INTERVAL ARITHMETIC WITH CONTAINMENT SETS*

JOHN D. PRYCE † AND GEORGE F. CORLISS ‡

Abstract. The idea of containment sets (csets) is due to Walster and Hansen, and the theory is mainly due to the first author. Now that floating point computation with infinities is widely accepted, it is necessary to achieve the same for interval computation. The cset of a function over a set in its domain space is the set of all limits of normal function values over that set. It forms a sound basis for defining a number of practical models for interval arithmetic that handle division by zero and related operations in an intuitive and consistent manner. Cset-based systems offer new opportunities for compiler optimization by rearranging interval expressions, achieving tighter bounds by reducing dependencies within the expression.

This paper presents basic theory. It discusses division by zero, the need for a global flag to support "loose" evaluation, performance, and semantics. It presents numerical examples using a trial Matlab implementation.

Key words. Interval arithmetic, validated computation, division by zero, infinity, containment set, cset.

AMS subject classifications. 65-02, 65G30, 65G40, 54D35

1. Context.

Validated calculation finds guaranteed bounds on computed numerical quantities. On a computer this is usually, but not necessarily, done by *interval arithmetic* (IA) with directed rounding. Usually it carries a time penalty compared with normal floating-point arithmetic, but this is reducing as algorithms and low-level support mature. For some applications such as global optimization, interval-based methods compete with and can outperform methods that use pure floating-point.

As a result intervals are moving into the mainstream of scientific computing. What is needed above all to make this happen is an easily usable collection of high quality interval software, portable across all major platforms. The authors, together with Baker Kearfott, Ned Nedialkov and Spencer Smith, and with encouragement from NAG Ltd and Sun Microsystems, have joined to initiate the *Interval Subroutine Library* (ISL) project to produce such a collection.

This is a major, many-layered effort. We see it taking a number of years to reach even the standard of an early release of the NAG library. However, it is more about integrating and standardizing the large body of existing high-quality software than about new mathematics. If we are even partly successful, access to a reliable interval infrastructure in production code will be simple and essentially universal, not time consuming and patchy as it is today.

An important ingredient is to take part in work on Interval Arithmetic Standards for commonly used languages. We are presently collaborating in such work for C++. The first author's role has been as "theory police", to ensure the mathematical foundations are well laid. One may think the foundations of a subject as mature as IA are secure, but regarding correct and useful handling of infinity — essential now that the IEEE standard does it so well for point arithmetic — this is far from the case.

*This work is partially supported by grant EP/D033373 from the UK Engineering and Physical Sciences Research Council and by a research contract with Sun Microsystems (now concluded).

†Dr John D. Pryce, Department of Information Systems, Shrivenham Campus, Cranfield University, Swindon SN6 8LA, UK. j.d.pryce@cranfield.ac.uk

‡Prof George F. Corliss, Department of Electrical and Computer Engineering, Marquette University, 1515 Wisconsin Avenue, P.O. Box 1881, Milwaukee, WI 53201-1881, USA. george.corliss@marquette.edu

This article argues that the mathematical foundation for infinity in IA should be *csets* (containment sets). The original idea for these is due to Walster and Hansen. The theory was developed mainly by Pryce in collaboration with them.

The theory of csets involves a choreography of topology, algebra and semantics that handles infinity in a systematic and (once one sees the principles) obviously consistent way. It has features, such as supporting *rearrangement* of functions into algebraically equivalent form — for tighter enclosures or other purposes — possessed by no other current interval theory.

2. Principles of validated calculation. As a general principle we believe:

PRINCIPLE 2.1. *Both traditional and cset interval arithmetic should be founded on set theory and real analysis.*

Why? Because intervals are becoming mainstream. We wish them to be in the mental toolkit of every engineer and numerical analyst — whose computational methods are based on real (and complex) analysis. There are other approaches. One is to treat intervals as an abstract algebraic system with its own axioms. Another uses a number-field containing infinitesimal values, as in "nonstandard" analysis; or a version of the reals containing distinct –0 and +0. We feel these can put people off intervals. We do not aim such criticism at a field such as "exact computational geometry", which applies analysis to arbitrary-precision computing and to which csets can contribute.

A consequence of Principle 2.1 is

PRINCIPLE 2.2. *An interval* $[a, b] = \{ x \mid a \le x \le b \}$ *is a particular kind of subset of the number system.*

— a subset that one can represent concisely and manipulate efficiently. We emphasize this because some IA approaches take an interval to "be", for instance, the number-pair (a, b). From our viewpoint, this confuses the genuine object with its representation.

IA's primary aim is to compute enclosures (bounds) on sets. This can be said as a possibly simplistic statement about software:

PRINCIPLE 2.3 (Thou shalt not lie). *There is but one requirement for interval codes: enclose what you claim to enclose. All else are quality of implementation (QOI) issues.*

The most basic ability is to enclose the range of a function given *explicitly* by an algebraic expression or by code (rather than implicitly as the root of an equation). Functions are built up from a library of *elementary functions* among which we class the basic arithmetic operations (BAOs) $+ \; - \; \times \; \div$, as well as sin, exp, log, etc. R. Moore's (1966) basic result [8] says:

THEOREM 2.1 (Fundamental Theorem of Interval Arithmetic).
Let each elementary *function be given an interval version that for any interval inputs computes an (interval) enclosure of its exact range.*

Then evaluating an arbitrary *explicit function* $f(x, y, \ldots)$*, using these interval elementary functions, yields an (interval) enclosure of the exact range of* f *for any input intervals* $X, Y, \ldots$*, provided no exceptions occur.*

Here the exact range is by definition

$$\mathrm{range}(f\,; X, Y, \ldots) = \{ f(x, y, \ldots) \mid x \in X,\, y \in Y,\, \ldots \}.$$

This process is called *interval evaluation.* Interval practitioners tend to use the word "function" interchangeably with "expression", while being quite aware of the difference. For instance $x \times (x + 1)$ and $(x + \frac{1}{2})^2 - \frac{1}{4}$ define the same function $f(x)$, but the second gives sharper bounds than the first under interval evaluation. With input

$x = [-2, 2]$, the first gives $[-6, 6]$; the second gives $[-\frac{3}{2}, \frac{5}{2}]^2 - \frac{1}{4} = [0, 6\frac{1}{4}] - \frac{1}{4} = [-\frac{1}{4}, 6]$, the exact range. Choosing a "good form" of a function is thus of some importance. When it matters, we use "function" for a mapping in the set theory sense, and "expression" for a symbolic representation thereof.

3. Interval models. Suppose I am an implementer. For me, Moore's theorem can be stated as:

PRINCIPLE 3.1. *To compute (interval) enclosures of the range for arbitrary explicit functions it suffices to implement them for the elementary functions.*

Just what do I implement? A crucial but often ignored part of this design question is what *abstract interval model* to choose. By this we mean: assuming an ideal machine without roundoff, (a) what class $\mathcal{I}$ of intervals do I choose to represent, and (b) how do I define the elementary functions on them, in particular when exceptional cases occur?

3.1. A simple model and its defects. In some older packages

(a) The number system is $\mathbb{R}$, and $\mathcal{I}$ is the set of all *finite* closed intervals $[a, b]$ with a and b real, $a \leq b$.
(b) For $X, Y \in \mathcal{I}$ and $\bullet$ an arithmetic operation, $X \bullet Y$ is defined as the smallest interval in $\mathcal{I}$ that contains the exact range. Similarly for other elementary functions.

This model is attractive because the result of (b) *equals* the exact range for any elementary function that is continuous on its input intervals — in particular for the four arithmetic operations, except for the case of dividing by an interval containing zero.

As implemented on a machine, in (a) after "all" insert "machine-representable" and in (b) replace "the smallest" by "some" (hopefully close to the smallest, but that is a QOI issue).

There are two overlapping but distinct cases that a system based on this model cannot handle, when evaluating some elementary function e. First, if no finite interval exists that contains the exact range (on a machine, this includes overflow cases). Second, if the set defined by the inputs $X, Y, \ldots$ is not contained in e's domain of definition. Division by an interval containing zero falls into both cases.

In the first case, the system has no choice but to raise an exception. In the second case, one has the choice between two paradigms: *strict* evaluation — used in most current packages — where one just raises an exception; and *loose* or relation-style evaluation — used in the current Sun compilers' interval system [12, 13] — where one evaluates over those points that lie within the domain, ignoring the others, and continues. For instance with strict evaluation, trying to compute $\sqrt{[-2, 2]}$ causes an exception, while with loose evaluation it returns $[0, \sqrt{2}]$. Later, we give our view on how loose evaluation should be implemented.

The simple model lacks the *empty set* $\emptyset$. In older packages this was considered acceptable because typically $\emptyset$ occurs infrequently and signals some special action to be taken. (Getting $\emptyset$ as the result of a correct interval computation is important information: it is a rigorous proof that some problem, as posed, has no solution.) So for performance reasons it was considered better to avoid it by tests in algorithm code rather than to support it within the model. However, from the standpoint that intervals are sets, an interval model without $\emptyset$ is as weird as a floating point model without zero.

3.2. Effects of the IEEE standard. There began a long process of removing the restrictions of the simple model above. The most obvious need was for better handling of division by zero, hence for support of infinite intervals. Vastly influential was the IEEE 754 arithmetic standard [1], which provided for the first time standardized support of $\pm\infty$ — as well as directed rounding, which was specifically targeted at interval computation. It is easy to forget how the standard has changed the conceptual and practical numerical landscapes.

MATLAB has also changed user perception by its fairly comprehensive support of IEEE features in an interactive system. IEEE support in Fortran and C++ has also improved. Thus interval packages that do not handle infinity and division by zero now look archaic.

3.3. Unacknowledged design options. The obvious data structure for intervals is a pair of numbers holding the bounds, that is $X = [a, b]$ is represented by an object `x` with fields `x.lo` and `x.hi` holding a and b respectively. Its simplicity may be why interval literature often sees no need to specify an abstract model. This is far from the truth! Here are some Design Options — possible ingredients of a model — that are rarely spelt out.

DO1 The number system remains $\mathbb{R}$ as in the simple model, and $\mathcal{I}$ is the set of all nonempty closed intervals in $\mathbb{R}$, including the finite ones $[a, b]$, and the infinite ones that mathematicians normally denote $(-\infty, b]$, $[a, +\infty)$ and the whole line $(-\infty, +\infty)$.
Here, $\pm\infty$ are "second class" objects, mere placeholders. One is just exploiting the fact that IEEE supports them, to let the `x.lo, x.hi` data structure represent infinite as well as finite intervals.

DO2 The number system is an extended real system such as $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$, and $\mathcal{I}$ is the set of all nonempty closed intervals $[a, b]$ with $a, b \in \mathbb{R}^*$, $a \leq b$.
Here, $\pm\infty$ are "first class" objects, actual numbers one does arithmetic on. Clearly the map $X \mapsto X \cap \mathbb{R}$ is an *almost* 1-1 correspondence between intervals in DO2 and DO1, the exception being that DO2 has two degenerate intervals $[-\infty, -\infty]$ and $[+\infty, +\infty]$ with no counterpart in DO1.

DO3 One can include the empty set in $\mathcal{I}$. On an IEEE platform, representing it by `x.lo = x.hi = NaN` is a good choice that can be implemented in software with little performance penalty.

DO4 One can include *wraparound* intervals in $\mathcal{I}$. In DO1 such an interval is of the form $(-\infty, b] \cup [a, +\infty)$, and in DO2 it is $[-\infty, b] \cup [a, +\infty]$, with $b < a$ in either case.
This is a neat way to extend the scope of the `(x.lo, x.hi)` data structure. One still uses the $[a, b]$ notation, e.g., with DO1 $[2, -2]$ denotes $(-\infty, -2] \cup [2, +\infty)$.

DO5 One can have a single ∞ instead of separate $-\infty$ and $+\infty$.

DO6 One can choose loose instead of strict evaluation.

DO7 One can include signed zeros in the interval model in some way. (Our view is that, as this changes the topology of $\mathbb{R}$, it is incompatible with Principle 2.1. Therefore, however appealing on first view, it causes more problems than it solves.)

D01/D02 are mutually exclusive "radio buttons", D03 to D06 are independent "checkboxes", and D07 covers a range of possibilities. Though the options are not quite orthogonal (e.g. D01+D04 gives the same model as DO1+DO4+DO5), they give over 20 potentially usable interval models, all using the `(x.lo, x.hi)` data structure —

and we haven't reached csets yet!

We give two examples from the literature. First, Kahan (1968) [4] defines a wraparound model with a single infinity that is a first-class member of the number system: that is, choosing D02, D04 and D05. D06 is not specified. This system's attraction is that $\mathcal{I}$ is closed under all four arithmetic operations. The reciprocal of an interval containing zero is a wraparound interval, and vice versa: e.g. $[-1, 2]$ and $[\frac{1}{2}, -1]$ are reciprocals of each other.

Second, Ratz (1996) [10] defines a different system to support division by intervals containing zero. Infinities are not part of the number system, and the interval result of X/Y is defined as the interval hull of all x/y for $x \in X,\ y \in Y$ and *such that* x/y *is defined in the normal real sense, i.e.* $y \neq 0$. Ratz thus chooses D01+D03+D06 (possibly the first author to propose loose evaluation). This gives the intuitively appealing result that the quotient of intervals that don't straddle zero doesn't contain values "of unexpected sign" — for instance $[1, 3]/[0, 2] = [\frac{1}{2}, +\infty)$ and $1/[-2, 0] = (-\infty, -\frac{1}{2}]$. However $1/0$ must be empty, and since wraparound is not used, division by an interval that straddles zero must give the whole line $(-\infty, +\infty)$.

Of current and widely used interval packages, all of BIAS [6], INTLIB [5], B4M [15] and INTLAB [11] allow arithmetic with IEEE infinities. Thus, they appear to use option D02, but with inconsistencies that suggest infinity has crept into their systems by stealth, not by design. The fact that an abstract interval model is not specified in the user manual of any of these packages strengthens this impression.

4. Cset theory. Csets (containment sets) are not an interval model. They are a piece of classical real analysis from which one may construct various interval models. They give a way to construct an extended version of each elementary function — *any* such function one chooses to put in the standard library — that is unambiguously specified by the real version of the function and is guaranteed to interoperate consistently with all the other functions. They require design option DO2, but DO3 to DO6 can be chosen either way.

All one needs to define an abstract *cset model* over $\mathbb{R}$ is an extended space $\widehat{\mathbb{R}}$ that is compact (in the topology sense) and in which $\mathbb{R}$ is dense.[1] In our context compact means that every sequence in $\widehat{\mathbb{R}}$ has a subsequence converging to a point of $\widehat{\mathbb{R}}$; dense means that each point of $\widehat{\mathbb{R}}$ is the limit of a sequence of points in $\mathbb{R}$. The points in $\widehat{\mathbb{R}}$ and not in $\mathbb{R}$ are *infinite*, by definition.

To convert the model to an *interval cset model* one needs a definition of the interval hull of a set. This is obvious in $\mathbb{R}^*$, but in wraparound models one needs to resolve ambiguities.

The most practical cset model for today's CPU architectures is as follows:

(a) The number system is $\mathbb{R}^*$, and $\mathcal{I}$ is the set of all closed intervals $[a, b]$ with $a, b \in \mathbb{R}^*$, $a \leq b$, together with the empty set.

(b) For X and Y in $\mathcal{I}$, and $\bullet$ an arithmetic operation, $X \bullet Y$ is defined as the smallest interval in $\mathcal{I}$ that contains $\text{cset}(\bullet\,; X, Y)$, see below. Similarly for other elementary functions. Loose evaluation is used.

That is, DO2+DO3+DO6. Currently, this system is supported by Sun's compilers and is an option in the FILIB++ package. Unless specified otherwise, this model is used henceforth, and "interval" shall mean a member of the $\mathcal{I}$ of this model.

4.1. Operations on infinite values. We choose whether to include infinities in our number system, as we choose whether to include negative numbers or $\sqrt{-1}$.

[1] In principle one could define csets in the Stone-Čech compactification of $\mathbb{R}$.

Systems with infinities have been used for centuries, probably as far back as ancient Greek work on conic sections. Examples are the extended reals $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$, familiar from Lebesgue integral theory; the complex sphere $\mathbb{C}^\dagger = \mathbb{C} \cup \{\infty\}$, used in analytic function theory; and its real counterpart $\mathbb{R}^\dagger = \mathbb{R} \cup \{\infty\}$. Their common theme is that infinite values are treated as a limit of finite values, and the arithmetic operations *extended by continuity.*

Traditional definition. When any of x, y or z is infinite, $z = x \bullet y$ is the limit of a sequence z_r, where $z_r = x_r \bullet y_r$ is defined in the normal $\mathbb{R}$-sense for all r, and $x_r \to x$ and $y_r \to y$, *provided this limit is unique.*

The reason for the last clause is the wish that the extended operation remain a *function*, i.e. return a single-point result from single-point inputs. For $\mathbb{R}^*$ this means that for the basic arithmetic operations, the following are traditionally *undefined*:

$$(+\infty) + (-\infty),\ (+\infty) - (+\infty),\ (-\infty) - (-\infty),$$
$$0 \times (\pm\infty),\ x/0 \text{ for any } x,\ (\pm\infty)/(\pm\infty),$$

and obvious permutations of these. All other combinations of finite and infinite values give a defined result. IEEE arithmetic is based on $\mathbb{R}^*$, and gives the same results except where its signed zeros are involved.

Csets just take extension by continuity a step further by removing the "unique limit" clause, thus allowing single point inputs to return a multi-point result, namely:

Cset definition. For an operation $z = x \bullet y$, the cset value at (x, y) is the set of *all possible limits* of sequences z_r, where $z_r = x_r \bullet y_r$ is defined in the normal $\mathbb{R}$-sense for all r, and $x_r \to x$, $y_r \to y$. The cset for set inputs, $X \bullet Y$, is the union of the csets for points $x \in X$ and $y \in Y$. For a function $f(x, y, \ldots)$ of any number of arguments, the cset is defined analogously.

For the arithmetic operations this gives the new results $x/0 = \{-\infty, +\infty\}$ if $x \neq 0$, $+\infty/+\infty = [0, +\infty]$ and $(+\infty) + (-\infty) = 0 \times (+\infty) = 0/0 = \mathbb{R}^*$. It also adds extra values at each point where a function is finite but discontinuous, e.g. the function $\operatorname{sign}(x) = -1\,(x < 0); 0\,(x = 0); 1\,(x > 0)$ has the cset value $\{-1, 0, 1\}$ at $x = 0$.

Notation. We write the cset of $f(x, y, \ldots)$ as $\operatorname{cset}(f; x, y, \ldots)$ or as $f^*(x, y, \ldots)$ when necessary, but use standard notation if the meaning is clear, as in the previous paragraph.

4.2. Features of csets. Four facts about csets seem especially noteworthy, so we state them as Principles. First, the ordinary value of a function at a point, if it has one, is always among the cset values there, whence a simple but important fact:

PRINCIPLE 4.1. *The cset always encloses the traditional (exact) range, and coincides with it when the function is continuous at each point of the input set.*[2]

The crucial foundation of the theory is an analogue of Moore's Theorem 2.1. Its proof is outlined in Subsection 4.4.

THEOREM 4.1 (Fundamental Cset Theorem). *Let each* elementary *function be given a cset interval version that computes an (interval) enclosure of its cset over arbitrary interval inputs.*

Then evaluating an arbitrary *explicit function $f(x, y, \ldots)$, using these interval elementary functions, yields an (interval) enclosure of the cset of f over any input intervals $X, Y, \ldots$.*

[2]This is not the same as "the function, restricted to the input set, is continuous". Let $f(x)$ be the "floor" function, defined as the largest integer $\leq x$. The restriction of f to $X = [0, \frac{1}{2}]$ is continuous, but f is not continuous at each point of X.

Note an important difference from Theorem 2.1 — the clause "provided no exceptions occur" is absent. We discuss the pros and cons of this in Subsection 5.2.

The following briefer result is a variant on Theorem 4.1: either can be deduced from the other with a little effort. Namely, under suitable compactness assumptions,

$$f(x) = g(h(x)) \quad \text{implies} \quad f^*(x) \subseteq g^*(h^*(x)). \tag{4.1}$$

Looked at from the implementer's viewpoint, Theorem 4.1 says:

PRINCIPLE 4.2. *To compute (interval) enclosures of the cset for arbitrary functions, it suffices to implement them for the elementary functions.*

Third, Principle 4.1 implies csets simply *extend* the capabilities of traditional intervals. A simplistic statement of this is as follows.

PRINCIPLE 4.3. *Let a program use a "traditional" interval library of elementary functions. Let it be modified by replacing this by a corresponding "cset" library. Then for any input data on which the original program runs without exceptions, the modified program produces identical output.*

This holds in exact arithmetic, and with some restrictions on use of discontinuous elementary functions such as "sign" and on control flow constructs — e.g. branching on the occurrence of a zerodivide exception is not allowed. To obtain "identical output" in inexact arithmetic requires making each cset library function have identical rounding properties to those of its traditional counterpart, which is possible but rather pointless. A more practical statement would replace "identical" by some concept of "identical within roundoff".

Fourth, showing that csets do things that other interval systems cannot:

PRINCIPLE 4.4 (The Rearrangement Theorem). *The cset of any rational function is unchanged by rearranging it into any algebraically equivalent form.*

This is proved in [9]. It is not about intervals as such, and it applies in $\mathbb{R}^\dagger$ systems (with a single infinity) as well as in $\mathbb{R}^*$. It gives clever compilers and humans the chance to rearrange code for tighter enclosures. E.g., $f(x, y) = x^2/(x^2 + y^2)$ has the same cset as $g(x, y) = 1/(1 + (y/x)^2)$, the latter giving a tighter result in interval arithmetic for most inputs as this table shows:

	$X = [-1, 2], Y = [0, 2]$		$X = [1, 3], Y = [0, 2]$	
	$f(X, Y)$	$g(X, Y)$	$f(X, Y)$	$g(X, Y)$
INTLAB	[NaN, NaN]	[NaN, NaN]	[0.0769, 9.0000]	**[0.1999, 1.0000]**
Cset	$[-\infty, +\infty]$	**[0, 1]**	[0.0769, 9.0000]	**[0.1999, 1.0000]**

These results were produced in MATLAB using the "short" display option, with S. Rump's INTLAB and Pryce's trial cset system. The correctly rounded exact cset, where computed, is in bold. For $X = [-1, 2], Y = [0, 2]$ INTLAB does not handle the divisions by zero; the cset system does, and using g it produces the exact cset. For $X = [1, 3], Y = [0, 2]$ there are no divisions by zero; on both INTLAB and the cset system, g returns the correct value while f gives a far worse enclosure.

4.3. The graph view of functions. The main difficulty in proving Theorem 4.1 is notation, which we now define in a way that, hopefully, clarifies the neat idea at the heart of the proof. We must describe precisely how $\mathbf{f}$ is built up from elementary functions. From its n *input* variables x_j, $\mathbf{f}$ computes a number, say p, of *intermediate* variables v_k and finally the m *output* variables y_i. Write this as a sequence of equations — a *code list*:

$$v_k = e_k(\text{suitable previously defined } v_l), \quad k = 1, \ldots, (p+m), \tag{4.2}$$

MATLAB-style code	Graph form
`function [y1, y2] = f(x1, x2, x3)`	
`v1 = x1 * x2`	$(x_1, x_2, v_1) \in$ "×"
`v2 = sin(v1)`	$(v_1, v_2) \in \sin$
`v3 = 2 * v2`	$(2, v_2, v_3) \in$ "×"
`v4 = v3 - x1`	$(v_3, x_1, v_4) \in$ "−"
`y1 = x3 * v4`	$(x_3, v_4, y_1) \in$ "×"
`y2 = 3 * v4`	$(3, v_4, y_2) \in$ "×"

FIG. 4.1. *A simple function* $\mathbf{y} = \mathbf{f}(\mathbf{x})$.

where the e_i are elementary functions. To simplify the notation, just in this equation, $v_{1-n}, \dots, v_0$ have been used as aliases for $x_1, \dots, x_n$ and $v_{p+1}, \dots, v_{p+m}$ as aliases for $y_1, \dots, y_m$, respectively. Code containing over-written variables, constant or global values, branches, and loops, can be put in this framework, but explaining how would digress too far.

We view a function $\mathbf{f}$ as being the same as its *graph*, that is the set of all $(\mathbf{x}, \mathbf{y})$ such that $\mathbf{f}(\mathbf{x})$ is defined and equal to $\mathbf{y}$. Thus

$$y = f(x) \quad \Leftrightarrow \quad (x, y) \in f.$$

We do similarly for each elementary function. For the arithmetic operators, we enclose the name in quotes for clarity, so for instance "×" is the set in real 3-space defined by

$$w = u \times v \quad \Leftrightarrow \quad (u, v, w) \in \text{“}\times\text{”}.$$

Converting the code list to graph form, we need to separate the roles of x's, v's and y's, so we shall write (4.2) in the form

$$(\mathbf{x}, \mathbf{v}, \mathbf{y})_{\text{args}_k} \in e_k, \quad k = 1, \dots, (p+m), \tag{4.3}$$

where "args_k" selects the appropriate components from the vector $(\mathbf{x}, \mathbf{v}, \mathbf{y})$ to form the list (input(s) to e_k, output(s) from e_k) — essentially an array subscripting process. Figure 4.1 illustrates the conversion to graph form for a simple function.

We now have the relation between the graph of $\mathbf{f}$ and the graphs of the elementary functions of which it is built:

$$(\mathbf{x}, \mathbf{y}) \in \mathbf{f} \quad \Leftrightarrow \quad \begin{array}{l} \text{there exists a vector } \mathbf{v} \text{ such that } (\mathbf{x}, \mathbf{v}, \mathbf{y})_{\text{args}_k} \in e_k, \\ k = 1, \dots, (p+m). \end{array} \tag{4.4}$$

Conceptually, (4.4) describes the same process as *navigating* a relational database to answer a query — the relations being the elementary functions.

4.4. Proof of the main theorem. The machinery is now set up. The definition of the cset $\mathbf{f}^*$ of $\mathbf{f}$ implies $(\mathbf{x}, \mathbf{y})$ is in (the graph of) $\mathbf{f}^*$ iff there exist sequences of vectors $\mathbf{x}^{(r)}$, $\mathbf{y}^{(r)}$ such that $\big(\mathbf{x}^{(r)}, \mathbf{y}^{(r)}\big) \in \mathbf{f}$ and $\mathbf{x}^{(r)} \to \mathbf{x}$ and $\mathbf{y}^{(r)} \to \mathbf{y}$ as $r \to \infty$. This is just the definition of topological closure, that is

For any function $\mathbf{f}$, the graph of the cset of $\mathbf{f}$ is the closure of the graph of $\mathbf{f}$. (4.5)

Let $(\mathbf{x}, \mathbf{y})$ be an arbitrary point in $\mathbf{f}^*$ and choose sequences $\mathbf{x}^{(r)}$ and $\mathbf{y}^{(r)}$ as above. From (4.4), for each r there exists a vector of intermediate variables $\mathbf{v}^{(r)}$ such that

$$\Big(\mathbf{x}^{(r)}, \mathbf{v}^{(r)}, \mathbf{y}^{(r)}\Big)_{\text{args}_k} \in e_k\,. \tag{4.6}$$

Here is the neat idea. $\mathbb{R}^*$ is a compact topological space, so we may assume, by repeatedly taking subsequences of the original sequence, that each component $\mathbf{v}_k^{(r)}$ ($k = 1,\ldots,p$) of $\mathbf{v}^{(r)}$ converges to some point of $\mathbb{R}^*$; that is, that the sequence $\left(\mathbf{v}^{(r)}\right)$ converges in $(\mathbb{R}^*)^p$ to some $\mathbf{v}$. This implies the sequence $\left(\mathbf{x}^{(r)},\mathbf{v}^{(r)},\mathbf{y}^{(r)}\right)_{\mathrm{args}_k}$ in (4.6), being a selection of components of convergent sequences, converges to $(\mathbf{x},\mathbf{v},\mathbf{y})_{\mathrm{args}_k}$. Thus applying (4.5), $(\mathbf{x},\mathbf{v},\mathbf{y})_{\mathrm{args}_k}$ is in the cset version e_k^* of e_k for each k. We have thus proved that

$$(\mathbf{x},\mathbf{y}) \in \mathbf{f}^* \quad \text{implies} \quad \begin{array}{l}\text{there exists a vector } \mathbf{v} \text{ such that } (\mathbf{x},\mathbf{v},\mathbf{y})_{\mathrm{args}_k} \in e_k^*,\\ k = 1,\ldots,(p{+}m).\end{array} \tag{4.7}$$

This is the definition of $\mathbf{y}$ being a member of the cset evaluation of $\mathbf{f}$ at $\mathbf{x}$. Apart from details about evaluating over a set rather than at one point, this proves the variant of Theorem 4.1 that results from deleting the word "interval" wherever it occurs. For the theorem as stated, replacing each set by an (interval) enclosure at each step through the code list can only make the output sets at each step larger. Thus if the non-interval version gives an enclosure, so must the interval version.

5. Issues of theory and implementation.

5.1. Improving division by zero. One effect of our $\mathbb{R}^*$ interval model is that $x/0$ for $x \neq 0$ neither causes an exception as in the simple model of Subsection 3.1; nor is the single point ∞ as in Kahan's wraparound model; nor is empty as in Ratz' model — it is the set $\{-\infty,+\infty\}$. Hence after taking the interval hull, the result of division by an interval containing 0 is always the whole line. When 0 is an end of the interval, both Kahan's and Ratz' model do better, while Kahan, but not Ratz, does better for dividing by an interval with 0 in its interior. The following table illustrates what happens. "!" denotes an exception. For the "exact cset", wraparound notation has been used, e.g. $[\frac{1}{3},-\infty]$ denotes $\{-\infty\} \cup [\frac{1}{3},+\infty]$.

	Model and result				
To compute	Simple	Kahan	Ratz	Exact $\mathbb{R}^*$ cset	Our model
$1/0$	!	∞	$\emptyset$	$\{-\infty,+\infty\}$	$\mathbb{R}^*$
$1/[0,3]$	!	$[\frac{1}{3},\infty]$	$[\frac{1}{3},+\infty)$	$[\frac{1}{3},-\infty]$	$\mathbb{R}^*$
$1/[-1,3]$	!	$[\frac{1}{3},-1]$	$\mathbb{R}$	$[\frac{1}{3},-1]$	$\mathbb{R}^*$

This defect is annoying but not fatal. It is caused, not by csets themselves, but by how csets interact with the interval hull operation.

One solution is to change the hull operation by adding wraparound intervals to the model. John Pryce has a proof-of-concept implementation of this model under MATLAB, following the design of INTLAB. Wraparound complicates the logic of most elementary functions including the BAOs. On current machines this inevitably impacts performance, but with hardware-implemented intervals this need not be so.

We outline an attractive alternative, namely to change the meaning of "cset" while staying within cset theory. What we do is to change the graphs of the elementary functions, by *range-constraints* (mathematically, these perhaps should be domain-constraints) on interval variables. This is a set within which all values of the variable inherently lie, for instance $[0,+\infty]$ for the property of being *non-negative*, as with physical quantities such as mass, electrical resistance, or chemical concentration.

How we exploit this depends on features of the programming language used. We assume a language with variable types determined at compile time and (as in Fortran

but not C++) the type of a function's output determined only by the type of the inputs and not by the type of the variable to which the result is assigned. We assume there is a small number of distinct constraint sets, known at compile time. Let `NONNEG` denote the non-negative type

Let C_x denote the range-constraint set of variable x (defaulting to $\mathbb{R}^*$). Then the *range-constrained graph* of an elementary function $w = e(x, y, \ldots)$ is just that subset of the normal graph for which $x \in C_x, y \in C_y$, etc. The (graph of the) cset is redefined to be the closure of the range-constrained graph.

For instance, the graph of division, when the divisor is inherently non-negative, is the set $G = \{\, (x, y, z) \in \mathbb{R}^3 \mid z = x/y \text{ is defined and } y \geq 0 \,\}$, which of course forces y to be > 0. The cset at $x = 1$ and $y = 0$, that is the value of $1/0$ under this constraint, is the set of limits of $z_r = x_r/y_r$ such that $x_r \to 1$, $y_r \to 0$ and $y_r > 0$. This is clearly just the singleton $+\infty$ instead of the unconstrained cset value $\{-\infty, +\infty\}$. We achieved this without changing the topology of $\mathbb{R}$ — no signed zeros needed.

For such a system to give useful answers with extended expressions, output from elementary functions must carry a range constraint when appropriate. E.g. $+\ \times\ \div$, on `NONNEG` inputs, and sqrt, exp and raising to an even power, with any inputs, should return `NONNEG`.

Details to be worked out include: what should be done at run time when a constrained (interval) variable is assigned a value that violates its constraint? Probably the value should be intersected with the constraint and the `DISCONTINUOUS` flag (see below) raised.

As a further example, consider two guises of the same formula:

$$\frac{1}{f} = \frac{1}{u} + \frac{1}{v} \quad \text{or} \quad \frac{1}{r} = \frac{1}{r_1} + \frac{1}{r_2}. \tag{5.1}$$

The first formula relates source and image for an ideal lens. f is its focal length, conventionally positive for a converging lens and negative for a diverging one. Light comes from the negative x direction (the left) and passes through the lens. The source is a distance u to the lens' left, and the image a distance v to its right. All six mathematically possible combinations of sign of u, v and f make physical sense for various combinations of "real" or "virtual" source or image. Any of f, u or v can be zero or infinite, with the exception that $f = 0$ is not meaningful. Since a slight perturbation of source position or focal length can make the image shoot to infinity in one direction and reappear from the other direction, a wraparound model is the most natural for interval computation with this equation.

The second formula of (5.1) describes the resistance r of a circuit made up of resistors r_1 and r_2 in parallel. All of r_1, r_2 and r are inherently ≥ 0, but physically any of them can be zero (a short-circuit) or infinite (perfect insulation). We assume r_1 and r_2 are the input, and r is evaluated as

$$r = 1/(1/r_1 + 1/r_2). \tag{5.2}$$

This is one of the advertisements for IEEE signed zero, but range-constraints handle it too:

- Using IEEE (point) arithmetic one must ensure that a zero resistance is taken as $+0$. Then with (5.2), any combination of values $r_1, r_2 \in [+0, +\infty]$ gives the physical correct result within roundoff, again in $[+0, +\infty]$.
- Using unconstrained interval cset arithmetic, (5.2) gives a very pessimistic enclosure when both inputs are, or contain, 0 or $+\infty$. This is even true

when exact csets or wraparound intervals are used, namely $r_1 = r_2 = 0$ gives $r = 1/(\{-\infty, +\infty\} + \{-\infty, +\infty\}) = 1/\mathbb{R}^* = \mathbb{R}^*$, while $r_1 = r_2 = +\infty$ gives $r = 1/(0 + 0) = 1/0 = \{-\infty, +\infty\}$.

- Using interval cset arithmetic with r_1, r_2 range-constrained to $[0, +\infty]$, (5.2) gives the correct result in all cases. For instance $r_1 = r_2 = 0$ gives $r = 1/(+\infty + +\infty) = 1/+\infty = 0$ irrespective of constraints on the intermediate results. And $r_1 = r_2 = +\infty$ gives $r = 1/(0 + 0)$ with both 0's constrained to $[0, +\infty]$, reducing to $1/0$ with the 0 constrained to $[0, +\infty]$, giving $+\infty$.

5.2. Loose evaluation and its flag.

A selling point of cset systems is exception-free execution (EFE). This is not a feature of csets as such, but of choosing loose over strict evaluation, see Subsection 3.1. The result of interval-evaluating a complicated function is guaranteed to enclose the value at each point of the input intervals *where the value is defined.* A traditional interval system could behave in this way, provided it supports the empty set (to handle cases such as $\sqrt{[-2, -1]}$).

Interval researchers have discussed the pros and cons of EFE for some years. An important area is validated global optimization. For some algorithms in this area, loose evaluation is appropriate because they bound the set of *defined* values of the objective function over a region. Points where the function is undefined are irrelevant. Bill Walster, architect of the Sun compilers' interval system [12, 13], has an optimization background and regards plain EFE as sufficient.

However, some interval applications benefit enormously from compiler-supported checks that a function is defined and/or continuous over a set. An example is validated solution of differential equations, which typically prove existence and enclosure of a solution by means of Brouwer's Theorem: if B is an n-dimensional box (vector interval), and $\mathbf{f} : B \to B$ is *defined* and *continuous* on the whole of B, then $\mathbf{f}$ has a fixed point in B. Here is an example from Markus Neher (private communication). Let $f(x) = \sqrt{x} - 1$ and $B = [-4, 4]$. Strict evaluation raises an exception, but with loose evaluation $f(B)$ evaluates as $[-1, 1]$, seeming to show the Brouwer conditions hold. If they did, there would be $x \in [-4, 4]$ with $\sqrt{x} - 1 = x$, which is not so.

Walster has argued that a programmer should check "manually" that the Brouwer conditions are satisfied. The cost may be prohibitive, however, as the code for $\mathbf{f}$ may be very long: consider converting a complicated PDE to an ODE by method-of-lines in order to use a validated ODE solver.

What is required is a `DISCONTINUOUS` flag on the same footing as the IEEE flags `OVERFLOW, INEXACT,` etc. The code for $\mathbf{f}$ lowers the flag on entry. An interval elementary function raises the flag if it receives arguments outside the set where it is *defined and continuous.* If on exit from $\mathbf{f}$ the flag is unraised, $\mathbf{f}$ is defined and continuous on $\mathbf{X}$. If the result interval is contained in $\mathbf{X}$, this constitutes a rigorous proof that Brouwer applies. Figures 5.1 and 5.2 show examples using Pryce's trial MATLAB system, for Neher's example and for the function $\sqrt{x} - 0.16$, which has two fixed points, at 0.04 and 0.64.

For applications that need it, the value of the information should far outweigh any speed penalty. As with IEEE flags, this flag should be switchable between a silent mode (the default) and one that throws an exception. There should be a way to remove it and its overhead entirely for those applications that do not require it.

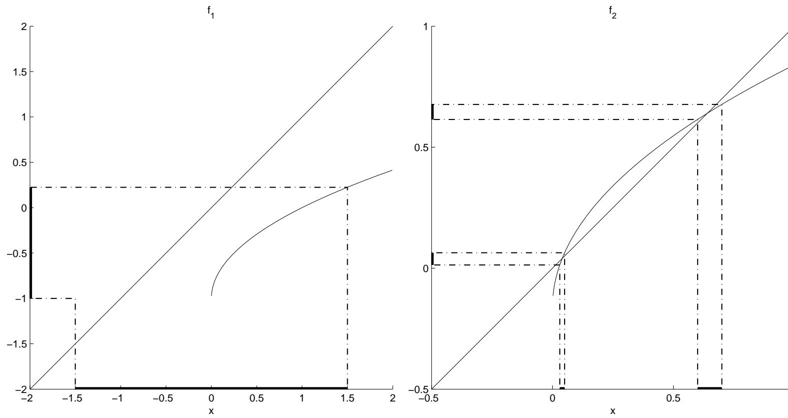

FIG. 5.1. *Left:* $f_1(x) = \sqrt{x} - 1$ *and image of* $[-1.5, 1.5]$. *Right:* $f_2(x) = \sqrt{x} - 0.15$ *and images of* $[0.03, 0.05]$ *and* $[0.6, 0.7]$.

```
function [y,flag] = f1(x)
cwresetdc
y = sqrt(x) - 1;
flag = cwgetdc;
```

(*i*)

```
x = cwinfsup(-1.5,1.5)
cwintval x = [ -1.5000, 1.5000]
[y,flag] = f1(x)
cwintval y = [ -1.0000, 0.2248]
flag = 1
```

```
function [y,flag] = f2(x)
cwresetdc
y = sqrt(x) - 0.16;
flag = cwgetdc;
```

(*ii*)

```
x = cwinfsup(0.03,0.05)
cwintval x = [ 0.0299, 0.0501]
[y,flag] = f2(x)
cwintval y = [ 0.0132, 0.0637]
flag = 0
```

(*iii*)

```
x = cwinfsup(0.6,0.7)
cwintval x = [ 0.5999, 0.7000]
[y,flag] = f2(x)
cwintval y = [ 0.6145, 0.6767]
flag = 0
```

FIG. 5.2. *Applying functions* f_1, f_2. *(i) No fipoint;* $f_1(B) \subseteq B$ *but flag raised. (ii)* B *contains fixpoint; flag not raised but* $f_2(B) \not\subseteq B$. *(iii)* B *contains fixpoint; flag not raised,* $f_2(B) \subseteq B$, *so Brouwer proved to apply. (*`cwresetdc` *lowers* `DISCONTINUOUS` *flag;* `cwgetdc` *gets its value.)*

5.3. Subtleties of rearrangement. Principle 4.4 gives new opportunities to optimize code. With csets, compilers have freedom to rearrange expressions to reduce, sometimes dramatically, effects of dependency, one of the major contributors to overestimation in any interval algorithm.

Typically, within code that defines a function $\mathbf{f}_0$, an optimizing compiler seeks a "window" defining a function $\mathbf{h}_0$ that it can replace by a "better" function $\mathbf{h}_1$ with the same cset. The mathematics underpinning this is the following, where $\mathbf{g}$ and $\mathbf{k}$ represent the code within $\mathbf{f}_0$ before and after $\mathbf{h}_0$. Since these functions denote symbolic expressions rather than set mappings, composition $\mathbf{h}(\mathbf{g}(\mathbf{x})))$ means code-concatenation: "execute $\mathbf{g}$, then execute $\mathbf{h}$".

THEOREM 5.1 (Window Optimization). *Let* $\mathbf{f}_0(\mathbf{x}) = \mathbf{k}(\mathbf{h}_0(\mathbf{g}(\mathbf{x})))$, $\mathbf{f}_1(\mathbf{x}) = \mathbf{k}(\mathbf{h}_1(\mathbf{g}(\mathbf{x})))$, *and* $\mathbf{h}_1^* = \mathbf{h}_0^*$. *Then interval cset evaluation of* $\mathbf{f}_1$ *encloses the cset of* $\mathbf{f}_0$.
The proof is simple, using Theorem 4.1 and its variant, (4.1). It may be extended by

induction to handle any number of such replacements within a given function.

In Theorem 5.1 the csets of $\mathbf{f}_1$ and $\mathbf{f}_0$ need *not* be equal. For a trivial example, let scalar functions $w = f_0(x)$ and $w = f_1(x)$ be defined by:

$$x \xrightarrow{\mathbf{g}(x)=(x,x)} \mathbf{y} \xrightarrow[h_0(y_1,y_2)=1/(1/(y_1-y_2))]{h_1(y_1,y_2)=y_1-y_2} z \xrightarrow{k(z)=z} w$$

Then f_0 is everywhere undefined (empty) while f_1 is everywhere zero.

This justification of rearrangement in the cset context is very satisfactory. However, a subtle semantic difficulty remains to be resolved. A cset system, on the lines here described, will execute code and produce output intervals from input intervals. But what function is actually defined by a section of code? This goes to the heart of Principle 2.3. If we don't know what set a computation is aiming to enclose, we cannot verify whether it does so correctly.

Code has a mixture of variables and constants, and the difficulty is the relation between them in the cset context. It is implicit in the example of converting to graph form in Figure 4.1. One of the lines is

$$(3, v_4, y_2) \in \text{“}\times\text{”}.$$

When converting this to a converging sequence as in the proof of Theorem 4.1, does the 3 (a) stay as is, or (b) become a sequence of values converging to 3? This is not a question about what is computed at run time, but about its *meaning*.

Choosing (a) forbids us to give a (nonempty) value to constant expressions that do not evaluate over the reals. For instance the expression `x/0`, where `x` is of interval type with value $[1, 1]$, cannot mean the cset of x/y at $x = 1, y = 0$. The graph formalism shows it equals the set of all limits of z_r where $x_r \to 1$ and $(x_r, 0, z_r) \in \text{“}\div\text{”}$. There are no such $(x_r, 0, z_r)$, so the result must be empty.

Choosing (b) is attractive. It treats interval constants as "hidden variables" that are predefined to range over the given interval, first promoting any point constants in interval expressions to intervals as is done in most current systems. It gives the expected cset value to expressions such as `x/0`, where `x` is of interval type. It gives a consistent meaning to functions containing interval constants, e.g., the cset value of

$$f(x) = \big([1, 2]\big)/\big(x - [3, 4]\big)$$

at a point x, by this definition, comprises all limits of sequences $c_1^{(r)}/(x^{(r)} - c_2^{(r)})$ where $x^{(r)} \to x$, and $c_1^{(r)}$ and $c_2^{(r)}$ converge to some point in $[1, 2]$ and $[3, 4]$, respectively.

Sadly, (b) violates the Rearrangement Theorem. If `x` is an interval variable, the expression `(x-2)/(x-2)` defines a function whose cset at $x{=}2$ is the set of limits of

$$y^{(r)} = \big(x^{(r)} - c_1^{(r)}\big)/\big(x^{(r)} - c_2^{(r)}\big)$$

where all of $x^{(r)}, c_1^{(r)}, c_2^{(r)}$ converge to 2. This cset is all of $\mathbb{R}^*$, but since $(x-2)/(x-2)$ simplifies to 1, the cset should be the single point 1.

Our recommended definition of the mathematical *meaning* of expressions and code is that interval constants and the infinite point constants $\pm\infty$ continue to represent "hidden variables"; while real point constants in interval expressions just mean themselves. This saves the Rearrangement Theorem.

This semantic definition constrains how a compiler converts cset-based interval source code to object code. Whether optimizing or not, it must compute *enclosures*

of the mathematical meaning of each assigned variable. To illustrate, consider the following code fragments in C- or MATLAB-style, where `interval(a)` denotes the constructor of the interval object $[a, a]$ from the real value a.

1.
```
y=(x-2)/(x-2);
```
2.
```
y=(x-interval(2))/(x-interval(2));
```
3.
```
y=(x-3+1)/(x-2);
```
4.
```
c=interval(2);
y=(x-c)/(x-c);
```

It is *valid* to generate the same code for (1) and (2), but when optimizing, the compiler can and should treat them differently. All of (1), (3) and (4) can be simplified to `y = 1;` , but (2) can not be, because the two occurrences of `interval(2)` must denote different hidden variables.

Consistently with this approach, one could give an optimizer knowledge of the exact meaning of basic constants such as `Pi` representing π. It could then for instance reduce `sin(x+Pi)` to $-\sin(x)$. Thus, the methods of "exact computational geometry" can contribute significantly to interval computation.

To understand our rationale for defining all interval constants (including ones of zero width) to represent "hidden variables," i.e. that interval constants represent bounds on csets, consider the mathematical expression

$$y = \frac{x - [1.9, 2.1]}{x - [1.9, 2.1]}. \tag{5.3}$$

The interval constant $[1.9, 2.1]$ in the numerator could represent the same (imprecisely known) value or a different (imprecisely known) value as the interval constant $[1.9, 2.1]$ in the denominator. Without additional syntax in the programming language, there is no way to know which. With our recommendation to interpret each occurrence of an interval constant as a separate hidden variable, the resulting value will contain the correct value, regardless of which interpretation is used (and hence, the compiler will never "lie," when examining a single statement.) Furthermore, this syntactic interpretation avoids complications of other interpretations. Finally, this does not constrain the programmer from explicitly spelling out the intent. For example, if the two occurrences of $[1.9, 2.1]$ in (5.3) were to represent the same imprecisely known value, then the programmer could impart this to the compiler with a sequence of statements, such as

```
c = interval(1.9,2.1);
y = (x-c)/(x-c);
```

The compiler could then simplify `(x-c)/(x-c)` to 1, consistent with our recommended semantics. In particular, such a simplification would be consistent with interpreting a variable as a single, albeit possibly imprecisely known, value. The important thing here is that the programmer understand the semantic distinctions and know how the compiler is interpreting the syntax elements.

We believe the existing Sun and FILIB++ interval systems require no change to comply with these semantics. A possible exception is how interval constructors behave when an argument has the value NaN — which is not a rearrangement issue.

In summary, the rearrangement feature of cset systems offers great benefits for interval computation. However, the underlying mathematical meaning of code — which with optimizing compilers is *always* different from its raw computational meaning — must be defined more precisely than it sometimes is in language specifications.

5.4. Performance issues. With current machine architectures, all interval operations done in software are substantially slower than the corresponding operations on floating point numbers, but the gap is narrowing as algorithms and hardware support improve. Cset interval features incur a further penalty compared with the simple model of Subsection 3.1.

However, this is not inherent. People used to assert that IEEE arithmetic was impracticably slow. It began to be implemented in hardware, and now no one would wish to do without it for general numerical computing. Cset interval arithmetic in hardware would have similar benefits.

Short of this, some pointers to improving cset performance are as follows. On most current "superscalar" machines, what hurt performance most are operations that flush the pipeline, such as branching and changing the rounding mode. First, for rounding, Rump's INTLAB work shows that vectorizing, so as to set the rounding only once or twice per vector, improves performance significantly. A data structure holding $[a, b]$ as $(-a, b)$ lets one vastly reduce rounding mode changes — compiler support can eliminate them entirely in a sequence of purely interval operations. Second, for the interval operations $+\ -\ \times$ in IEEE arithmetic the $\mathbb{R}^*$ cset results differ from those of the simple model exactly in those cases where NaN is produced. There are ways to implement them with few branches. For FILIB++, tests [14] show the speed difference between the cset and non-cset options is quite small. Branimir Lambov [7] describes ways to use the x86 instruction set to do interval operations in "packed registers", and by case-selection without branching, reducing the penalty further.

The revised arithmetic standard IEEE 754R [2] mandates max and min operators with the property that if one argument is NaN, the result is the other argument — these can be used to speed up cset operations. Finally, there is a small change to IEEE 754 that would completely remove the speed difference between cset and ordinary interval arithmetic. Namely, if the rounding is set to "down" [respectively "up"], an operation that at present returns NaN should return the smallest [resp. largest] value of the cset result. For instance $+\infty/+\infty$ should return 0 when rounded down and $+\infty$ when rounded up. It is late in the current revision round, but this proposal has been submitted to the IEEE 754 panel.

Ideally, hardware should be able to use both the $\mathbb{R}^*$ model with two infinities, and the $\mathbb{R}^\dagger$ model with one; and to work with or without wraparound intervals.

6. Summary. Interval arithmetic systems need to be founded on a clear model of what intervals are supported and how operations are defined. They should handle infinity using a consistent theory, grounded in standard real analysis so as to permeate the general scientific computing community. Csets are such a theory, from which one may construct several practical interval models. The idea is to extend the graph of a function of real variables by topological closure in a suitably extended space.

In $\mathbb{R}^*$, the cset arithmetic operations are sufficiently close to the IEEE ones that they can be implemented efficiently. At least three implementations currently exist: those of Sun for Fortran and C++ and that of FILIB++ for C++.

For cset intervals, the Fundamental Cset Theorem plays the role of Moore's Fundamental Theorem for traditional intervals, showing (roughly speaking) that to enclose any function it suffices to enclose the elementary functions.

The cset of a function $\mathbf{f}$ over an input set $\mathbf{X}$ contains the classical range $\mathbf{f}(\mathbf{X})$. In practical computing, the two are equal most of the time. If a program executes successfully using a traditional interval system then it does so using a cset system, and gives identical results "within roundoff" assuming both systems have tight im-

plementations of the elementary functions (plus other provisos).

A feature not found in other interval systems is the Rearrangement Theorem — rearranging a rational expression to algebraically equivalent form does not change its cset. This gives optimizing compilers new freedom to rearrange expressions to reduce, sometimes dramatically, effects of dependency. To define what optimizations are valid, the semantics of constants in expressions must be specified carefully.

Cset systems are normally implemented with "loose evaluation", giving exception-free execution even when evaluating functions outside their domains. This feature must be supplemented by a `DISCONTINUOUS` flag, used for instance to justify the application of Brouwer's theorem.

Cset interval systems are currently slightly slower than traditional ones but need not be; for basic arithmetic on IEEE 754 machines, the penalty can be removed entirely by a small change to the standard.

7. Acknowledgements. We are grateful for important contributions to this article made by discussions with the other members of the ISL team and with: Bill Walster (Sun Microsystems); Guillaume Melquiond (Lyon) and Sylvain Pion (INRIA); Markus Neher (Karlsruhe); Jacques Carette (McMaster); Paul Taylor (Manchester) and Andrej Bauer (Ljubljana); and many others.

REFERENCES

[1] American National Standards Institute. IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std. 754–1985. New York, 1985.

[2] American National Standards Institute. Proposed revision of IEEE 754. `754r.ucbtest.org`

[3] Michael Beeson and Freek Wiedijk. The meaning of infinity in calculus and computer algebra systems. In *Proceedings of the Joint International Conferences on Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, pages 246–258. Springer-Verlag, 2002.

[4] W. M. Kahan. A more complete interval arithmetic. In *Lecture Notes for a Summer Course at the University of Michigan*, 1968.

[5] R. B. Kearfott, M. Dawande, K.-S. Du, and C.-Y. Hu. Algorithm 737: INTLIB, a portable FORTRAN 77 interval standard function library. *ACM Trans. Math. Software*, 20(4):447–459, December 1994.

[6] O. Knüppel. BIAS – basic interval arithmetic subroutines. Technical Report 99.3, Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, Technische Universität Hamburg-Harburg, 1993.

[7] B. Lambov. Talk at Dagstuhl seminar 06021, Jan 2006. BRICS, Department of Computer Science, University of Aarhus, Denmark. `barnie@brics.dk`

[8] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood, N.J., 1966.

[9] J. D. Pryce. An introduction to containment sets. Technical report DoIS/TR01/05, Department of Information Systems, Shrivenham Campus, Cranfield University, Swindon SN6 8LR, UK, 2005.

[10] D. Ratz, Inclusion-isotone extended interval arithmetic—a toolbox update. Institut für Angewandte Mathematik, Univ. Karlsruhe (TH), Germany, 1996. Available at `ftp://iamk4515.mathematik.uni-karlsruhe.de/pub/documents/reports`

[11] S. M. Rump. INTLAB Interval Toolbox, version 5.2. Technical University Hamburg-Harburg, Germany, 1999–2006. `www.ti3.tu-harburg.de/intlab.ps.gz`

[12] Sun Microsystems. Interval Arithmetic in the Forte[tm] C++ Compiler. `www.sun.com/forte/cplusplus/interval/`, 2000.

[13] Sun Microsystems. Interval Arithmetic in the Forte[tm] Fortran 95 Compiler. `www.sun.com/forte/fortran/interval/`, 2000.

[14] J. Wolff von Gudenberg. Private communication at Dagstuhl seminar 06021, Jan 2006. Institut für Informatik III, Universität Würzburg, Würzburg, Germany. `wolff@informatik.uni-wuerzburg.de`

[15] Jens Zemke. b4m: A free interval arithmetic toolbox for MATLAB. Technical report, Technische Universitaet Hamburg-Harburg, Technische Informatik III, Eissendorfer Str. 38, 21071 Hamburg, Germany, 1998. Available at `http://www.ti3.tu-harburg.de/~zemke/b4m/`