Interval Arithmetic for Maxima: A Brief Summary

Richard J. Fateman
Computer Science Division
Electrical Engineering and Computer Sciences
University of California at Berkeley

May 4, 2016

Abstract

This is a short introduction to an implementation of interval arithmetic data structures and operations for the Maxima computer algebra system. We expect that further algorithms necessary to implement the forthcoming IEEE 1788 standard for interval arithmetic can be readily fit into this structure.

1 Introduction

Computer algebra systems (CAS) have a quite flexible way of introducing new concepts into their mathematical systems: functional notation. In particular, it is tempting to just introduce an interval which mathematicians might write as [a,b] in Maxima by typing `interval(a,b)`. Using this most obvious representation of an interval and using the usual techniques for programming operations with it results in some uncomfortable consequences. In particular, one encounters the so-called dependency problem. We can fix this and then proceed to build one (or several different) models for intervals that are more convenient for users.

2 Why the obvious method is unsatisfactory

If `interval(1,2)` is an implementation for the interval consisting of some x such that $1 \leq x \leq 2$. then `interval(1,2) - interval(1,2)` is, by the standard simplification for `x - x`, reduced to 0. This is correct if the computation is of $s - s$ where each s is *the same* `interval(1,2)`. On the other hand, it is not zero but `interval(-1,1)` if the computation is of $s - t$ where t is some *other* y such that $1 \leq y \leq 2$. That is, t is independent of s.

These kinds of results affect other operations as well.

3 A solution

There are many solutions possible, see Fateman[3] for a more expansive discussion of alternative representations in an algebra system. Also, we may wish to support some of the numerical tools which can be found on internet for interval computations. However, our context is different. We are not restricted to endpoints consisting of machine-precision floating-point approximations. We are capable of manipulating symbolic expressions which are later committed to interval evaluation, e.g. $9\,x^2 + 24\,x + 9$ is inferior, as an interval expression, to $(4 + 3\,x)^2 - 7$. We would also like to be compatible with the forthcoming IEEE standard

1788 for interval arithmetic[1]. Our favorite solution to the dependence problem is to use Lisp structures for intervals. The Maxima system sees the intervals, at least "at first glance", as atomic symbols. Thus we can define a program to be used as `ri(1,2)` to constructs an "atomic" real interval representing x for $1 \leq x \leq 2$. Uttering `ri(1,2)` a second time constructs a *different* atomic object that happens to have the same values (1,2) stored internally. Maxima simplifies the difference of two identical structures (that is, stored in the same memory location) to zero, but not if two such intervals are independent. The independence problem is solved.

Arithmetic is defined in the usual prescribed manner first by using accessor functions to extract upper and lower limits from the operands, perform arithmetic (with rounding as appropriate) and build new interval objects. A method for printing must also be supplied, and we have decided that it would be better to have an *unambiguously different* display for intervals. We have programmed `<1..2>` to be the display format for the interval produced by (for example) typing `ri(1,2)`.

4 Endpoints

We are considering *real intervals*, and so the form `ri(`a`, `b`)` is acceptable only if a, b are Maxima (real) numbers, with $a < b$. Maxima real numbers include integers like 1, rationals like 1/3, floats like 3.14, bigfloats like 3.1415b0, decimal bigfloats (if the decimal package is loaded) like 3.1415L0. An argument can be made that an interval implementation in a CAS should allow symbolic expressions as endpoints, hence `ri(x*y, z+cos(w))`. While this would be consistent with the "symbolic" philosophy that anything that is ordinarily numeric can be symbolic in a CAS. Unfortunately, computing with such endpoints rapidly becomes quite clumsy. Arguably there are no compelling large-scale applications for such intervals, but this may be because this feature has not been available[2]. There is an important edge case to consider, when endpoints are not quite numbers. It is necessary to allow the representation of intervals open at infinity, e.g. x such that $0 \leq x < \infty$. These can easily be produced if one divides by an interval containing zero. Thus representing intervals naturally requires a way of denoting infinity. The Common Lisp standard does not require IEEE 754 floating-point representations, but most[3] implementations allow the creation and storage of instances of (for example), double-float-positive-infinity. That, and its negative, are what we use for these forms. By convention, such intervals do not *include* infinity.

Furthermore, the IEEE 1788 standard needs representations for "not-an-interval" (NaI) and "empty interval" supplied either by special interval objects, or symbols. For example, $[a, b]$ with $a > b$ is empty; to be specific, 1788 suggests $[\infty, -\infty]$. A NaI may involve floating-point NaNs (Not a Number) but the standard does not constrain the format.

There is an important issue that arises, since a package to do operations on intervals must do more that arithmetic on intervals *per se*. Among these: we can try to extract an endpoint from an "improper" interval `<-oo..oo>`. What are we to see? One possibility is to signal an exception. This seems harsh at first glance, but if we can test for improper intervals perhaps we can side-step this possibility. Even more extreme, note that extracting left/right endpoints from NaI cannot yield any regular number, so a careful program must either check for the NaI (again, extracting an endpoint results in an exception), or later check for the value (whatever is extracted) before the next step, which might be arithmetic or comparison.

If we return one of the IEEE 754 floating-point number "infinities", or for NaI, a NaN, we then must support further operations on them. Since intervals may not even involve floats, as for example z=[-1/2, 1/3], why should 1/z involve floats? The endpoints of z are rational, so should the endpoints of 1/z be rational infinities? Are we talking nonsense here?

Perhaps we should return symbols, which have been used for decades in Maxima, but whose support is spotty: "inf" or "infinity". (Note: Maxima, in its standard configuration will simplify "inf-inf" to zero.)

[1]http://grouper.ieee.org/groups/1788/

[2]Mathematica 10 allows for the *construction* of such intervals, but does not appear to perform any manipulations on them.

[3]Notable exception is the popular implementation CLISP.

Consequently, our current design is to return an indexed object such as `infinity[n]` with a running, automatically-incremented index. There are other choices we have examined and implemented including changing the system to allow 1/0 as acceptable data to return as representing infinity. In such a system with related other rational-like objects, we can also construct something akin to "not a number" by using 0/0. It would be trivial to make this change to the interval data design, but there are ramifications for existing programs.

In particular, this rational number style frees the representation of ∞ from the domain of IEEE floating-point but requires that we insert new checks in Maxima's simplifier code to treat such "rational infinities" appropriately. For example 1/0-1/0 returns 0/0 etc.

Computing with infinities *outside intervals* in any of several designs for Maxima is discussed in another paper, forthcoming.

5 Next

Beyond the current level of implementation we can incrementally build a whole host of library routines for operating on intervals, using the IEEE 1788 standard as a template when appropriate. Additionally, a computer algebra system can support quite different operations from those of a numerical library. As one example, programs striving to rearrange computations as "single use expressions" (SUE) [13], or other beneficial forms, or increasing precision and/or subdividing intervals for improved results.

A sample version of an interval trigonometric routine (sine) is provided, to exhibit an approach using Lisp. Writing in Lisp is not at all required though. Models of such routines from libraries written in other (typically C-like) languages can either be loaded into Lisp, or can be effectively transliterated into Lisp language programs. The latter case may be preferable if the library does not implement arbitrary-precision arithmetic.

In the example below, `f()` computes the negation of a proper interval, K[a,b] by returning -K= [-b,-a]

```
matchdeclare(anint,ri_p)$
/* In a matching rule or "tellsimp" the variable anint matches an object that passes
   the test ri_p */

tellsimp(f(anint),ri(-ri_hi(anint),-ri_lo(anint)));
/*  for example, f(ri(1,3)), returns ri(-3,-1) which prints as <-3..-1>
```

A more bulletproof version of this program would have to check for NaI, and (perhaps) worry about infinite endpoints.

6 How to get this into your system

The program features are defined in this file: `http://www.cs.berkeley.edu/~fateman/lisp/interval.lisp` which can be downloaded and loaded into any Maxima system. This defines functions `ri, ri_p, ri_lo or left, ri_hi or right, widen_ri, interval_within, interval_intersection ?bu, ?bd, inf, minf, ind, nai_p, ri_lo_minf_p, ri_hi_inf_p, doint.`

Expressions like `H: ri(3,4)-r(3,4)` are left unchanged, but the `doint` command "does interval simplification". Therefore `doint(H)` results in an `<-1..1>`. Sometimes the expression can simplified without `doint`, as for example, sin(interval). The rationale for `doint` is that we do not wish to burden the general simplifier with the task of incessantly checking for intervals[4].

[4]An additional rationale is that we do not wish to burden the author/programmer with the task of re-programming so much of the Maxima general simplifier.

`ri(a,b)`, where `a` and `b` are numeric creates a real interval. Note, `ri(1,2)+x` does *not* produce `<1+x,2+x>` but is left alone. As mentioned earlier, the utility of symbolic endpoints for intervals is doubtful. `widen_ri(a,b)` produces an interval wider than `ri(a,b)` if it is given floating-point numbers, by rounding outward. Exact numbers are not widen. A single argument `widen_ri(a)` bumps the single float down and up.

?bu and ?bd for bump up and bump provide a facility that is general but slightly cruder than we would like. It would be technically preferable for machine floats – that is, it would produce tighter bounds, if we could wrap operations with floating-point rounding modes as `block([?round_mode:up], a+b)` but this is not supportable using only Common Lisp standard operations. Consequently we perform the operations and then bump the result. This loss in tightness (at the cost of speed) can be remedied to some extent by using software bigfloats of higher precision for the endpoints.

The Lisp tradition is to indicate programs that return true or false, (predicates) with a name ending in `-p`. This use of "-" is inconvenient, so we use `_p` as in the predicate testing to see if an object is a real interval, `ri_p`.

The other operations are obvious from their names.

7 Further information

The bibliography provides links to many more detailed commentaries on interval arithmetic, as does the IEEE 1788 standard. This brief paper indicates a bare minimum of information on the program code available in Maxima.

References

[1] Berz, M., COSY INFINITY system, http://bt.pa.msu.edu/index_cosy.htm

[2] Ioannis Emiris, Richard Fateman, "Towards an Efficient Implementation of Interval Arithmetic," Technical Report UCB/CSD-92-93, July, 1992 (UC Berkeley) `http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/6247.html`

[3] Richared Fateman, "To Infinity and Beyond", `http://www.cs.berkeley.edu/~fateman/papers/infinity.pdf`

[4] Eldon R. Hansen, "Sharpness in Interval Computations", *Reliable Computing* 3: 17-29 (1997).

[5] IEEE, `http://standards.ieee.org/findstds/standard/1788-2015.html`

[6] Ralph Martin, Huahao Shou, Irina Voiculescu, Adrian Bowyer and Guojin Wang, "Comparison of interval methods for plotting algebraic curves," Computer Aided Geometric Design, Volume 19, Issue 7, July 2002, Pages 553-587.

[7] Richard Fateman. file directory `http://www.cs.berkeley.edu/~fateman/generic`

[8] W. Kahan, "How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?" `http://http.cs.berkeley.edu/~wkahan/Mindless.pdf`

[9] Hongkun Liang and Mark A. Stadtherr, "Computation and Application of Taylor Polynomials with Interval Remainder Bounds," `http://citeseer.ist.psu.edu/cache/papers/cs/8022/`

[10] http://www.nd.edu/ markst/la2000/slides272f.pdf

[11] MPFI group (Revol, Rouillier) INRIA http://www.cs.utep.edu/interval-comp/interval.02/revo.pdf

[12] Nedialko S. Nedialkov, Vladik Kreinovich, Scott A. Starks, “Interval Arithmetic, Affine Arithmetic, Taylor Series Methods: Why, What Next?” (2003)
`http://citeseer.ist.psu.edu/nedialkov03interval.html`

[13] G. W. Walster, “Sun Forte Fortran,” http://developers.sun.com/prodtech/cc/products/archive/whitepapers/tech-interval-final.pdf