A Multiple-Precision Interval Arithmetic Package

David M. Smith

Loyola Marymount University

This article describes a collection of Fortran routines for evaluating basic arithmetic, elementary functions, and special functions having intervals for input and output.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General – *computer arithmetic, multiple precision arithmetic;* G.4 [**Mathematical Software**]: – *Algorithm design and analysis, efficiency, portability*

General Terms: Interval arithmetic, multiple precision, algorithms

Additional Key Words and Phrases: Accuracy, function evaluation, floating point, Fortran, mathematical library, portable software

---

1. INTRODUCTION

Interval arithmetic has been used since the 1950's as a way to have guaranteed accuracy for floating-point computations. An early influential book was R.E. Moore's *Interval Analysis* [1966].

This package of Fortran routines extends the FM package [Smith 1991], to perform multiple-precision interval arithmetic.

In place of the usual floating-point value x for approximating the "true" real value of a result, we want to have an interval $[x_l, x_r]$ whose left and right endpoints are chosen so that the correct result must lie in the interval.

Calculations are done using intervals and will automatically deliver an interval result that gives the answer as the midpoint with an error no more than half the width of that interval.

However, the bane of interval arithmetic is that for some calculations, the size of the intervals representing intermediate results grows exponentially as the calculation proceeds. This can produce intervals for the final results that are too large to guarantee any accuracy at all. Many of these calculations are not ill-conditioned, and using ordinary arithmetic will give quite accurate results, despite the fact that interval arithmetic fails to give interval results with small width.

Multiple-precision can delay the onset of intervals that are too large, but sometimes even carrying high precision will not allow the interval calculation to succeed. See the examples in Section 4 below.

2. ALGORITHMS

To find the interval that results from an arithmetic operation or function with interval inputs, we find the left endpoint of the result using higher precision and round toward $-\infty$, then find the right endpoint and round toward $+\infty$.

2.1 Basic Arithmetic

The four basic arithmetic operations are fairly straightforward.
Let $[a, b]$ and $[c, d]$, with $a \le b$, $c \le d$, be the input arguments.

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

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

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

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

If zero is contained in the interval $[c, d]$, then division is undefined and we set $[a, b]/[c, d] = [-\infty, +\infty]$.

Multiplication and division can be done faster than these definitions by splitting the logic into several special cases. These are the 9 cases for multiplication.

(1) If $a \ge 0$ and $c \ge 0$, then return $[a * c, b * d]$,
(2) If $a \ge 0$, $c < 0$ and $d \ge 0$, then return $[b * c, b * d]$,
(3) If $a \ge 0$, and $d < 0$, then return $[b * c, a * d]$,
(4) If $a < 0$, $b \ge 0$, and $c \ge 0$, then return $[a * d, b * d]$,
(5) If $a < 0$, $b \ge 0$, $c < 0$ and $d \ge 0$, then return $[\min(a * d, b * c), \max(a * c, b * d)]$,
(6) If $a < 0$, $b \ge 0$, and $d < 0$, then return $[b * c, a * c]$,
(7) If $b < 0$ and $c \ge 0$, then return $[a * d, b * c]$,
(8) If $b < 0$, $c < 0$ and $d \ge 0$, then return $[a * d, a * c]$,
(9) If $b < 0$ and $d < 0$, then return $[b * d, a * c]$.

All but one of these cases can be done with two multiplications, case (5) takes four multiplications.

Division is similar, except that the three cases where $c \le 0 \le d$ all return $[-\infty, +\infty]$, and the other six cases each use two divisions.

2.2 Monotonic Functions

Functions that are either increasing or decreasing over their whole domain are easy in interval arithmetic. For example, e^x is increasing, so

$\exp([a, b]) = [e^a, e^b]$.

$\arccos(x)$ is decreasing, so

$\arccos([a, b]) = [\arccos(b), \arccos(a)]$.

Other monotonic functions include $\arcsin(x)$, $\arctan(x)$, $\sqrt{x}$, $\sinh(x)$, $\tanh(x)$, $\ln(x)$, $\sinh^{-1}(x)$, $\cosh^{-1}(x)$, $\tanh^{-1}(x)$, $\mathrm{erf}(x)$, $\mathrm{erfc}(x)$, $\mathrm{chi}(x)$, $\mathrm{shi}(x)$.

2.3 Non-Monotonic Functions

If the function has local maxima or minima or if it has singularities, then the logic for computing an interval result can be much more complicated.

For $\sin([a,b])$ or $\cos([a,b])$, all the extreme points are known, so we can quickly determine whether any extreme point lies between a and b. If not, then the function is monotonic on $[a,b]$ and we can proceed as in section 2.2. If one or more extreme points are in $[a,b]$, we have to consider the corresponding extreme function values (± 1 for sin or cos) as well as the function values at endpoints a and b.

A function like $\tan(x)$ has singularities that must be considered. If any singularity is contained in $[a,b]$, then the result returned for $\tan([a,b])$ is $[-\infty,+\infty]$. Otherwise $\tan(x)$ is increasing on $[a,b]$, so the result is $[\tan(a),\tan(b)]$.

$\psi(x)$ is similar to $\tan(x)$. $\mathrm{li}(x)$ is decreasing on one side of its singularity and increasing on the other, making it slightly more complicated.

$\Gamma(x)$ has both extreme points and singularities. The singularities are easy to handle, since they are at the non-positive integers. Extreme points are not known from simple formulas, as with sin and cos, so they must be computed if they lie in $[a,b]$. Brent's method [1973] is used to find these max/min values, which makes this case for $\Gamma([a,b])$ much slower.

Other oscillatory functions like the sine and cosine integrals $\mathrm{si}(x)$ and $\mathrm{ci}(x)$, Fresnel integrals $\mathrm{S}(x)$ and $\mathrm{C}(x)$, and Bessel functions $\mathrm{J}_n(x)$ and $\mathrm{Y}_n(x)$ are similar to the gamma function in the way extreme points are handled.

3. EFFICIENCY

As shown in the previous section, most of the time an interval operation, whether arithmetic or function, can be done with two ordinary FM operations of the same type with directed rounding. This means the typical time for an interval operation is slightly more than twice the time for the corresponding FM operation. See the function timing page at the FM web site:

http://myweb.lmu.edu/dmsmith/v3timing.html

The exceptions to this are cases where the input interval is large and the function has one or more extreme points. Solving for each max/min point in the input interval can take 10 or 20 function calls to the corresponding FM function, making the interval operation that much slower.

4. EXAMPLES

To give some feeling for how interval calculations behave, here are some examples. The program IntervalExamples.f95 at the FM web site has the code.

In these examples the FM precision level is set to 50 significant digits using `CALL FM_SET(50)`. The default base used internally by FM is 10^7, and asking for 50 digits will set precision slightly higher. Using base 10^7 packs 7 decimal digits into each word, except that normalization can give from 1 to 7 decimals in the first one. This means using 9 base 10^7 digits gives precision that is always at least 8*7+1 = 57 significant digits, and may be as high as 9*7 = 63.

Example 1.

Many basic mathematical identities are false with interval arithmetic. For example, if the input interval is $x = [-0.5, 1.0]$, then $x^2 \neq x * x$. $x^2 = [0.0, 1.0]$, but $x * x = [-0.5, 1.0]$. This happens because the general multiplication routine for $a*b$ has to assume that a and b are different intervals and that a number in interval a can be chosen independently from a number in interval b.

The FM interval package has separate routines for squaring and multiplication, and they will give different results for x^2 vs $x * x$ whenever zero is inside the interval x. For this reason, x^2 is preferred over $x * x$ when doing interval arithmetic, since x^2 will sometimes give a better result (smaller interval) and never give a worse result.

We will see in the examples below that different calculations can result in the width of the intervals growing at quite different rates. When the intervals grow at a linear rate, the graph of the base 10 log of the width of the intervals for intermediate results curves and looks roughly like $\log(x)$. When they grow at an exponential rate, the graph is roughly a straight line.

As a simple example, evaluate the quadratic polynomial $f(x) = x^2 - x + 3$ using several mathematically equivalent but computationally different ways of coding the formula, and several different input intervals.

Define the "magnification" factor for a function $f(x)$ evaluated at input interval x to be the width of the output interval of the computation divided by the width of the smallest interval containing the set $\{f(a) \,|\, a \in x\}$. In the case of the interval $x = [-0.5, 1.0]$ and the formula $x * x$, the magnification factor is 1.5.

formula	input	output	magnification
x**2 - x + 3	[-0.5 , 1.0]	[2.00 , 4.50]	2.500
x*x - x + 3	[-0.5 , 1.0]	[1.50 , 4.50]	3.000
x*(x - 1) + 3	[-0.5 , 1.0]	[1.50 , 3.75]	2.250
(x - 0.5)**2 + 2.75	[-0.5 , 1.0]	[2.75 , 3.75]	1.000

```
x**2 - x + 3           [ 0.1 , 1.0 ]     [ 2.01 , 3.90 ]     7.560
x*x - x + 3            [ 0.1 , 1.0 ]     [ 2.01 , 3.90 ]     7.560
x*(x - 1) + 3          [ 0.1 , 1.0 ]     [ 2.10 , 3.00 ]     3.600
(x - 0.5)**2 + 2.75    [ 0.1 , 1.0 ]     [ 2.75 , 3.00 ]     1.000

x**2 - x + 3           [ 0.9 , 1.0 ]     [ 2.81 , 3.10 ]     3.222
x*x - x + 3            [ 0.9 , 1.0 ]     [ 2.81 , 3.10 ]     3.222
x*(x - 1) + 3          [ 0.9 , 1.0 ]     [ 2.90 , 3.00 ]     1.111
(x - 0.5)**2 + 2.75    [ 0.9 , 1.0 ]     [ 2.91 , 3.00 ]     1.000

x**2 - x + 3           [ 0.99 , 1.0 ]    [ 2.98 , 3.01 ]     3.020
x*x - x + 3            [ 0.99 , 1.0 ]    [ 2.98 , 3.01 ]     3.020
x*(x - 1) + 3          [ 0.99 , 1.0 ]    [ 2.99 , 3.00 ]     1.010
(x - 0.5)**2 + 2.75    [ 0.99 , 1.0 ]    [ 2.99 , 3.00 ]     1.000
```

In all of these cases, for the natural way of coding the formula, $x^2 - x + 3$, the output interval is always at least 2.5 times bigger than it could be. Also, as the width of the input interval approaches zero when the left endpoint of the input interval approaches 1, the magnification factor stays above 3. This means even early in a calculation when the width of the intervals is very small, the widths will increase quickly. In any sequence of calculations, if the magnification factors stay above 1 the whole time (and don't converge to 1), then the width of the intervals will grow exponentially.

The formula $(x - 0.5)^2 + 2.75$ gives the optimal result in each case because the x appears only once in the formula, so the interval calculation cannot go wrong by assuming different elements from interval x could be used in different instances of x in the formula. But this sort of algebraic manipulation cannot usually be done for the much more complicated calculations in a real program.

Example 2.

Evaluate a simple sum: $\displaystyle\sum_{n=1}^{100{,}000} \frac{1}{n^7}$.

The two endpoints of the interval result print as:

1.00834927738192282683979754984963009793313856056 52

1.00834927738192282683979754984963009793313856056 53

This is an example where interval arithmetic works well and the two endpoints agree to about 51 decimal digits. Looking at the intermediate results of the sum shows the width of the intervals is increasing fairly slowly.

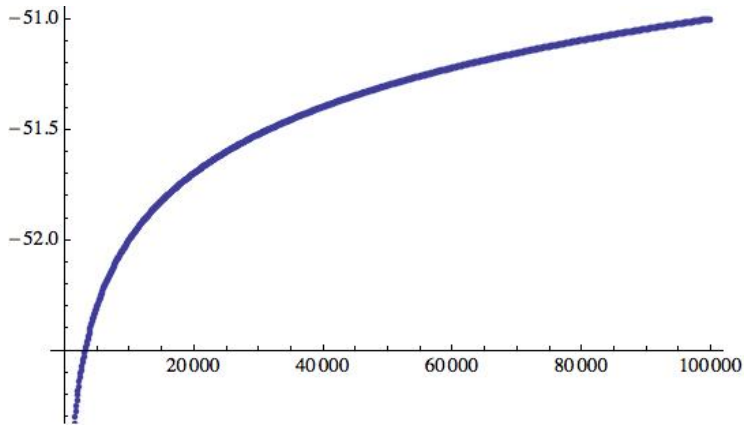

Figure 1. $\log_{10}$(interval width) as a function of number of terms in the sum

So after 10,000 terms the width of the interval for the partial sum is about 10^{-52}, and at the end of 100,000 terms, the interval width is about 10^{-51}.

Example 3.

Use composite 9-point Gauss quadrature with 1,000 subintervals to approximate $\displaystyle\int_0^{20} \frac{\sin(t)}{t}\, dt$.

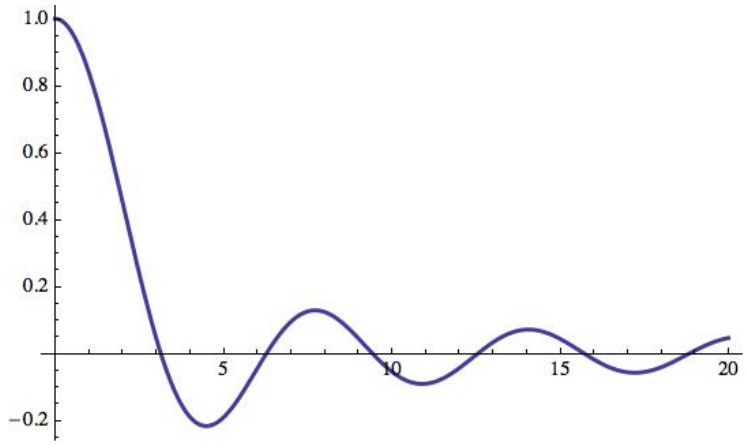

Figure 2. $\sin(t)/t$

The two endpoints of the interval result print as:

1.54824170104343984016364334212951369226157336210 93

1.54824170104343984016364334212951369226157336210 94

This calculation is more complicated than the previous example, but the integration formula is still basically a sum, and again interval arithmetic works well with the two endpoints agreeing to about 52 decimal digits.

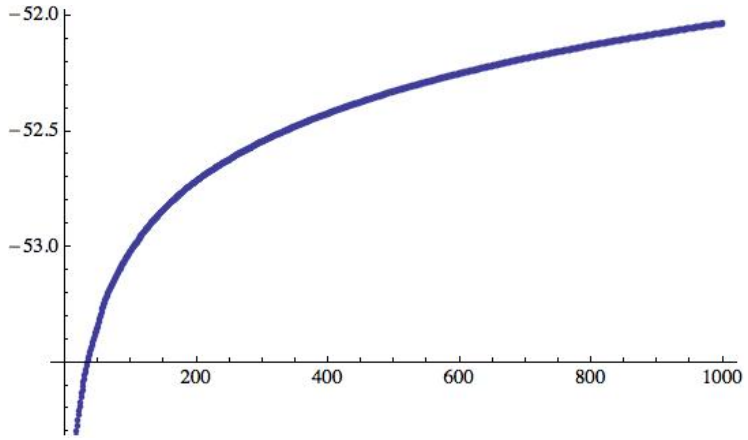

Figure 3. $\log_{10}$(interval width) as a function of subinterval number in the integration formula

Example 4.

Start with $(x, y) = (1, 0)$ and step around the unit circle using the recurrence $(x, y) \leftarrow (x\,c - y\,s\ ,\ y\,c + x\,s)$, where $c = \cos(2\pi/n)$ and $s = \sin(2\pi/n)$.

Taking n steps with this recurrence should bring (x, y) back to $(1, 0)$. The program uses $n = 100$ and makes 23 trips around the circle, at which point the final intervals for x and y are: $x = [-58.564, 60.564]$, $y = [-59.564, 59.564]$, which is useless as an approximation to $x = 1$ and $y = 0$. It happens that the midpoints of these two intervals are accurate in approximating x and y, but that is certainly not true in general, so an output interval this big gives no guaranteed accuracy.

To make sure that the failure is really due to the nature of interval arithmetic and not some normalization error effect brought on by using a large base, this case was also run using base 2 arithmetic with 189 digits (bits), which also gives about 57 significant digits base 10.

The red graph below is the base 2 interval width for x, the blue graph is from base 10^7.

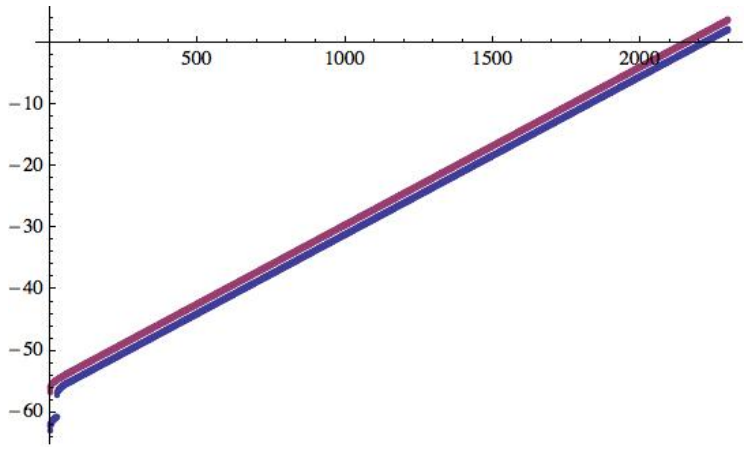

Figure 4. $\log_{10}$(interval width) as a function of step number in the recurrence

There was a similar exponential growth in the interval size in a more realistic and more complicated program for computing the 3-dimensional elliptical orbit of a satellite around the earth. After a few dozen orbits, the intervals for (x, y, z) had left and right endpoints with no significant digits of agreement.

These are not numerically ill-conditioned problems. Running the recurrence with non-interval FM arithmetic gave accurate results after 23 trips of 100 steps each:

$x = 1.000000000000000000000000000000000000000000000000$

$y = -2.27451021\text{e-}55$

Also, it is not the fact that x and y oscillate through positive and negative values that causes this rapid rise in the interval widths. The following graph comes from starting from the point $(x, y) = (\cos(8 * 2\,\pi/n), \sin(8 * 2\,\pi/n))$, 8 steps into the original iteration, then doing 10 steps with that recurrence, then returning after 10 backward steps using the reverse recurrence

$(x, y) \leftarrow (x\,c + y\,s\ ,\ y\,c - x\,s)$, where $c = \cos(2\pi/n)$ and $s = \sin(2\pi/n)$.

Five of these back-and-forth cycles of 20 steps each corresponds to one 100-step trip around the circle, so 5*23 = 115 of these cycles takes the same number of steps as before, as (x, y) varies from $(0.8763, 0.4818)$ along the circle to $(0.4258, 0.9048)$ and back, never crossing zero for x or y.

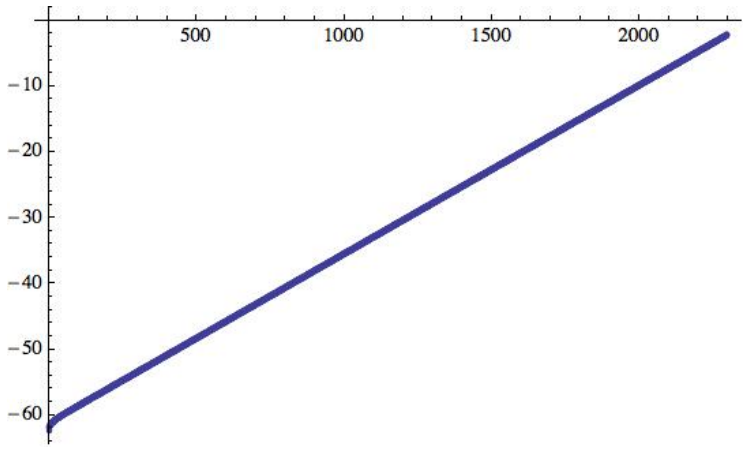

Figure 5. $\log_{10}$(interval width) as a function of step number in the back-and-forth recurrence

Example 5.

Evaluate a product: $\displaystyle\prod_{n=1}^{10{,}000} \frac{4\,n^2}{4\,n^2-1}$.

The two endpoints of the interval result print as:

3.14151411868192204697855805071387755134251333947 00

3.14151411868192204697855805071387755134251333947 01

The interval widths in this case are about as well-behaved as those in the summation of example 2.

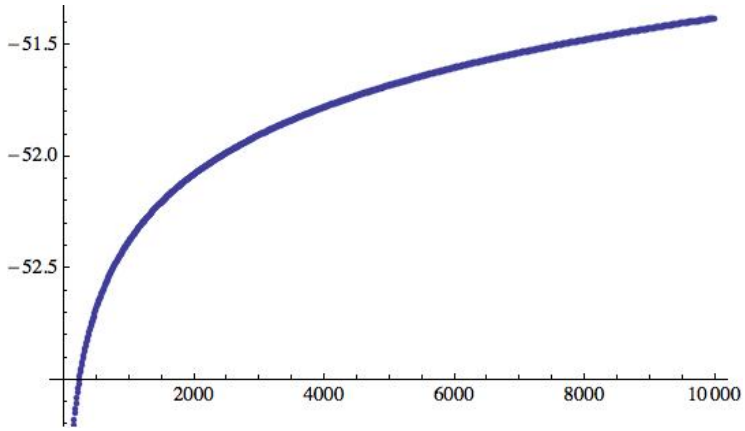

Figure 6. $\log_{10}$(interval width) as a function of number of terms in the product

Example 6.

Differential equation. $y'' = -\frac{1}{10}\,y' - \frac{2}{x+2}\,y, \quad y(0) = 0, \quad y'(0) = 1.$

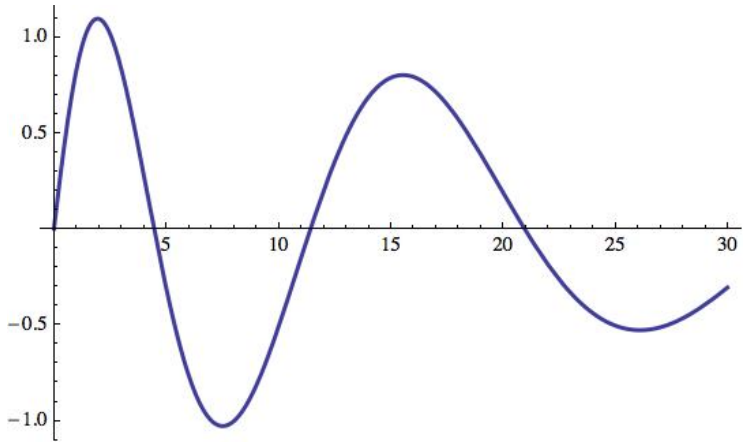

Figure 7. Solution $y(t)$ to the differential equation

The two endpoints of the interval result print as:

-0.30776688167603169203382987651096461581717811055464

-0.30776688167603169203382987651096461581717811055418

This came from using 4th-order Runge-Kutta with 10,000 steps.

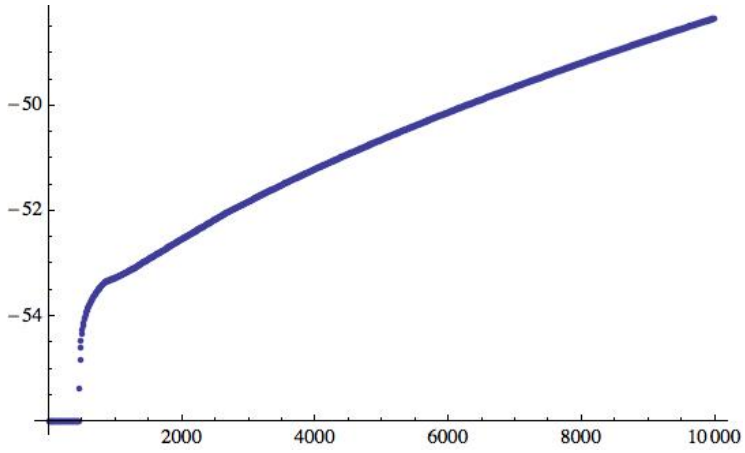

Figure 8. $\log_{10}$(interval width) as a function of step number

Here the accuracy of the interval arithmetic is degrading somewhat faster than in examples 2 and 5, but at the end the left and right endpoints for $y(30)$ agree to 48 significant digits, so the

interval arithmetic has handled this case successfully.

Example 7.

Differential equation. $y'' = -\frac{1}{10}\,y' - \frac{200}{x+2}\,y, \quad y(0) = 0, \quad y'(0) = 1.$

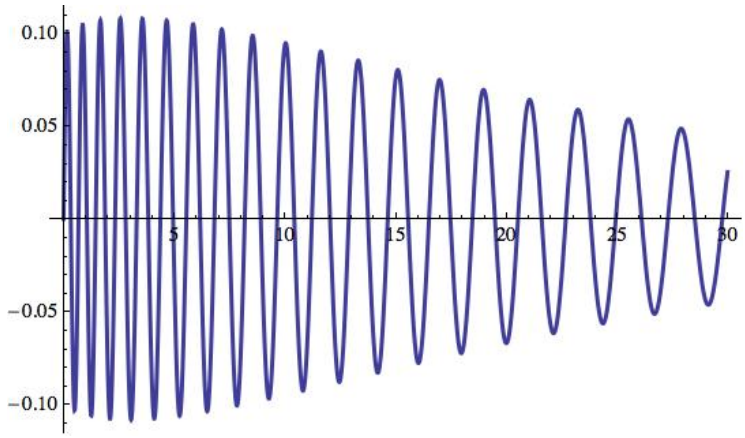

Figure 9. Solution $y(t)$ to the differential equation

The two endpoints of the interval result print as:

0.025268274663776318683290452013354524008325842304305

0.025282426126969610817979865312140233217038151226782

This came from using 4th-order Runge-Kutta with 10,000 steps.

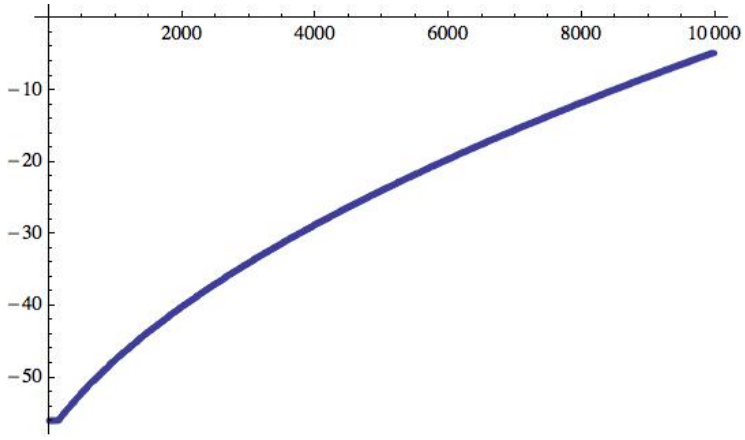

Figure 10. $\log_{10}$(interval width) as a function of step number

Here the solution has 38 roots between 0 and 30, and the many oscillations seem to be causing

rapid growth in the interval size, like the cases in example 4. There is only 3 significant digit agreement in the left and right endpoints of $y(30)$. Unlike the circle recurrence of example 4, here the midpoint of the final interval is not much better, having only 6 correct significant digits.

Example 8.

Solution of $n \times n$ linear systems. Matrix equations $A\,x = b$ were formed for the 10 cases $n = 10, 20, 30, \ldots, 100$. In each case the entries of the A matrix were random numbers between 0 and 1, and the b vector was $(1, 2, 3, \ldots, n)$.

After solving the system using Gauss elimination with partial pivoting, the smallest and largest intervals of the n entries of the x solution vector were found.

For n = 10 the solution elements agree to between 51 and 53 significant digits.
For n = 20 the solution elements agree to between 47 and 50 significant digits.
For n = 30 the solution elements agree to between 44 and 48 significant digits.
For n = 40 the solution elements agree to between 36 and 46 significant digits.
For n = 50 the solution elements agree to between 35 and 43 significant digits.
For n = 60 the solution elements agree to between 28 and 40 significant digits.
For n = 70 the solution elements agree to between 27 and 37 significant digits.
For n = 80 the solution elements agree to between 23 and 34 significant digits.
For n = 90 the solution elements agree to between 16 and 33 significant digits.
For n = 100 the solution elements agree to between 16 and 30 significant digits.

This is another type of problem where interval arithmetic is having more trouble as n gets larger, but ordinary FM arithmetic would produce close to full accuracy.

5. CONCLUSION

As Kahan [2006] points out, interval arithmetic can often give a guarantee that the computed result is accurate enough. But he also warns that there are no universal methods that work for all problems.

Examples 4, 7, and 8, where interval arithmetic has trouble, show how multiple precision interval arithmetic can overcome some of the hard cases. By using much higher precision levels, even some of the cases where the width of the intervals is growing exponentially can be finished before the intervals become too wide.

Besides problems where the intermediate results oscillate, the most common reason why interval results are often far too pessimistic is that correlation between terms in formulas is ignored by interval arithmetic.

Kahan lists several ways that a given numerical computation can be tested to estimate how much accuracy may have been lost due to rounding errors. These include re-running the program with the same precision and different rounding modes, running the program with a sequence of increasing precision levels, and running the program using interval arithmetic.

The basic FM package for floating-point multiple-precision computation supports the four different rounding modes and dynamic precision control. This package for multiple-precision interval arithmetic can support the third option.

References

Brent, R.P. 1973. *Algorithms for Minimization Without Derivatives*,
Prentice-Hall, Englewood Cliff, New Jersey.
More recently reprinted by Dover Publications (March 20, 2013).

Kahan, W. 2006. How Futile are Mindless Assessments of Roundoff in Floating-Point Computation? http://www.cs.berkeley.edu/∼wkahan/Mindless.pdf

Moore, R.E. 1966. *Interval Analysis*, Prentice-Hall, Englewood Cliff, New Jersey.

Smith, D.M. 1991. A Fortran Package for Floating-Point Multiple-Precision Arithmetic.
ACM Trans. Math. Softw. 17, 2 (June), 273–283.
The current version of the code is on the FM website:
http://myweb.lmu.edu/dmsmith/FMLIB.html