration, \( m(\left[ \frac{2}{3}, 1 \right]) = \frac{1}{6} \) and the Newton step intersected with the interval, \( \left[ \frac{2}{3}, 1 \right] \), is \( \left[ \frac{1}{3}, \frac{1}{6} \right] \).

The midpoint of this interval is negative, resulting in an empty value for \( f(\left[ m(X) \right]) \), terminating the sequence with the determination that no root exists in the interval \([-1, 3]\). While, as in the Brouwer example, the correct result is ultimately achieved, the initial result that a root is proved to exist is incorrect. A good test to add to any interval Newton iteration is that if \( N(x_n, X_n) = \emptyset \) then the
continuity assumption has been violated or some other error has occurred. This is an example of an intersection operator hiding an erroneous empty interval. (See the Interpreting Empty Interval Results section on page 23.)

Figure 3-4  \( f(x) = \frac{1}{2} + x^2 + \sqrt{x} \)

For further consideration of an extended interval Newton algorithm, see [7].

**Summary**

With the development described above, closed interval systems can be constructed to evaluate any finite-length, code-list-representable expression. In these systems, there are no restrictions on an expression’s domain of definition. These systems have all the properties of the classical interval system applied to rational functions of real intervals. In particular, the fundamental theorem of interval arithmetic carries over. Definitions of a containment failure, the containment set of an expression, and containment-set-equivalent expressions permit more interval expression substitutions than are possible with point expressions. In an interval context, containment set equivalence is the operational definition of *mathematical equivalence* in the Fortran Standard. Therefore, it is within the spirit of the Fortran standard to permit a compiler to substitute containment set equivalent expressions for the purpose of decreasing the interval result width and/or processing time.
Ease-of-Use

Having established the foundation for closed interval systems, the remainder of this white paper deals with four ease-of-use features in Forte Developer Fortran (f95):

• context-dependent interval constants,
• widest-need expression processing,
• interval-specific relational operators, and
• single-number input/output.

Context-Dependent Interval Constants

Many characteristics and properties of intervals violate common intuition. Even the notion that a constant is invariant need not be true for intervals — this is not an oxymoron. Given that containment is the one overriding requirement of any interval system implementation, the contained constant is the invariant, not necessarily the containing interval. When operationalizing interval constants, it is the value they must contain that is invariant, not their internal machine approximation.

In Fortran, literal constants (hereafter, simply constants), are the internal approximation of a mathematical constant. The problem is that accuracy of internal approximations is never made explicit in the Fortran standard. The standard specifies that constants are approximations of something — without specifying what the something is. Because containment is a requirement of an interval implementation, it is necessary to specify what an interval constant must contain. The value that an interval constant must contain is the constant’s external mathematical value, or simply the constant’s external value as contrasted
with the constant’s *internal machine approximation*, or simply the constant’s *internal value*. For the `REAL` constant, 0.1, there is no requirement in the Fortran standard for the Fortran constant 0.1 to be close to the number 1/10. In contrast, the interval constant [0.1] must contain the mathematical value, 1/10.

To have an unambiguous way to denote the external value of a `REAL` or `INTERVAL` constant, the notation `ev(constant)` is used to represent the external value of a Fortran constant. The letters `ev` are a mnemonic for *external value*. For example

\[
ev(0.1) = 0.1, \text{ and } \\
ev([0.1, 0.1]) = [0.1, 0.1].
\]

It is the external value of an interval constant that is invariant. The choice of the internal approximating interval that contains its external value can be allowed to depend on the context provided by the expression within which the interval constant exists. For example, depending on the precision of other interval constants and variables in an expression, a single-, double-, or quad-precision interval constant may be more appropriate. The contained value does not change, but the containing interval constant can be allowed to depend on context.

Context-dependent interval constants are used in mixed-mode expressions. For more details regarding interval constants, see Section 2.1.1 in the *f95 User Documentation*, and [11].

**Mixed-Mode Interval Expression Processing**

Language and compiler support for interval data types must do more than simply produce results that satisfy the containment constraint. Whether and how to evaluate interval expressions containing different precision intervals and non-interval data items is a significant quality of implementation opportunity that can dramatically increase ease-of-use.

**Strict Expression Processing**

One alternative is to constrain any single interval expression to contain only interval variables and constants with the same precision. In this case, non-interval data items cannot be mixed with intervals, and all interval data items must have the same precision. This style of expression processing is supported in *strict* expression processing mode. Strict mode is invoked with *f95* using the command line in Figure 4-1.
In strict mode, users must explicitly convert all non-interval data items in any given expression to the INTERVAL type, and all interval data items to the same precision. With guaranteed containment, the INTERVAL type conversion intrinsic function is provided to perform the conversions between different precision intervals. To guarantee containment of a constant’s external value, it is safer to use interval constants or widest-need expression processing. (See the following section on widest-need expression processing, [5], and Section 2.8.2 on page 81 in the Interval Arithmetic Programming Reference for more details).

Care must be taken when creating an interval data item from REAL constants with the INTERVAL type conversion intrinsic function. Under strict expression processing, explicit precision and type conversions are error prone because they are tedious to write and result in code that is difficult to read. These are not characteristics of an easy-to-use interval implementation. Therefore, in f95, a second widest-need mode of interval expression processing is also implemented.

**Widest-Need Expression Processing**

Widest-need expression processing is invoked using the command line illustrated in Figure 4-2.

In widest-need mode, the following steps, or their logical equivalent, are taken prior to expression processing:

1. All non-interval data items are converted to intervals having precision sufficient to permit non-interval data items to be exactly represented. Therefore, integers with a given precision (kind type parameter value, or KTPV, in Fortran parlance) are converted to intervals with the next higher KTPV.
2. All interval and converted-interval data items are promoted to the highest precision found anywhere in the expression, including the left-hand side of an assignment statement.

Interval constants with context-dependent precision are needed to implement widest-need expression processing. Automating non-interval constant precision conversion was actually the original motivation for widest-need expression processing. (See [12] and [13].)

Relational Operators

Intervals are more interesting to numerically order than are points. Given two intervals, \([a, b]\) and \([c, d]\), if \(a < d\), at least the element \(a\) in the interval \([a, b]\) is less than the element \(d\) in \([c, d]\). If \(a < c\), \(a\) in \([a, b]\) is less than all elements of \([c, d]\). If every sequence of relational tests must be hand crafted using interval endpoints, logical mistakes are likely and code is difficult to read. As was seen in Figure 3-1, incomplete relational test sequences can hide erroneous empty intervals. Carefully constructed, complete relational test sequences can be used to expose unanticipated empty intervals. If meaningful relational operators are available and used, correct code is more likely to be written and understood when read. One way to untangle interval order relations is to introduce two classes of operators: certainly-true and possibly-true relations.

Certainly-True Relations

In interval arithmetic, a certainly-true, or simply certainly, relation is true only if the relation in question is true for every element of the operand intervals. If either operand of a certainly relation is empty, the result is false. (The certainly not equal relation is the one exception, and is true when either operand is empty.)

Fortran supports the certainly form of all the normal relational operators:

- Certainly equal (.CEQ.)
- Certainly greater or equal (.CGE.)
- Certainly greater (.CGT.)
- Certainly less or equal (.CLE.)
- Certainly less than (.CLT.)
- Certainly not equal (.CNE.)
Possibly-True Relations

A possibly-true, or simply a possibly, relation is true if the relation in question is true for any element of the operand intervals. If either operand of a possibly relation is empty then the result is false. (The possibly not equal relation is the one exception, and is true in this case.)

£95 supports the possibly form of all the normal relational operators:
- possibly equal (.PEQ.),
- possibly greater or equal (.PGE.),
- possibly greater (.PGT.),
- possibly less or equal (.PLE.),
- possibly less than (.PLT.), and
- possibly not equal (.PNE.).

Set Relations

Intervals have a dual interpretation, as either sets of real numbers or sets without regard to the fact that interval elements are real numbers. Order relations rely on the fact that intervals are sets of real numbers. Some set relations do not rely on the fact that interval elements are real numbers. For example, the set operation intersection (.IX. in £95) has its usual set-theoretic meaning. In interval arithmetic, set relations either treat intervals as sets, or apply only to the interval operand’s endpoints.

The most commonly used set operations are set-equal and set-not-equal. These two operations are the only order operations for which overloaded default .EQ. (or ==) and .NE. (or /=) operators are recognized. In all other cases, the specific type of operation: certainly, possibly, or set, must be made explicit.

For completeness, the remaining set order-relations are included. The relations are true if they hold for the respective endpoints. For example, if $a < c$ and $b < d$, then $[a, b]$ is set-less-than $[c, d]$.

£95 supports the set form of all the normal relational operators:
- set equal ($X=Y$ or $X.SEQ.Y$),
- set greater ($X.SGT.Y$),
- set greater or equal ($X.SGE.Y$),
- set less or equal ($X.SLE.Y$),
- set not equal ($X/=Y$ or $X.SNE.Y$), and
- set less than ($X.SLT.Y$).
Relationship Between Certainly and Possibly Relations

A useful relationship exists between certainly and possibly relations. Let \( .C_{\text{op}} \) and \( .P_{\text{op}} \) stand for the certainly and possibly forms of the operation \( \text{op} \). Also, let \( \text{nop} \) stand for the complement of \( \text{op} \). For example, if \( \text{op} \) is less-than, \( \text{nop} \) is greater-than-or-equal. The relationships are illustrated using the INTERVAL variables \( A \) and \( B \). Provided neither \( A \) nor \( B \) is [empty]:

- \( A.C_{\text{op}}.B = .NOT.(A.P_{\text{nop}}.B) \)
- \( A.P_{\text{op}}.B = .NOT.(A.C_{\text{nop}}.B) \)

For example, \( A.CLT.B = .NOT.(A.PGE.B) \)

If the intervals \( A \) and \( B \) are both degenerate intervals, the certainly, possibly, and set relational operators converge to the normal order relations between points. See Section 2.7.8 in the Interval Arithmetic Programming Reference for more information.

Empty Operands

Affirmative relational operators are all those except not-equal, (.NE. in Fortran). Not equal is anti-affirmative. For all certainly and possibly affirmative operators, empty operands produce a false result. Just the opposite is the case for the anti-affirmative operator, not-equal, whether in its certainly or possibly form. Because two empty intervals are set-equal to each other, empty interval operands do not interact with set relational operators the same way they do with certainly and possibly operators. For more details, see Section 2.7.8 in the Interval Arithmetic Programming Reference.

Complete Relational Test Sequences

With both the certainly and possibly classes of relational operators, it is simple to make sure that relational test sequences include all possibilities so that empty operands are identified. Not losing an empty operand can help to detect errors. It is the duality between certainly and possibly relations that is used in Figure 3-2 on page 25 to construct a complete relational test sequence.
Single-Number Interval Input/Output

Visually scanning interval output and interactively entering interval input is often frustrating. Indeed, the requirement for users to compare infima and suprema to count the number of digits that agree is not only time consuming, it can easily lead to interpretation errors. Scanning a large number of interval values to identify which are more or less accurate is almost impossible. For example, consider the eye strain caused by scanning the following list of [inf, sup]-style values in Figure 4-3.

```
[ 0.14385979, 0.14386012]
[ 3.87564567, 3.87566567]
[12.43123458, 12.43123526]
```

Figure 4-3  Conventional interval output hides the relative width of interval results

In conventional interval input, users must enter explicit lower and upper bounds. In some situations, this is inconvenient. For example:

- when the input interval value is degenerate, whether or not it is machine representable, or
- when interval endpoints are different only in the last of a long string of digits.

Single-number interval input/output overcomes some of the limitations of conventional interval input/output. By allowing each interval to be read or displayed using a single decimal number, single-number interval input/output provides a display format in which any number not enclosed in square brackets represents an interval.

The single-number interval display format represents an interval by adding the interval, \([-1, +1]_{uld}\), to the last displayed digit. The subscript, \(uld\), is a mnemonic for “unit in the last digit.” This technique enables each interval to be externally represented as a single decimal number. Single number input/output provides a convenient way to enter and display interval data. Figure 4-4 contains two simple examples.
Figure 4-4  Single-number interval representation

Note that with single-number input immediately followed by single-number output, a decimal digit of accuracy can appear to be lost, as shown in Figure 4.5:

<table>
<thead>
<tr>
<th>Single-number input</th>
<th>Single-number output</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>0.1000</td>
<td>0.100</td>
</tr>
</tbody>
</table>

Figure 4-5  Echoing exact input character strings

In the second case one digit appears to be lost because 0.1 is not machine representable, causing outward rounding to be performed on input and/or output. Outward rounding on input and output is required to satisfy the interval containment constraint. The stored value has width that is increased by 1-ulp, or unit in the last place of the internal representation. Therefore, if the input value is not machine representable, the single number used to represent the internal approximation on output must contain one less decimal digit.

To exactly echo input character strings, character input and internal conversion can be performed. See code example 1-6 on page 22 of the Interval Arithmetic Programming Reference.

**Interval Input**

Let $x_d$ represent a decimal number that can be input to a formatted READ statement using the D$w$.d edit descriptor. Interval input can be performed using any of three formats:

- $[x_d]$, represents the degenerate interval $[x_d, x_d]$,
- $x_d$, represents the interval $x_d + [-1, +1]_u l d$, and
- $[x_{d1}, x_{d2}]$, is the traditional [inf, sup] form.
It all cases, input conversion constructs an internal approximation that contains the input interval. For example, when the interval \([0.1, 0.2]\) is read, the constructed internal approximation is the narrowest machine representable interval containing the input interval. The interval containment constraint mandates this behavior. Code examples 2-38 on page 102 through 2-53 on page 114 in the *Interval Arithmetic Programming Reference* illustrate many of the interval-specific details associated with formatted interval input/output. All code examples in the *Interval Arithmetic Programming Reference* are available on the Web at http://www.sun.com/forte/fortran/interval.

Empty intervals can be input using the [empty] character string. Infinite interval endpoints can be input using -inf or +inf, as for example in \([-\infty, +2]\).

**Interval Output**

All the standard edit descriptors in Fortran can be used to format intervals for output. In addition, there are a number of interval-specific edit descriptors. The most important new interval-specific edit descriptor is \(\text{Y} w.d\), which is used for single number output. Users have all the normal control over the formatting of Fortran output. The letter \(\text{Y}\) is a mnemonic for:

\[
\begin{align*}
\text{[inf, sup]} \\
\text{\text{Y}} \\
\text{single-number}
\end{align*}
\]
The utility of intervals and interval algorithms are well known and appreciated in the interval research community. For the utility of intervals to become widely available in commercial applications, interval algorithms must be built into commercial software. Requisites for commercial interval applications to be developed include: intrinsic compiler support for interval data types, a quality interval software development environment, and libraries of linear and nonlinear interval solvers. Commercial quality compiler support for interval data types requires more than satisfying the interval containment constraint. Other contributing factors to quality of interval support include:

- narrow width interval results,
- speed, and
- ease-of-use.

Within the constraints imposed by existing hardware, the code generated by the f95 compiler quickly computes narrow interval results. In addition, several ease-of-use features are provided, including:

- a closed interval system, with support for the empty and extended (infinite) intervals, thereby eliminating runtime exceptions and unnecessary defensive code;
- strict expression processing;
- widest-need expression processing to support mixed-type and mixed-precision expressions;
- interval-specific intrinsic operators and functions, with certainly-, possibly-, and set-relational-operators;
a complete interval implementation of the Fortran 95 standard, with interval versions of every intrinsic function and operator that returns a REAL value or has REAL arguments; and

single-number input/output to facilitate entering and reading interval data.

For intervals to become core computing technology in the general engineering, scientific, and commercial communities, computer manufacturer support for interval arithmetic is required. With its latest release of the Forte Developer Fortran compiler, Sun is setting the standard for interval support with guaranteed interval containment, high performance, reliability, narrow width, and ease-of-use features. As with any core technology, Sun continues to invest in interval arithmetic and related technologies to ensure that Sun customers always have access to the best available products. Sun also pursues relationships with other industry leaders to find new ways of delivering products and services.
Sun Microsystems posts product information in the form of data sheets, specifications, and white papers on the Internet at http://www.sun.com.


Other white papers of interest:


Web sites of interest:

http://www.sun.com

http://docs.sun.com

http://www.sun.com/forte/fortran/interval
Sales Offices

Africa (North, West and Central): +33 1 30674680
Argentina: +54-11-4317-5600
Australia: +61-2-9844-5000
Austria: +43-1-60563-0
Belgium: +32-2-704-80-00
Brazil: +55-11-5187-2100
Canada: +905-477-6745
Chile: +56-2-3724510
Colombia: +571-629-2323
Commonwealth of Independent States: +7-502-935-8411
Czech Republic: +420-2-3200-9331
Denmark: +45 4556 5000
Egypt: +202-570-9442
Estonia: +372-6-308-900
Finland: +358-9-925-561
France: +33-01-30-67-50-00
Germany: +49-89-46008-0
Greece: +30-1-618-8111
Hungary: +36-1-202-4415
Iceland: +354-563-3010
India: +91-80-5599595
Ireland: +353-1-8055-666
Israel: +972-3-933465
Italy: +39-039-60551
Japan: +81-3-5717-5000
Kazakhstan: +7-3272-466774
Korea: +82-2-3699-1114
Latvia: +371-750-3700
Lithuania: +370-729-8468
Luxembourg: +352-2011-331
Malaysia: +60-264-9988
Mexico: +52-5-258-6100
The Netherlands: +31-33-450-1234
New Zealand: +64-4-499-2344
Norway: +47-2203-3900
People's Republic of China:
  - Beijing: +86-10-6803-5588
  - Chengdu: +86-28-619-9333
  - Guangzhou: +86-20-8755-5900
  - Hong Kong: +852-2802-4188
  - Shanghai: +86-21-6466-1228
  - Tokyo: +81-3-5717-5000
  - Taipei: +886-2-2369-9900
  - Tianjin: +86-22-699-9444
  - Switzerland: +41-1-908-9000
Taiwan: +886-2-2369-9900
Thailand: +66-236-3555
Turkey: +90-212-236-3300
United Arab Emirates: +971-4-3366333
United Kingdom: +44 (0)1252 420000
United States: +1-800-555-9777 OR +1-650-960-1300
Venezuela: +58-2-905-3800
Worldwide Headquarters:
Internet: www.sun.com