Validated Constraint Compilation

Timothy J. Hickey   David K. Wittenberg
Technical Report CS-99-201
Computer Science Department
Brandeis University
$\{tim|dkw\}$@cs.brandeis.edu

April 19, 1999

Abstract

We propose a method of using validated interval constraint contraction operators to build routines for numerical computation libraries. We illustrate the method by using it to construct a procedure that efficiently computes a small interval containing $f^{-1}(h)$ where $f(x) = x\sin(x)$. This can be transformed into a numerical routine by returning the midpoint of the interval and signalling an exception if the relative width of the interval exceeds a specified bound. We compare the interval method with a strictly numerical approach. The chief advantage of our method is that it results in a procedure which returns an interval that is guaranteed to contain the correct solution (assuming of course that the hardware and the compiler meet the necessary specifications). By automating those parts of the computation which could effect the correctness of the procedure, we reduce the number of places where the programmer can err. The user is free to design complex algorithms for solving the problem at hand, with the restriction that the algorithm consists of applying a sequence of validated constraint contractors which are constructed automatically from the constraint set specifying the problem. Modulo assumptions about the correctness of the contractors, the only error that the user can make is to produce a procedure which returns intervals that are too large too be useful. This is in contrast to traditional methods which always return answers with 52 bits of precision but with little or no indication of how many of those bits are correct.

1 Introduction

Inaccurate scientific computation is useless at best and dangerous at worst. We address several major sources of inaccuracy. Roundoff error is well known and there is a great deal of work on minimizing it [Wil63, Act96, Tay97]. By using interval constraints, we don't eliminate roundoff error, but we make it explicit, so each answer comes with a clear indication of its accuracy. Another source of error arises from misapplying an algorithm (e.g. starting the Newton method with a poor initial choice, or using the Newton method in a case where it does not perform well). Programming errors are a further major source of incorrect results. Hatton *et al.* [HR94] measured a cumulative, numeric error of 1% per 4000 lines in very carefully written, high quality code for certain scientific computations. How much worse it is in normal cases is unclear.

We propose a method for reducing the chance of programming errors in scientific programming by casting the problem as the design of an appropriate constraint solving algorithm and then separating the algorithm design process into two steps.

- First, the automatic construction of a set of constraint contractors which can be applied to the initial intervals in any order without affecting the correctness of the algorithm, and
- Second, the specification of the algorithm which determines which contractors are applied in which order. The specification of which operators are applied in what order determines the speed of convergence of the algorithm, but has no effect on correctness, which depends only on the correctness of the constraint contractors.

In this paper, we begin with a discussion of the avoidable and unavoidable sources of errors in scientific computing and then discuss several methods that have been used to minimize these errors. We then provide a comparison of the classical method with our proposed method for creating a procedure to invert $f(x) = x\sin(x)$ on a subset of its domain. We conclude with a discussion of future research directions suggested by this approach.

1.1 Errors and blunders in scientific computing

In the seminal paper [vNG47] "Numerical Inverting of Matrices of High Order," von Neumann and Goldstine describe four sources of errors in scientific computing (shown below). These errors are inherent in physical science computations, but we provide tools to lessen the damage caused by all but the first.

1. **Theory Errors** – *errors in the underlying scientific theory.*
2. **Measurement Errors** – *errors in the observational data.* Because we do all our calculations on intervals, it is easy to enter a measured value as an interval (or as a value with an error estimate.) We then carry the calculations through allowing the measurement to take on any value in the interval. This means that if a calculation is heavily dependent on the precision of an imprecise measurement, the answer given will show the possible large range of answers.
3. **Truncation Errors** – *errors due to approximation of solutions by algebraic expressions.* The use of intervals allows us to use classical series approximations, while still maintaining complete accuracy. We express the error term at the end of the series as an interval, and carry that through our calculations. If we were to truncate a series too soon, it would lead to an overly wide interval in the answer.
4. **Roundoff Errors** – *errors due to the fixed precision of computer arithmetic.* IEEE-754[IEE85] provides control over rounding. By using the ability to specify whether a value will be rounded up or rounded down, we insure that our calculated intervals are a superset of the actual intervals.

The last two classes of errors are due to the continuous nature of the problems we are trying to solve and the essentially finite nature of digital computation.

There are obviously some important sources of error that are not on this list, these are the "avoidable" errors or blunders, that arise from the mistakes made by individuals involved in setting up the computation. Some of the most common of these mistakes are:

1. **Analysis Blunders** – this occurs when the analyst makes a mistake in estimating the errors introduced by his computational solution.
2. **Algorithmic Blunders** – these are the errors that arise when the idealized algorithm proposed by the analyst does not correctly solve the mathematical problem.
3. **Programming Blunders** – these are the errors that arise when the algorithm is incorrectly implemented in a particular programming language.
4. **Software-use Blunders** – this covers the case of users who apply the software in ways for which it was not intended, e.g., applying it with variables that are outside of the range for which it was designed.
5. **Hardware Blunders** –these are the errors that arise when the hardware does not implement the arithmetic operations according to the published specifications (currently IEEE-754[IEE85] is most widely used). For the remainder of this paper, we will assume that the underlying hardware correctly implements IEEE-754, but this is clearly an area where formal verification is sorely needed.

In the everyday world of scientific computing, most of the practitioners are working scientists and not numerical analysts. This significantly raises the likelihood of the computed results being subject to one or more of these computational blunders.

Knuth [Knu92] provides a complete list of bugs he fixed in a ten year period while working on TEX. He divided his errors into fifteen categories. We are most interested in his categories A (Algorithm Awry), B (Blunder), and S (Surprising Scenario). We also hope to lessen his categories L (Language Liability), M (Mismatch between Modules), and R (Robustness). Despite counting requested improvements as errors, the blunders were still more than 5% of the total number of errors in TEX.

2 Interval Arithmetic

Interval Arithmetic [AK93, AH83, Han92, Moo66, Neu90] is one well-studied approach to handling some of these blunders. The idea behind this method is to free the analyst from the difficulties of estimating the effects of roundoff, measurement, and truncation errors by automatically computing an estimate of the error for each variable in the model. This is done by representing each variable in the mathematical model by a floating point interval (or a set of intervals) and defining all arithmetic operations and mathematical functions on these intervals in such a way that the actual result of any primitive operation or function is guaranteed to be contained in the computed result.

Just because the error is automatically computed does not mean that the vast literature of classical numerical analysis is not useful. Interval arithmetic algorithms are still subject to many of the same kinds of algorithmic errors that arise with floating point computation, but rather than result in incorrect values, these blunders typically result in wide intervals. For example, evaluating $1 - \cos^2(x)$ in floating point arithmetic for x small, will result in a loss of about $2b$ bits of precision if x is approximately 2^{-b}. If the same expression is evaluated using interval arithmetic, one gets an interval containing about 2^{2b} floating point numbers. Thus, interval arithmetic introduces a new kind of problem as well as making an old one slightly worse:

- *Precision Problem* – this covers the case where the program is not able to sufficiently narrow the result intervals.
- *Performance Problem* – this covers the case where the calculation requires an unreasonably large amount of time.

When either of these problems is detected, the insights of classical numerical analysis can then be used to suggest ways of restructuring the computation to increase the interval convergence rate of the algorithm.

Interval Arithmetic techniques can be used by expert numerical analysts to produce software which adheres rigorously to the mathematical model and computes result intervals which are guaranteed to contain the theoretical result values (barring any blunders by the analyst). Interval arithmetic thus has brought numerical analysis fully into the realm of pure mathematics for the first time, since one no longer needs to rely on heuristic arguments about the statistical unimportance of the roundoff and truncation errors.

The price for this mathematical rigor is that interval arithmetic routines often take longer to provide an answer and always overestimate the error. Empirically it is noted that these overestimates are sometimes so large as to render the classical algorithms useless (particularly iterative algorithms with many iterations). There has been considerable effort in the last thirty years at developing interval arithmetic algorithms which are not only mathematically correct, but are efficient and precise as well [AK93, Boh93]. As promising as this recent work in interval analysis has been, it does not address most of the common blunders in scientific computing and still requires the working scientist to use tools for solving very specific types of numerical problems or to write a solver in some standard language but using interval variables rather than doubles or floats at certain places. This latter option does little to remove many of the opportunities for blunders, and in some sense may aggravate this situation since it requires scientists to be aware of the difference between interval variables and numerical variables and when to use which.

2.1 Interval Arithmetic Constraints

The key idea behind interval arithmetic constraints [BO97, Cle87, Hic94, HvEW98, Hyv89, Ju98, OV93, vE97a, vE97b] is to view numeric computing problems as constraint systems that relate a set of real (or complex) variables or functions. The variables whose values are to be computed are initially unbounded in this model (i.e., they have the value $[-\infty, \infty]$). The goal of the computation is to shrink the intervals of the variables in such a way that no solution to the original system is removed. The shrinking is done by iteratively applying various contraction operators which are automatically generated from the constraint set.

There are several interval arithmetic constraint solvers which are currently available. In all of these the user specifies only the constraints to be solved and the system determines which contractors to apply and in which order.

For example, in IAsolver [HQvE99], the constraints are compiled into a set of primitive constraints (much the same as 3-address code is generated from expressions by a compiler), and contractors for these primitive constraints are repeatedly called until a fixed point is reached or some resource bound is exceeded. Note that since IEEE floating point has a successor function, there are only a finite number of contractions which can take place before a fixed point is reached.

The Numerica system [HMD97] uses interval arithmetic techniques to solve non-linear optimization problems with a relatively small number of constraints and variables. This system uses three types of contractors, one of which is the multi-variable interval Newton contraction, and also relies on domain-splitting to search for solutions.

These two systems share the property that the choice of contractors and the strategy in which they are selected is hidden from the user. Thus, if the user correctly enters the mathematical formulas describing the problem to be solved, then the system will produce intervals which are guaranteed to contain the correct solution.

There are also several Constraint Logic Programming languages which provide general non-linear interval arithmetic constraint solving. For example, in CLP(RI) [Sys96], a constraint is specified by a standard mathematical expression enclosed in curly braces. In CLP(F),[Hic94], the constraint language is extended to allow for differentiable functions and ODE constraints.

This scheme eliminates all possible blunders except for misuse of the software. It does still allow precision problems (i.e. the resulting intervals being unnecessarily large) and performance problems. The price of such protection is that the user plays no role in formulating the algorithm to solve the problem, and if the system is not able to sufficiently narrow the intervals quickly enough the user has few if any alternatives.

2.2 Validated Constraint Compilation

In this paper we propose a model of scientific computing which splits the problem solving process into two pieces:

1. Development of validated contraction operators which have been proved to shrink the variables in a set of constraints without removing any solutions to the constraints and which operate as efficiently as possible.

2. Development of constraint solving strategies for selecting the contraction operators to apply and the order in which to apply them to accelerate the convergence of the system to a solution

The contraction operators are typically specified declaratively in terms of the original mathematical constraint.

To demonstrate the feasibility of this approach, we consider the following problem which is often encountered by numerical analysts (although they generally describe the problem using different language):

Definition 1 *Assume we are given a set $\mathcal{C}$ of constraints on a set $\mathcal{V}$ of variables. Further assume that there exists disjoint sets of variables $\{X_i\}$ and $\{Y_i\}$ in $\mathcal{V}$ such that for any values a_i within given domains D_i, the set of constraints*

$$\mathcal{C} \cup \bigcup_i (X_i = a_i)$$

has a unique solution $(Y_1, \ldots, Y_m) = (b_1, \ldots, b_m)$. The **Validated Constraint Compilation Problem** *is to generate an efficient procedure which can compute approximations b'_i of the b_i given the a_i, and which produces an explicit guarantee of precision ϵ_i, for the relative error $|b'_i/b_i - 1|$ and absolute error $|b'_i - b_i|$.*

Rather than return both an approximate answer and an error bound, one could also just return the approximation and signal a runtime exception if the error exceeds some predetermined value.

The general method we propose to solve this problem is to first build a toolkit which can automatically construct validated contractors from symbolic constraints. The second step is to build tools for combining these contractors and for analyzing their performance.

In the remainder of this paper, we describe our experience applying this method to a simple but illustrative example from Acton [Act96] in which he describes the standard numerical analysis approach to this problem. We then contrast his approach with the constraint contraction approach.

3 The Hyperbola-Sine Intersection Example

Consider the following simple numerical problem ([Act96], pp. 77-80, pp. 223-224): Given a positive value of h, find all x and y such that

$$x * y = h, \quad y = \sin(x) \tag{1}$$

This simple constraint system describes the intersection of a (possibly degenerate) hyperbola and the sine curve. Clearly y can be computed from x, so we can simplify the problem to that of solving $h = f(x)$, where

$$f(x) = x \sin(x)$$

for x given h. As can be seen from Figure 1, for any h there are infinitely many solutions. Indeed, the function f has a local minimum at 0 and a local maximum at a point a_1 around 2 and hence has a uniquely defined inverse on the interval $[0, a_1]$. If we precompute a_1, we find that it has the value $a_1 \approx 2.028757838110434223$ and $b_1 = a_1 \sin(a_1) \approx 1.819705741159653046$ Thus, for $h \in [0, b_1]$, this constraint

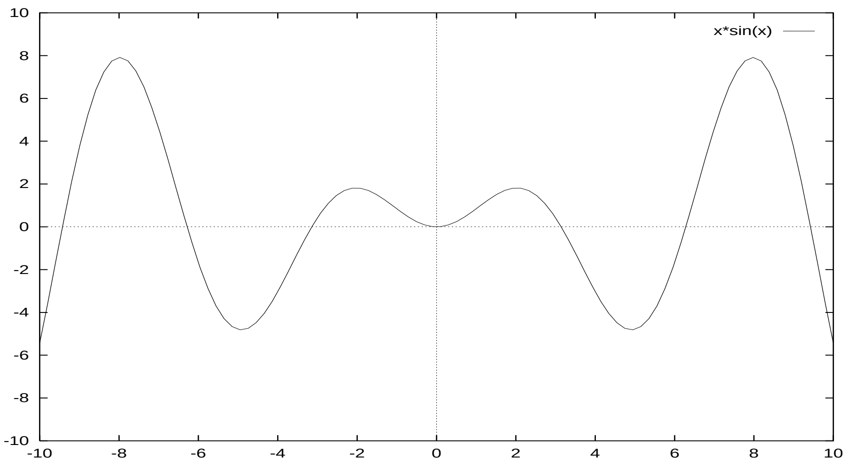

Figure 1: The Graph of $f(x) = x\sin(x)$

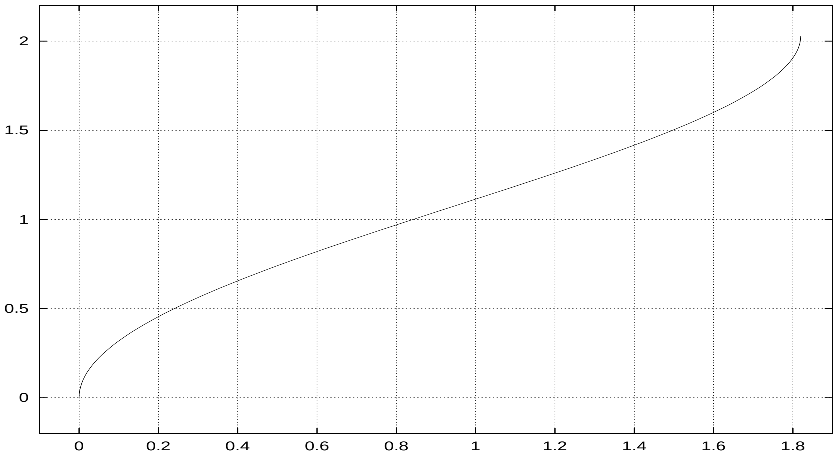

Figure 2: The Inverse Function to be computed.

system has a unique solution with $x \in [0, a_1]$. Similarly, if we let (a_i, b_i) be the sequence of local maxima and minima starting with $a_0 = b_0 = 0$, and (a_1, b_1), as above, then f can be inverted on each of the intervals $[a_i, a_{i+1}]$. We consider only the interval $[a_0, a_1]$, but the methods we describe apply equally well to all of the other cases. The inverse function we will compute is shown in Figure 2.

We first define the *relative error* of a calculated approximation x' to a correct value x to be $|x'/x - 1|$. Note that this is not defined where $x = 0$, and may be misleading for x close to 0. In this paper we can safely ignore this problem.

To simplify the discussion, we consider now only the problem of writing a program to compute this inverse function quickly and precisely. In other words, we are solving the validated constraint contraction problem for the following constraint:

$$h = x\sin(x), \quad 0 \le x \le a_1,$$

where h is the input variable and x is the output variable.

This example is typical of many of the kinds of problems that arise in scientific computing, but as we will see below, employing standard techniques to write a procedure which is fast and efficient is fairly subtle. The additional requirement that it should always return an answer and a guaranteed relative error

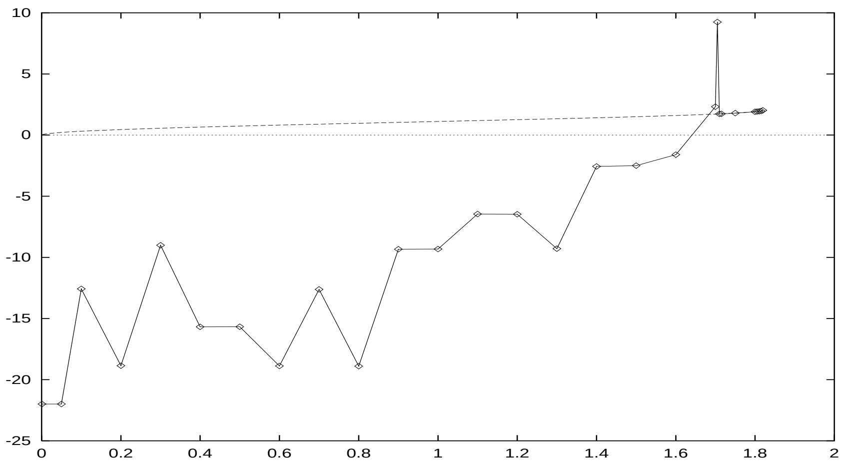

Figure 3: Illustration of error from poor choice of starting point in the Newton method.

bound is generally not even attempted because of the difficulty it presents for classical methods.

3.1 The Newton Method

The method of first choice in numerical analysis is the Newton method. The general method uses the following iteration

$$x_{i+1} = x_i + \frac{h - f(x_i)}{f'(x_i)}$$

to find a solution to the constraint $f(x) = h$. When it is started with a good initial point, it often converges rapidly to a solution (usually with a doubling of the precision achieved in each step!). When it doesn't work, it can fail dramatically. It may converge to the "wrong" solution, it may fail to converge, or it may simply converge too slowly to be practical.

For the problem at hand the iteration is:

$$x_{i+1} = x_i + \frac{h - x_i * \sin(x_i)}{\sin(x_i) + x_i * \cos(x_i)} \tag{2}$$

If we solve the constraint $x * \sin(x) = h$ using the initial guess of $x_0 = h$, then this generally converges rapidly to a solution. An injudicious choice of starting point for the Newton method can yield disastrous results in this case. For example, if we were to start the Newton iteration at $x_0 = 2$, then for all values of x except those very near 2, the first Newton iteration would jump out of current domain and converge (rapidly) toward some other solution of the constraint than the one we intended. Figure 3 shows the function computed by this badly initialized Newton method (the solid line) compared to the actual desired function (the dashed line).

Figures 4, 5, and 6 show the precision estimates for three different methods over three ranges. The methods are the Newton method using Eqn. 2 (solid), the pseudoconstant method using Eqn. 3 (dotted), and the quadratic method using Eqn. 4 (dashed), each iterated six times. The last two methods will be described in the next section. We observe in these figures that after only six iterations, Newton's method has achieved near optimal precision (around 52 bits) for all x between about 0.1 and 1.8. However, we also observe that near the two endpoints ($b_0 = 0.0$ and $b_1 \approx 1.8197$), the Newton method performs quite poorly. These "bits of precision" graphs were created by using multi-precision arithmetic to obtain a hopefully more accurate result at 20 particular points in the range and then using the difference between the multi-precision result and the computed double precision result to estimate precision. Using multi-precision arithmetic helps add to the confidence in our results, but it doesn't constitute a proof of correctness.

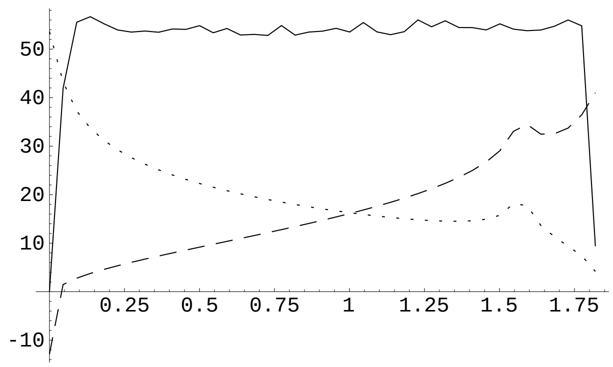

Figure 4: Bits of Precision of the three numerical iterations

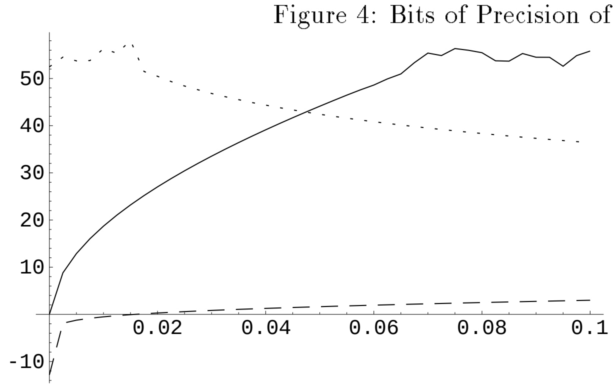

Figure 5: Bits of Precision near $b_0 = 0$

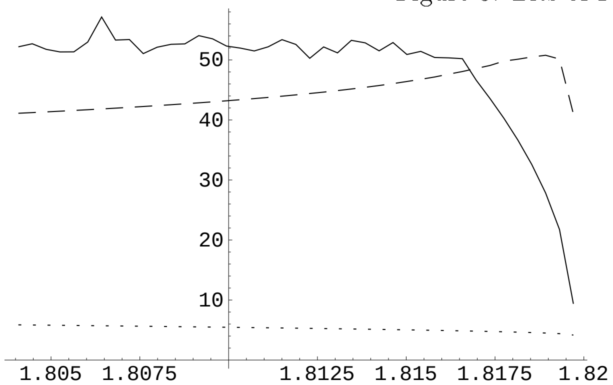

Figure 6: Bits of Precision near b_1

The slow convergence near $b_0 = 0$ and near $b_1 = 1.8197\ldots$ is due to the fact that the function $f(x)$ is approximately quadratic near those points, and so the slope is very close to zero and this impedes the convergence of Newton. Equivalently, as can be seen in Figure 2, the inverse function we are trying to compute has slope which approaches infinity at the two endpoints, and these near infinite slopes slow Newton to a crawl.

3.2 The Pseudoconstant Quadratic iterations.

Acton suggests two different approaches near these dangerous points. Near zero, he uses what he calls the pseudoconstant gambit, in which the original constraint $x * \sin(x) = h$ is rewritten as $x^2 * (\sin(x)/x) = h$, and with the observation that $\sin(x)/x$ is nearly constant near zero, he introduces the following iteration:

$$x_{i+1} = \sqrt{h/\left(\sin(x_i)/x_i\right)} \tag{3}$$

This converges rapidly for h near zero, but is slow for other h.

Acton shows how to improve the convergence near the right endpoint $b_1 \approx 1.8197$ by introducing new variables δ and γ with $x = a_1 + \delta$ and $h = b_1 - \gamma$ which translate the problem to the origin and put it in the form $\beta = \delta^2 * g(a_1, \delta)$ where $g(a_1, \delta)$, like $\sin(\delta)/\delta$ is nearly constant for δ near zero.

$$b_1 - \gamma = h = x \sin x = (a_1 + \delta) \sin(a_1 + \delta)$$

We now use the fact that the point (b_1, a_1) is a local maximum for f which implies that

$$\begin{aligned} a_1 \sin(a_1) &= b_1 \\ \sin(a_1) + a_1 \cos(a_1) &= 0 \end{aligned}$$

These relations, and the definitions $\{h = b_1 - \gamma,\ x = a_1 + \delta\}$, can be used to transform the equation $h = x \sin(x)$ into a rapidly converging iteration as follows:

$$\begin{aligned}
h &= x \sin(x) \\
b_1 - \gamma &= (a_1 + \delta) \sin(a_1 + \delta) \\
\gamma &= b_1 - ((a_1 + \delta) \sin(a_1 + \delta)) \\
\gamma &= a_1 \sin(a_1) - ((a_1 + \delta)(\cos(\delta) \sin(a_1) + \sin(\delta) \cos(a_1))) \\
\gamma &= a_1 \sin(a_1) - ((a_1 + \delta)(\cos(\delta) \sin(a_1) - \sin(\delta) \sin(a_1)/a_1)) \\
\gamma &= a_1 \sin(a_1) - ((a_1 + \delta) \sin(a_1)(\cos(\delta) - \sin(\delta)/a_1)) \\
\gamma/\sin(a_1) &= a_1 - (a_1 \cos(\delta) - \sin(\delta) + \delta \cos(\delta) - \delta \sin(\delta)/a_1)) \\
\gamma/\sin(a_1) &= a_1(1 - \cos(\delta)) + (\sin(\delta) - \delta \cos(\delta)) + \delta \sin(\delta)/a_1)
\end{aligned}$$

If we expand the occurrences of $\sin(\delta)$ and $\cos(\delta)$ using power series in the last equation, and simplify, we find that it has the form: $\gamma/\sin(a_1) = \delta^2 g(a_1, \delta)$, where

$$g(a_1, \delta) \quad = \quad \frac{1}{2}(a_1 + 2/a_1) + \frac{\delta}{3!}(2) + \frac{\delta^2}{4!}(-(a_1 + 4/a_1)) + \frac{\delta^3}{5!}(-4) + \ldots$$

is a function which is nearly constant for δ near zero. Thus, proceeding as in the pseudoconstant gambit we find

$$\begin{aligned} \gamma/\sin(a_1) &= \delta^2 g(a_1, \delta) \\ \delta^2 &= (\gamma/\sin(a_1))/g(a_1, \delta) \end{aligned}$$

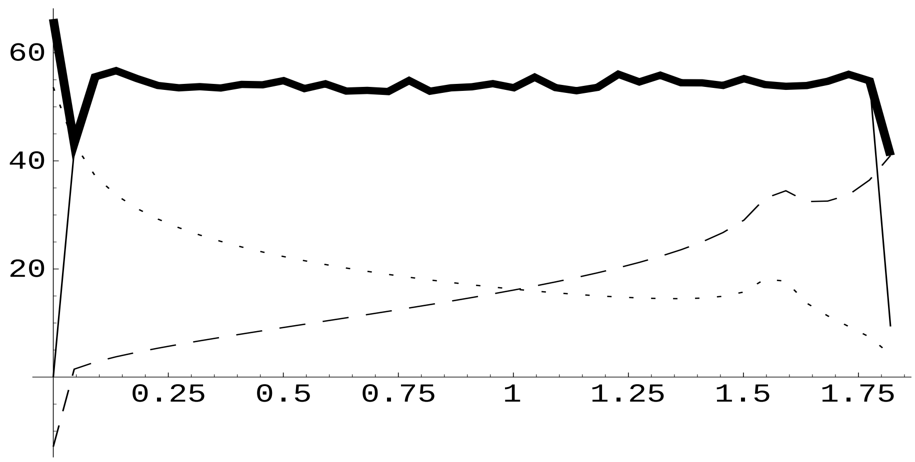

Figure 7: Bits of Precision of the combined method

$$\begin{aligned} \delta &= -\sqrt{(\gamma/\sin(a_1))/g(a_1,\delta)} \\ x - a_1 &= -\sqrt{(b_1 - h)/\sin(a_1))/g(a_1, x - a_1)} \end{aligned}$$

Hence, we get the following iteration in x:

$$x_{i+1} \leftarrow a_1 - \sqrt[2]{\frac{(b_1 - h)/\sin(a_1)}{g(a_1, x_i - a_1)}} \tag{4}$$

Although this required more symbolic manipulation than the case when x is near zero, the underlying idea for speeding up convergence is the same – express it as $\delta^2 * c(\delta) = d$ where $c(\delta)$ is near constant when δ is near zero, and then iterate on δ using $\delta = \pm\sqrt{d/c(\delta)}$, where the sign depends on which root you are seeking.

3.3 Combining the iterations in a numerical procedure

The end result of this standard numerical approach is to write a procedure which uses some number of iterations of the pseudoconstant iteration (Eqn. 3) for h near zero, some number of iterations of (Eqn. 4) for h near the upper bound b_1, and some number of iterations of the Newton method (Eqn. 2) for the remaining cases. For example, Figure 7 shows a plot of the estimated number of bits of precision for the function, which uses the Newton method for all h in the range $[0.05, 1.8]$ and uses the better of the two quadratic methods near the end points. The thick line shows the estimated precision, while the solid, dashed, and dotted lines are the Newton method, the Pseudoconstant method, and the Quadratic method respectively.

This method of constructing and analyzing numerical routines is highly heuristic and does not provide any guarantee that the stated precision bounds will be achieved. Either an interval arithmetic approach or a detailed line-by-line numerical analysis will be required if one wants a "provably correct" estimate of the error.

4 The Constraint Contractor Method

In this section we present the constraint contraction method. The basic idea is to define a general family of validated contractors and to experiment with various combinations of these contractors. When a reasonably good sequence of contractions is found, one can then generate a procedure to implement these contractors and to signal an exception if the width of the computed intervals exceeds the required error bounds. This approach has three key benefits:

- First, the contractors can be generated automatically from the constraints using various parameters specified by the user (thereby eliminating a potential source of programming errors).
- Second, if the particular sequence of contractors does not sufficiently narrow the result intervals, then this can be detected at runtime and an exception can be thrown.
- Third, the contractors can be applied in any order without affecting the correctness of the algorithm. The only possible danger is that for some inputs, the particular sequence of contractors does not narrow the result interval sufficiently, which as noted above can be detected and handled at runtime.

We first present a method that can be used with general constraint solvers such as IAsolver, CLP(RI), or CLP(F). Then we describe a more direct approach which promises to be more efficient.

4.1 The Taylor Constraint

The starting place for all of our validated contractors is the Taylor formula with remainder.

Theorem 1 (The Taylor Formula with Remainder) *Assume that f is n-times continuously differentiable on an interval D, then for any a in D there is a $\xi_{a,x}$ between a and x such that*

$$f(x) = \left(\sum_{i=0}^{n-1} c_i(a)(x-a)^i \right) + c_n(\xi_{a,x})(x-a)^n \tag{5}$$

where for any t in D, $c_i(t) = \frac{f^{(i)}(t)}{i!}$ is the normalized ith derivative of f at t.

An immediate consequence of the Taylor formula is that every solution (x, y) to the constraint $y = f(x)$ can be extended to a solution (x, y, t) of the **Taylor constraint** below. This is really a family of constraints, parameterized by a variable a at which the Taylor formula is centered. The Taylor constraint can be used to get the quadratic Taylor convergence in a general purpose constraint solver, by repeatedly adding a Taylor constraint with a chosen as the current midpoint of the variable X, contracting to a fixed point (or a fixed number of primitive contractions), and then adding another (redundant) Taylor constraint with a set to the new midpoint of X. The Taylor constraint also introduces an extra variable T, which is used to represent a value $z = \xi_{a,x}$ which lies between a and x. Any such value can be represented as $z = a + t(x - a)$ for some $t \in [0, 1]$.

Corollary 1 (The Taylor Constraint) *Assume that f is n-times continuously differentiable on an interval D, and let (x, y) be a solution to the constraint $y = f(x)$. Then, for any a in D there is a $t \in [0, 1]$ such that $(X, Y, T) = (x, y, t)$ is a solution to the following constraint:*

$$Y = \left(\sum_{i=0}^{n-1} c_i(a)(X-a)^i \right) + c_n(a + T * (X - a)), \quad 0 \leq T \leq 1 \tag{6}$$

We call this constraint the degree n Taylor constraint for f centered at a. Observe that this constraint holds for any a in the domain D of f.

For example, if $f(x) = x \sin(x)$, then f is infinitely differentiable on the reals, and we can easily compute the derivatives of f to obtain the following Taylor formula with remainder:

$$f(x) = \left(\sum_{i=0}^{n-1} c_i(a)(x-a)^i \right) + c_n(\xi_{a,x})(x-a)^n \tag{7}$$

where

$$c_j(t) = \begin{cases} -j\cos(t) + t\sin(t) & \text{if } j \equiv 0 \pmod 4 \\ j\sin(t) + t\cos(t) & \text{if } j \equiv 1 \pmod 4 \\ j\cos(t) - t\sin(t) & \text{if } j \equiv 2 \pmod 4 \\ -j\sin(t) - t\cos(t) & \text{if } j \equiv 3 \pmod 4 \end{cases} \tag{8}$$

Thus, the degree 1 Taylor constraint for $x\sin(x)$ centered at the point a would be

$$Y = a\sin(a) + (\sin(Z) + Z\cos(Z))(X - a), \quad Z = a + T * (X - a), \quad 0 \le T \le 1$$

and the degree 2 Taylor constraint for $x\sin(x)$ centered at the point a would be

$$\begin{aligned} &Y = a\sin(a) \; + \; (\sin(a) + a\cos(a))(X - a) \; + \; \tfrac{(2\cos(Z) - Z\sin(Z))}{2!}(X - a)^2, \\ &Z = a + T * (X - a), \quad 0 \le T \le 1 \end{aligned}$$

Observe that these constraints *are not* valid if Z is replaced by X. Indeed, this would be tantamount to stating that $\xi_{a,x}$ must lie in X, when in fact all that we know is that it lies between a and some $x \in X$. From an operational point of view, we could well choose an initial $a \in X$, in which case we would have $Z = X$, but after enough contractions one will usually have $a \notin X$ in which case Z contains but is not equal to X. In fact, Z is the smallest interval containing both a and X.

4.2 The Newton Contractors

In this section, we describe a family of contractors derived from the Taylor constraint. In the context of our problem, which is to generate a procedure for solving one particular constraint problem, we do not actually need to employ the general constraint solving machinery. We show later how these contractors can be used to easily solve the problem of inverting $x\sin(x)$.

To simplify notation we introduce the following definition so that we can avoid the cumbersome interval expression $a + [0,1] * (X - a)$ which represents the value $\xi_{a,x}$ in the Taylor formula.

Definition 2 *For any number a and interval X, let $\alpha_a(X) = a + [0,1] * (X - a)$ be the smallest interval that contains a and X. Equivalently, it is the set of all elements between a and every $x \in X$. Equivalently, it is the set of all solutions to the constraint:*

$$Z = A + T * (X - A), 0 \le T \le 1$$

Definition 3 (The order k, degree n Newton contractor.) *Let f be a function which is n times continuously differentiable on an interval D and let $a \in D$ and $X \subset D$. Then the order k, degree n Newton contractor for $f(x) = y$ centered at a is denoted $N_{f,k,n,a}$ and defined to be the function that maps the interval pair (X, Y) to (X', Y), where*

$$\begin{aligned} X' &= X \cap \sqrt[k]{\frac{Y - p_k(a, X)}{q_{k,n}(a, X, \alpha_a(X))}} \\ p_k(a, X) &= \sum_{i=0}^{k-1} \frac{f^{(i)}(a)}{i!}(X - a)^i \\ q_{k,n}(a, X, Z) &= \left(\sum_{i=k}^{n-1} \frac{f^{(i)}(a)}{i!}(X - a)^{i-k} \right) + \frac{f^{(n)}(Z)}{n!}(X - a)^{n-k} \end{aligned}$$

There are some subtle points in the previous definition. First, the basic interval arithmetic operations must be implemented correctly (e.g. as in [HvEW98]). Moreover, the nth-root operator on intervals, must return a union of disjoint intervals if n is even and its argument is a positive interval. More precisely, we define

$$\sqrt[n]{([a,b])} = \begin{cases} [\sqrt[n]{a},\ \sqrt[n]{b}] & n \text{ odd} \\ [-\sqrt[n]{b}, -\sqrt[n]{a1}] \cup [\sqrt[n]{a1},\ \sqrt[n]{b}] & n \text{ even, } b >= 0 \\ \emptyset & n \text{ even, } b < 0 \end{cases}$$

where $a1 = \max(0, a)$.

Theorem 2 (Correctness of Newton contractors) *Suppose f is n-times continuously differentiable in a domain D and let $\gamma_1, \gamma_2, \ldots, \gamma_r$ be Newton contractors of the form*

$$\gamma_i = N_{f,k_i,n_i,a_i}$$

for some $a_i \in D$ and integers $n_i \in [0, n]$, $k_i \in [0, n_i - 1]$. Then, for any such choice of a_i, n_i, and k_i, the interval function

$$\gamma(X, Y) = \gamma_r(\ldots\gamma_2(\gamma_1(X, Y)))$$

is a valid contractor for the constraint $y = f(x)$ in the sense that if $(x, y) \in X \times Y$ is a solution of $y = f(x)$, then $(x, y) \in \gamma(X, Y)$

Proof: The correctness of γ depends on the correctness of the γ_i. So we need only prove correctness of each γ_i. Let (x, y) be a solution to $y = f(x)$, then there is a $\xi_{a,x}$ between a and x such that

$$y = \left(\sum_{i=0}^{n-1} c_i(a)(x-a)^i\right) + c_n(\xi_{a,x})(x-a)^n$$

where for any t in D, $c_i(t) = \frac{f^{(i)}(t)}{i!}$ is the normalized ith derivative of f at t. Thus,

$$\begin{aligned} y &= \left(\sum_{i=0}^{k-1} c_i(a)(x-a)^i\right) \\ &\quad +(x-a)^k\left(\sum_{i=k}^{n-1} c_i(a)(x-a)^{i-k} + c_n(\xi_{a,x})(x-a)^{n-k}\right) \\ &= p_k(a,x) + (x-a)^k q_k(a,x,\xi_{a,x}) \end{aligned}$$

Rearranging this equation, we find:

$$\begin{aligned} (y - p_k(a,x)) &= (x-a)^k q_k(a,x,\xi_{a,x}) \\ (x-a)^k &= \frac{(y - p_k(a,x))}{q_k(a,x,\xi_{a,x})} \end{aligned}$$

The theorem then follows from our definition of $\sqrt[k]{\ldots}$ and the assumed correctness of the interval arithmetic operators. **QED**

Observe that in the case $k = n = 1$ and $a \in X$, we have $\alpha_a(X) = X$ and so the degree 1 order 1 Taylor contraction gives the standard Newton-Hansen-Sengupta operator [HS81], and its correctness is a corollary of correctness of the general Newton contractor.

Corollary 2 (Correctness of the Newton-Hansen-Sengupta Operator) *If f is continuously differentiable in an interval D and $X \subset D$, then for any $a \in X$, the operator*

$$S_{f,a}(X, H) = X \cap \left(a + \frac{H - f(a)}{f'(X)}\right)$$

has the property that $S_{f,a}(X, Y) = (X', Y')$ where $X' \times Y'$ contains all solutions to $y = f(x)$ in $X \times Y$.

Our experiments have indicated that the most useful Newton contractors are those of order 1 and 2, which we call the degree n Newton contraction and degree n Quadratic contraction, respectively, and define as follows:

Definition 4 (The Linear Newton Contraction) *If f is n-times continuously differentiable in an interval D and that $X \subset D$, then for any $a \in D$, the degree n Newton contraction is*

$$\begin{aligned} N_{f,1,n,a}(X,H) &= X \cap a + \left(\frac{H - f(a)}{q(a,X)}\right) \\ q(a,X) &= \left(\sum_{i=1}^{n-1} \frac{f^{(i)}(a)}{i!}(X-a)^{i-1}\right) + \frac{f^{(n)}(\alpha_a(X))}{n!}(X-a)^{n-1} \end{aligned}$$

Definition 5 (The Quadratic Newton Contraction) *If f is n-times continuously differentiable in an interval D and that $X \subset D$, then for any $a \in D$, the degree n Quadratic contraction is*

$$\begin{aligned} N_{f,2,n,a}(X,H) &= X \cap \left(a + \sqrt{\frac{H - f(a) - f'(a)(X-a)}{q(a,X)}}\right) \\ q(a,X) &= \left(\sum_{i=2}^{n-1} \frac{f^{(i)}(a)}{i!}(X-a)^{i-2}\right) + \frac{f^{(n)}(\alpha_a(X))}{n!}(X-a)^{n-2} \end{aligned}$$

Observe that these two contractors are defined for any $a \in D$. Typically we choose a to be an endpoint or midpoint of the interval X. The linear Newton contractor is best applied when $f'(a)$ is far from zero, whereas the quadratic Newton contractor is best applied when $f'(a)$ is nearly zero, but $f''(a)$ is far from zero, i.e. near simple local maxima and minima. The interval functions $f'(Y)$ and $f''(Y)$ must be approximated soundly. This can be done either by evaluating a symbolic expression for $f'(x)$ or $f''(x)$ using interval arithmetic, or it can be done using the Taylor series with remainder, where the remainder term is evaluated using a symbolic representation of the nth derivative.

4.3 Estimating the precision of composed contractions

In our example of solving $h = x\sin(x)$ for x given h, we use three order k degree 19 Newton contractors.

$$\begin{aligned} \gamma_0 &= N_{f,2,19,0}(X,H) \\ \gamma_1 &= N_{f,1,19,mp(X)}(X,H) \\ \gamma_2 &= N_{f,2,19,a_1}(X,H) \end{aligned}$$

where $mp(X)$ denotes the midpoint of X and a_1 is the upper bound of the interval on which we are inverting $f(x) = x\sin(x)$. These are instances of the degree 19 Linear Newton and Quadratic Newton contractions.

In Figure 8 we show the upper and lower bounds after one iteration of each of the three contractors. The upper plot is γ_0, the middle γ_1, and the lower γ_2. The horizontal axis is h and the vertical axis shows the upper and lower bounds of the new interval obtained by applying the contractor to the initial interval $[0, a_1]$. Thus, we see that γ_0 contracts well near zero and poorly near b_1, while γ_2 exhibits the opposite behavior. Each plot also shows the upper and lower bounds after the contraction

$$\gamma_0 \circ \gamma_2 \circ \gamma_0 \circ \gamma_1 \circ \gamma_1 \circ \gamma_1$$

which appears as a line since the error is quite small.

The graph in Figure 9 shows the number of bits of precision for the six prefixes of the contractor

$$\gamma_0 \circ \gamma_2 \circ \gamma_0 \circ \gamma_1 \circ \gamma_1 \circ \gamma_1$$

For the full contractor, the number of bits of precision achieved is better than 50 over the range [0:1.7], but it drops to about 45 at 1.8 and then to a low of 29 near the right endpoint a_1. Notice that the correctness of these error bounds depends only on the correctness of the underlying interval arithmetic operators and the correctness of the constraint contractors. In particular, we don't need to use high precision arithmetic, nor do we need to reason about the correctness of the algorithm, since the algorithm in this case is just a matter of composing constraint contractors.

5 Future Directions

In this paper we have discussed one interesting example in detail and demonstrated that the interval constraint contraction approach provides a simpler and more rigorous method of generating numerical routines. Before such an approach can be widely accepted, it will need to be extended in a number of ways. In particular, one would need to be able to handle constraints sets involving a large number of variables and one would need a correspondingly wider repertoire of validated constraint contractors. The Newton contractors described here extend easily to the multivariable case by selecting one variable in a constraint and treating all other variables as constants. We are in the process of building a system which incorporates not only these Taylor contractors but also the Numerica contractors (including higher dimensional Newton) and domain splitting primitives.

A further important area of research concerns the partial evaluation and optimized compilation of interval arithmetic procedures. In our experience, current compilers routinely ignore rounding mode changes when optimizing code and thereby generate code in which the rounding modes are applied at the wrong times. This points out the necessity of developing special case compilers for the types of interval arithmetic procedures that a system such as ours would generate. There are also numerous low level optimizations (such as partial evaluation of the low level interval arithmetic routines for addition, multiplication, etc.) that can greatly improve the performance of interval arithmetic code. Our long term goal is the development of interval constraint-based tools for generating efficient and precise numerical routines with automatically generated rigorous error bounds.

References

[Act96] Forman S. Acton. *Real computing made real: Preventing Errors in Scientific and Engineering calculations.* Princeton University Press, Princeton, New Jersey, 1996.

[AH83] Götz Alefeld and Jürgen Herzberger. *Introduction to Interval Computations.* Academic Press, 1983.

[AK93] E. Adams and U. Kulisch. *Scientific Computing with Automatic Result Verification.* Academic Press, 1993.

[BO97] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming*, 32, 1997.

[Boh93] Gerd Bohlender. Bibliography on enclosure methods and related topics. In U. Kulisch E.Adams, editor, *in Scientific Computing with Automatic Result Verification.* Academic Press, 1993.

[Cle87] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2:125–149, 1987.

[Han92] Eldon Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, 1992.

[Hic94] Timothy J. Hickey. CLP(F) and constrained ODEs. In *Proceedings of the Workshop on Constraints and Modelling*, 1994.

[HMD97] Pascal Van Henternryck, Laurent Michel, and Yves Deville. *Numerica: A modeling Language for Global Optimization.* MIT Press, 1997.

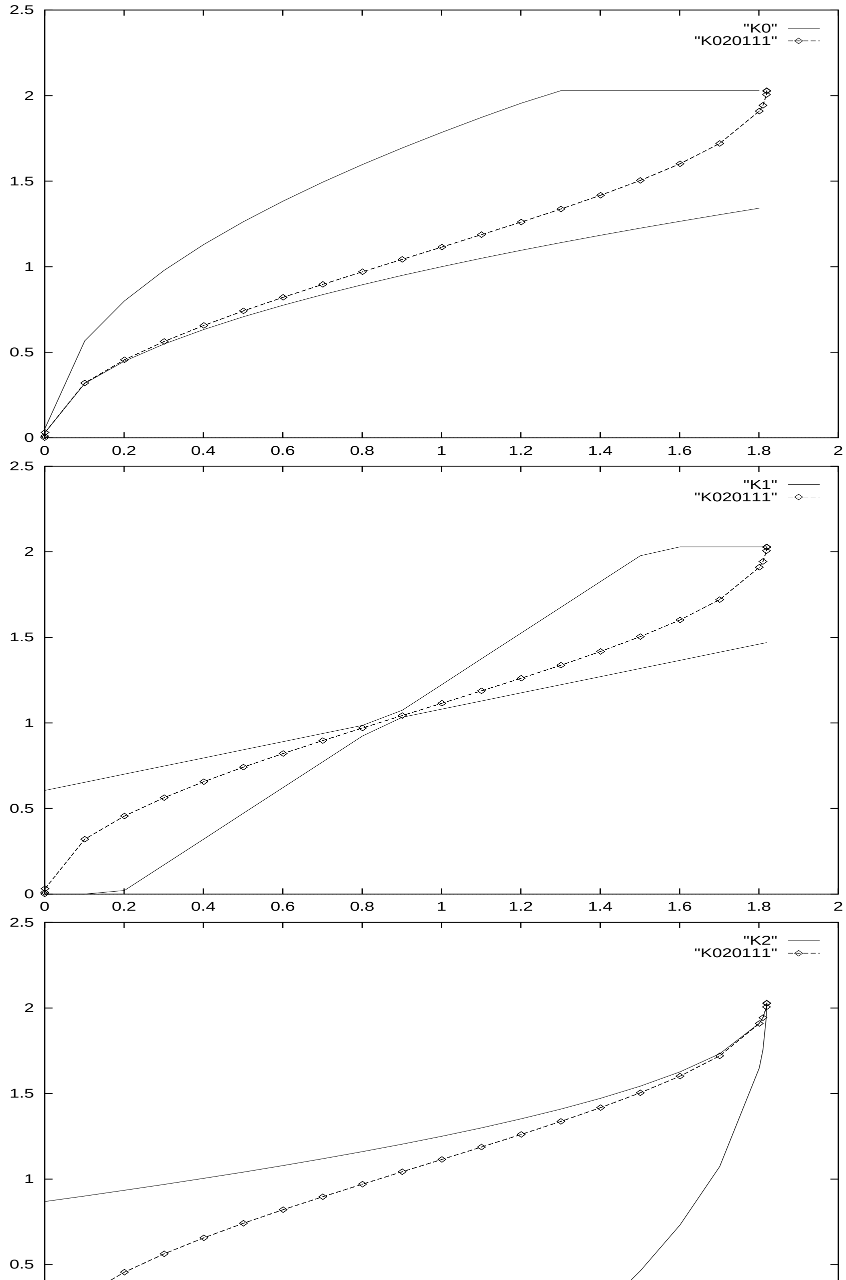

Figure 8: Convergence of the interval contractors

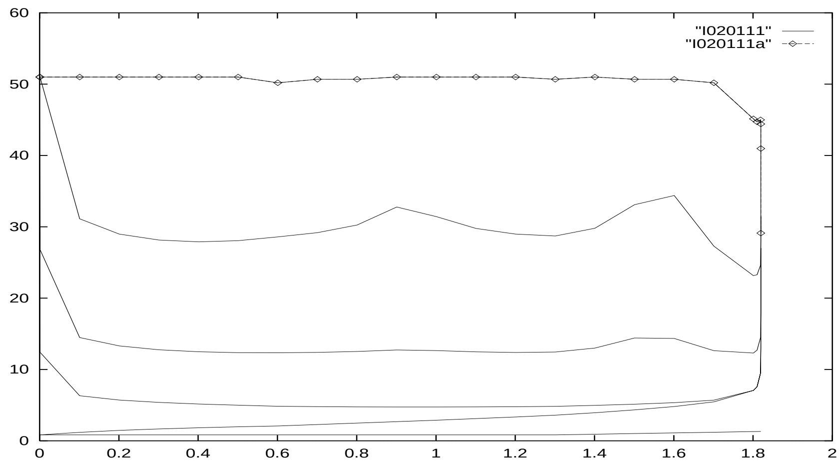

Figure 9: Bits of precision of the interval contractors

[HQvE99] T. Hickey, Z. Qiu, and M.H. van Emden. Interval constraint plotting for interactive visual exploration of implicitly defined relations. *accepted for publication in Reliable Computing*, 1999.

[HR94] Les Hatton and Andy Roberts. How accurate is scientific software? *IEEE Transactions on Software Engineering*, 20(10):785 – 797, October 1994.

[HS81] E.R. Hansen and S. Sengupta. Bounding solutions of systems of equations using interval analysis. *BIT*, 21:203–211, 1981.

[HvEW98] T.J. Hickey, M.H. van Emden, and H. Wu. A unified framework for interval constraints and interval arithmetic. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 250–264. Springer-Verlag, 1998.

[Hyv89] E. Hyvönen. Constraint reasoning based on interval arithmetic. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1193–1198, Detroit, USA, 1989.

[IEE85] IEEE. IEEE standard 754-1985 for binary floating-point arithmetic. *SIGPLAN*, 22(2):9–25, 1985.

[Ju98] Qun Ju. *A Sound Interval Constraint Logic Programming System.* PhD thesis, Brandeis University, CS Dept, May 1998.

[Knu92] Donald E. Knuth. *Literate Programming*, volume 27 of *CSLI Lecture Notes*, chapter 10, pages 243–291. Center for the study of language and information, 1992. Reprinted from Software—Practice and Experience 19 (1989) pg. 607–685.

[Moo66] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[Neu90] Arnold Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.

[OV93] W. Older and A. Vellino. Constraint arithmetic on real intervals. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research.* MIT Press, 1993.

[Sys96] Applied Logic Systems. *CLP(RI) reference manual.* Applied Logic Systems, 1996.

[Tay97] John R. Taylor. *An introduction to Error Analysis: The Study of Uncertainties in Physical Measurements.* University Science Books, second edition, 1997.

[vE97a] M.H. van Emden. Canonical extensions as common basis for interval constraints and interval arithmetic. In *Proceedings of the Sixth French Conference on Logic and Constraint Programming*, Orléans, France, 1997.

[vE97b] M.H. van Emden. Value constraints in the CLP Scheme. *Constraints*, 2:163–183, 1997.

[vNG47] J. von Neumann and H.H. Goldstine. Numerical inverting of matrices of high order. *Bull. Amer. Math. Soc.*, 53:1021–1099, 1947.

[Wil63] J. H. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice-Hall, 1963. republished by Dover books, 1994.