Interval Computations
No 3, 1993

Interval Arithmetic in Mathematica

Jerry B. Keiper

The use of interval methods to solve numerical problems has been very limited. Some of the reasons for this are the lack of easy access to software and the lack of knowledge of the potential benefits. Until numerical analysts and others become generally familiar with interval methods, their use will remain limited. Mathematica is one way to educate potential users regarding the usefulness of interval methods. This paper examines some of the ways that intervals can be used in Mathematica.

ИНТЕРВАЛЬНАЯ АРФМЕТИКА В СИСТЕМЕ Mathematica

Дж. Б. Кейпер

Использование интервальных методов для решения численных задач до сих пор было весьма ограниченным. Отчасти причины этого заключались в отсутствии легкого доступа к программному обеспечению и отсутствии знаний о потенциальной выгоде. Пока исследователи, занимающиеся численным анализом и смежными дисциплинами, не познакомятся достаточно близко с интервальными методами, их использование будет оставаться ограниченным. Доступ к системе Mathematica является одним из способов ознакомить потенциальных пользователей с полезностью интервальных методов. В статье рассматриваются некоторые возможности использования интервалов в системе Mathematica.

1 Introduction

Mathematica, like all computer-algebra systems, has several types of arithmetic. In addition to exact arithmetic (*i.e.*, integers and rational numbers) and machine-precision floating-point arithmetic, it has high-precision floating-point arithmetic and interval arithmetic.

The behavior of the high-precision floating-point arithmetic can be altered (by changing `$MinPrecision` and `$MaxPrecision`), but the default behavior is essentially a form of interval arithmetic in which the intervals are assumed to be "short" relative to the magnitude of the numbers represented. This form of arithmetic is sometimes referred to as significance arithmetic or range arithmetic. The length of the interval that is implicitly represented by a high-precision floating-point number is

$$10^{-a}$$

where a is the "accuracy" of the number: roughly the number of digits to the right of the decimal point, although it is usually not an integer. In the implementation, each high-precision floating-point number has a value for a associated with it, stored as a machine float. This value represents the logarithm of the length of the interval; the length itself is not explicitly calculated nor are the digits actually counted.

There are problems, however, where range arithmetic is not sufficient and you really do want to use interval arithmetic. The interval arithmetic in Mathematica Version 2.2 is genuine interval arithmetic, complete with outward rounding and multi-intervals. Note however that compromises must be made in a system like Mathematica, which must run on a variety of machines. The outward rounding is done *a posteriori* rather than as directed rounding in hardware, which many machines do not support. For example, to evaluate $\sin x$ on the interval $2.1 < x < 2.2$, where $\sin x$ is monotonic, $\sin(2.1)$ and $\sin(2.2)$ are evaluated using the standard library functions provided with the computer. These values are then rounded outward. Thus the intervals tend to grow slightly more rapidly than is absolutely necessary. Also, an assumption is made that is known to be false: library functions for the elementary functions are assumed to be correct to within one ulp and directed rounding by one ulp is used to "ensure" that resulting interval contains the image of the argument. There are no known examples for which the elementary functions are in error by more than an ulp for high-precision arithmetic. Complex intervals have not yet been implemented.

2 Range arithmetic

The "accuracy" of a number is given by the function `Accuracy[ ]`. This function simply gives access to the number a explained above, but by default its result is rounded to the nearest integer. This rounding can be disabled to see more clearly what is happening.

This disables the rounding of the results of **Accuracy[ ]** and **Precision[ ]**:

```
In[1]:= SetPrecision[Round, False];
```

Define a function **len[ ]** to give the length of the implicit interval associated with the real number **x**:

```
In[2]:= len[xReal] := 10^(-Accuracy[x])
```

Define **a** to be the 30-digit approximation to 3 and examine the length of its implicit interval. (**Accuracy[a]** ≈ 29.5229 and $10^{-29.5229} \approx 2.99999 \times 10^{-30}$.)

```
In[3]:= a = N[3, 30]; len[a]

                      -30
Out[3]= 2.99999 10
```

Do the same for the number 4:

```
In[4]:= b = N[4, 30]; len[b]

                -30
Out[4]= 4. 10
```

Find the lengths of the intervals resulting from various arithmetic operations on these two numbers:

```
In[5]:= {len[a+b], len[a-b], len[a b], len[a/b]}

                       -30              -30              -29          -30
Out[5]= {7.00001 10   , 7.00001 10   , 2.39999 10   , 1.5 10   }
```

Note that from the point of view of rigorous interval arithmetic, the lengths of the intervals implicit in range arithmetic are a bit sloppy. This is because the purpose of range arithmetic is merely to give a good estimate of the number of correct digits in the result. To reduce memory requirements, the logarithm of the length of the interval is stored only as a float. To increase the speed of the length calculations, a rational minimax approximation is used to evaluate the accuracy of the result of an operation. Because interval arithmetic is rather pessimistic and because range arithmetic is not intended to be a substitute for interval arithmetic, sloppy length calculations are not problematic. Note that a much more serious problem with attempting to view range arithmetic as interval arithmetic is that the length calculations assume that the interval lengths are "short" relative to the magnitude of the numbers they represent. For example, if we have two intervals $x(1+\varepsilon_x)$ and $y(1+\varepsilon_y)$ where their relative half-lengths are represented by ε_x and ε_y, the product of the two intervals is calculated as $xy(1+\varepsilon_x+\varepsilon_y)$, *i.e.* the second order term $\varepsilon_x\varepsilon_y$ in the relative half-length is ignored. (Note: the implementation does not deal with half-length, but rather with the "accuracy" a. Rational minimax approximations are used to achieve the behavior explained here in terms of half-length.) For even moderate precision, say 10 digits or so, this works quite nicely, but for numbers with only 1 or 2 digits the length calculations for certain operations can be inaccurate. In many problems range arithmetic is all that is needed.

If we start with a 30-digit approximation to π and subtract a rational approximation we can lose many digits: the total number of digits displayed in (*i.e.* the "precision" of) a number corresponds to the logarithm of the *relative* length of the interval:

```
In[6]:= N[Pi, 30] - 3141592653589793238462 6/1000000000000000000000
```

```
                         -23
Out[6]= 4.338328 10
```

For very well-conditioned functions we can *gain* many digits. In this example we started with a 20-digit approximation to the number 20 and got a 175-digit result:

```
In[7]:= Erf[N[20, 20]]

Out[7]= 0.99999999999999999999999999999999999999999999999999999\
   9999999999999999999999999999999999999999999999999999999999999\
   9999999999999999999999999999999999999999999999999999999999999
```

Rump [?] describes the following problem: evaluate

$$c = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

where $a = 77617$ and $b = 33096$. With fixed-precision floating-point arithmetic one does not know how many digits of the result are correct. In fact, unless rather high-precision arithmetic is used the answer will be completely wrong. In Rump's calculations, single precision, double precision, and even extended precision gave the *same wrong* result: even the sign was wrong.

This function evaluates c starting with **n** digits:

```
In[8]:= f[n_] := Block[{a = N[77617, n], b = N[33096, n]},
                   N[333+3/4, n] b^6 + a^2 (11 a^2 b^2 - b^6
                       - 121 b^4 - 2) + N[5 + 1/2, n] b^8 + a/(2 b)]
```

10 digits is less than **$MachinePrecision**, so the first calculation is done using machine numbers and is completely wrong:

```
In[9]:= {f[10], f[20], f[30], f[40], f[50]}

                     21       8
Out[9]= {-1.18059 10   , 0. 10 , 0.0, -0.83, -0.827396059947}
```

The first three digits of the result which used 30 digits were in fact correct, but none of the digits was known to be correct so none was displayed. We can uncover the hidden digits:

```
In[10]:= SetPrecision[%[[3]], 10]

Out[10]= -0.8274078369
```

We can let Mathematica worry about how much precision is required to get a specified precision in the result.

```
In[11]:= n = 20; While[Precision[c = f[n]] < 10, n += 5]; c

Out[11]= -0.827396059947
```

3 Interval arithmetic

When range arithmetic is not sufficient to solve a problem the interval arithmetic provided by Mathematica can be used.

This is the interval from 2 to 3:

```
In[12]:= a = Interval[{2, 3}]

Out[12]= Interval[{2, 3}]
```

This is the interval from -2 to 1:

```
In[13]:= b = Interval[{-2, 1}]

Out[13]= Interval[{-2, 1}]
```

You can do arithmetic with intervals:

```
In[14]:= c = a + b

Out[14]= Interval[{0, 4}]
```

Division by an interval containing 0 results in two half-infinite intervals:

```
In[15]:= a/b

Out[15]= Interval[{-Infinity, -1}, {2, Infinity}]
```

Inequalities also work with intervals:

```
In[16]:= a > b

Out[16]= True
```

The functions **IntervalUnion[ ]**, **IntervalIntersection[ ]**, and **IntervalMemberQ[ ]** can be used to manipulate intervals:

```
In[17]:= IntervalUnion[a, b]

Out[17]= Interval[{-2, 1}, {2, 3}]
```

The elementary functions are defined on intervals:

```
In[18]:= Sin[a]

Out[18]= Interval[{Sin[3], Sin[2]}]
```

Internal extrema are not a problem:

```
In[19]:= Sin[b]

Out[19]= Interval[{-1, Sin[1]}]
```

Exact singletons are rewritten as intervals of length 0:

```
In[20]:= Interval[Pi]

Out[20]= Interval[{Pi, Pi}]
```

Inexact singletons are also rewritten. This looks like another interval of length 0:

```
In[21]:= d = Interval[N[Pi]]

Out[21]= Interval[{3.14159, 3.14159}]
```

In fact it has a length of 2 ulps: 1 ulp from rounding each endpoint:

```
In[22]:= d[[1,2]] - d[[1,1]]

                      -16
Out[22]= 8.88178 10
```

4 Interval plotting

Interval plotting can be done by plotting the curves defined by the endpoints of the resulting intervals.

Define a function that graphs a single interval function:

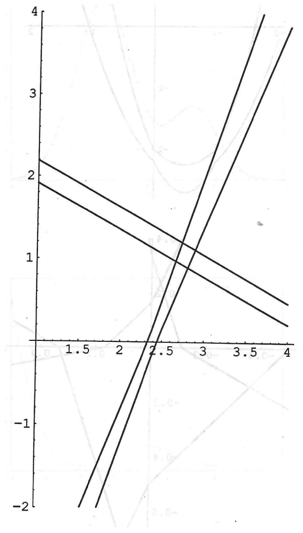
```
In[23]:= iplot[y_, r_, opt___] :=
             Plot[{Min[y], Max[y]}, r, AspectRatio -> Automatic, opt]
```

Define how it should plot a list of interval functions:

```
In[24]:= iplot[{y__}, r_, opt___] :=
             Show[Map[iplot[#, r, DisplayFunction->Identity]&, {y}],
                 DisplayFunction:>$DisplayFunction, opt]
```

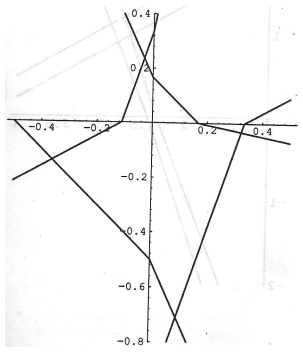
Plot two "lines":

```
In[25]:= iplot[{(Interval[{5.7,5.8}]-Interval[{1.2,1.3}] x)/
                        Interval[{2.1,2.3}],
                (Interval[{7.7,7.8}]-Interval[{3.2,3.3}] x)/
                        Interval[{-1.1,-1.3}]},
               {x, 1, 4}, PlotRange -> {-2, 4}]
```



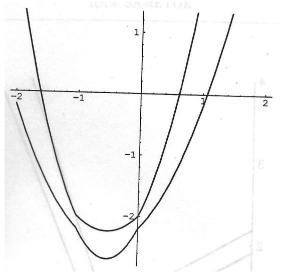
The intersection of two "lines" can be quite irregular:

```
In[26]:= iplot[{(Interval[{-0.3,0.1}]-Interval[{0.6,1.3}] x)/
                        Interval[{0.6,2.9}],
                (Interval[{-0.1,0.3}]-Interval[{0.9,1.6}] x)/
                        (-Interval[{0.3,1.6}])},
               {x, -.5, .5}, PlotRange -> {-.8, .4}]
```



iplot[ ] can plot nonlinear functions as well:

```
In[27]:= iplot[Interval[{1,2}] (x^2+x) - Interval[{2,2.2}], {x,-2,2}]
```



5 Interval rootfinding

The following examples are not intended to be complete, robust algorithms. They are merely intended to illustrate some of the possibilities for interval algorithms in Mathematica.

5.1 Bisection

In a bisection search a given interval is recursively bisected and each half is tested for roots. In interval arithmetic, testing for roots is done by testing whether 0 is in the image of the interval.

Define an error message to warn the user when the recursion limit is encountered:

```
In[1]:= intervalbisection::rec = "MaxRecursion exceeded.";
```

Define the recursive element of the algorithm:

```
In[2]:= split[f_, x_, int_Interval, eps_, n_] :=
            Block[{a = int[[1,1]], b = int[[1,2]], c},
                If[!IntervalMemberQ[f /. x -> int, 0], Return[{}]];
                If[b-a < eps, Return[int]];
                If[n == 0, Message[intervalbisection::rec]; Return[int]];
                c = (a+b)/2;
                split[f, x, #, eps, n-1]& /@
                    {Interval[{a,c}], Interval[{c,b}]}
            ];
```

Give **intervalbisection** the option **MaxRecursion** with a default value of 7:

```
In[3]:= Options[intervalbisection] = {MaxRecursion -> 7};
```

Define the function **intervalbisection[ ]**:

```
In[4]:= intervalbisection[f_, x_, intab_, eps_, opts___] :=
            Block[{int, n},
                n = MaxRecursion /. {opts} /. Options[intervalbisection];
                int = Interval /@ (List @@ intab);
                IntervalUnion @@ Flatten[split[f, x, #, eps, n]& /@ int]
                ];
```

Find the roots of the function **Sin[ ]** on **Interval[{2., 20.}]**. Here 7 recursive bisections were not sufficient to shorten the enclosing intervals to 0.1:

```
In[5]:= intervalbisection[Sin[x], x, Interval[{2., 20.}], .1]

intervalbisection::rec: MaxRecursion exceeded.
General::stop: Further output of intervalbisection::rec
     will be suppressed during this calculation.
Out[5]= Interval[{3.125, 3.26562}, {6.21875, 6.35938}, {9.3125, 9.45313},
  {12.5469, 12.6875}, {15.6406, 15.7813}, {18.7344, 18.875}]
```

We can continue from where the previous calculation encountered **MaxRecursion**:

```
In[6]:= intervalbisection[Sin[x], x, %, .1]

Out[6]= Interval[{3.125, 3.19531}, {6.21875, 6.28906}, {9.38281, 9.45313},
  {12.5469, 12.6172}, {15.6406, 15.7109}, {18.8047, 18.875}]
```

Setting **MaxRecursion** higher is another way to get convergence:

```
In[7]:= intervalbisection[Sin[x], x, Interval[{2., 20.}], .1,
            MaxRecursion -> 10]

Out[7]= Interval[{3.125, 3.19531}, {6.21875, 6.28906}, {9.38281, 9.45313},
  {12.5469, 12.6172}, {15.6406, 15.7109}, {18.8047, 18.875}]
```

Of course we cannot separate all of the roots when there are infinitely many:

```
In[8]:= intervalbisection[Sin[1/x], x, Interval[{-1., 1.}], .01]

intervalbisection::rec: MaxRecursion exceeded.
General::stop: Further output of intervalbisection::rec
     will be suppressed during this calculation.
Out[8]= Interval[{-0.328125, -0.3125}, {-0.171875, -0.15625},
  {-0.109375, 0.109375}, {0.15625, 0.171875}, {0.3125, 0.328125}]
```

5.2 Newton's method

With Newton's method the idea is to pick a point in the given interval and evaluate the function at that point. Evaluating the derivative of the function on the interval allows us to eliminate parts of the original interval (assuming that the derivative is bounded).

Define an error message to warn the user when the recursion limit is encountered.

```
In[1]:= intervalnewton::rec = "MaxRecursion exceeded.";
```

Define the recursive element of the algorithm:

```
In[2]:= intnewt[f_, jac_, x_, {a_, b_}, eps_, n_] :=
            Block[{xmid, int = Interval[{a, b}]},
                If[b-a < eps, Return[int]];
                If[n == 0, Message[intervalnewton::rec]; Return[int]];
                xmid = Interval[(a+b)/2];
                int = IntervalIntersection[int,
                xmid - N[f /. x -> xmid]/N[jac /. x -> int]];
                (intnewt[f, jac, x, #, eps, n-1])& /@ (List @@ int)
                ];
```

Give **intervalnewton** the option **MaxRecursion** with a default value of 7:

```
In[3]:= Options[intervalnewton] = {MaxRecursion -> 7};
```

Define the function **intervalnewton[ ]**:

```
In[4]:= intervalnewton[f_, x_, int_Interval, eps_, opts___] :=
            Block[{jac, n},
                n = MaxRecursion /. {opts} /. Options[intervalnewton];
                jac = D[f, x];
                IntervalUnion @@ Select[ Flatten[
                    (intnewt[f,jac,x,#,eps,n])& /@ (List @@ int)],
                    IntervalMemberQ[N[f /. x -> #], 0]&]
                ];
```

Find the roots of the function **Sin[ ]** on **Interval[{2., 20.}]**:

```
In[5]:= intervalnewton[Sin[x], x, Interval[{2., 20.}], .1]

Out[5]= Interval[{3.11564, 3.20189}, {6.27942, 6.29789}, {9.40122, 9.4251},
  {12.5661, 12.588}, {15.6936, 15.7121}, {18.7924, 18.8764}]
```

Continue working until each interval is less than 0.00001 in length:

```
In[6]:= intervalnewton[Sin[x], x, %, 0.00001]

Out[6]= Interval[{3.14159, 3.14159}, {6.28318, 6.28319}, {9.42478, 9.42478},
  {12.5664, 12.5664}, {15.708, 15.708}, {18.8496, 18.8496}]
```

Multiple roots simply take more iterations and hence more time:

```
In[7]:= intervalnewton[Sin[x]^2, x, Interval[{2., 20.}], .1]

Out[7]= Interval[{3.12024, 3.20076}, {6.25056, 6.34969}, {9.41445, 9.4889},
  {12.5054, 12.5809}, {15.6982, 15.7491}, {18.7928, 18.8748}]
```

When the derivative is unbounded there is a problem:

```
In[8]:= intervalnewton[Sin[1/x], x, Interval[{-1., 1.}], .01]

Infinity::indet: Indeterminate expression -Infinity + Infinity encountered.
General::stop: Further output of Infinity::indet
     will be suppressed during this calculation.
intervalnewton::rec: MaxRecursion exceeded.
Out[8]= Interval[{-1., 1.}]
```

6 Centered forms of functions

Mathematica can also do symbolic manipulation and although manipulating forms of expressions to reduce the excess width is not as trivial as the previous examples it is still quite easy. The code in this example is only 50 lines long.

Read in the file that contains the necessary code:

```
In[9]:= « IntervalTaylorForm.m
```

Find the centered sixth-order Taylor form for the function **Sin**. The expression defining how to evaluate it is hidden, although it can be displayed:

```
In[10]:= ff = IntervalTaylorForm[Sin[x], x, 6]

Out[10]= IntervalTaylorFunction[Sin[x], <>, 6]
```

This evaluates the previous result on the interval $[1, 2]$:

```
In[11]:= ff[Interval[{1,2}]]

                        5761     667 Cos[1]              385    667 Cos[1]
Out[11]= Interval[{-(-----) - ---------- + Sin[1], --- + ----------}]
                        46080       1280                 384       1280
```

This evaluates it on the same interval using Horner's rule rather than evaluation of powers of the symmetric interval:

```
In[12]:= ff[Interval[{1,2}], Horner -> True]

Out[12]= Interval[{
                          1      Cos[1]
                      -(----) - ------
              1          1440     120
            -(--) + ----------------
              24           2
            ------------------------- - Cos[1]
      1                2                ------
    -(-) + ------------------------------------
      2                   2                 6
    ------------------------------------------- - Cos[1]
                      2
    ----------------------------------------------------- + Sin[1],
                        2
                     1     Cos[1]
                    ---- + ------
             1      1440     120
             -- + ------------
             24         2              Cos[1]
             ---------------------- + ------
         1             2                 6
         - + -------------------------------
         2                2
         --------------------------------- + Cos[1]
                       2
    1 + ---------------------------------------------}]
                        2
```

In this example Horner's rule gives a wider interval:

```
In[13]:= N[{%%, %}]

Out[13]= {Interval[{0.434901, 1.28415}], Interval[{0.432297, 1.40917}]}
```

7 Conclusions

While Mathematica is not a complete interval analysis package it does provide many of the elements necessary for studying and experimenting with interval methods. Because it has a user base of many tens of thousands and because programs written in its language are completely portable, exchange of ideas is quite easy. With its graphic, symbolic, and numerical capabilities, Mathematica is an excellent way to introduce interval methods to students. To aid in the sharing of programs an automated repository has been established at the email address `mathsource@wri.com`. (For information send the single-line message `help info`.) For more ambitious projects for which Mathematica is too slow, one of the established interval analysis packages could be used for the computation while Mathematica serves as a more friendly interface, communication being done using the MathLink protocol. (Write `info@wri.com` or `mathlink@wri.com` for more information.)

References

[1] Rump, S. M. *Algorithms for verified inclusions—Theory and practice.* In: Moore, R. E. (ed.) "Reliability in Computing", Academic Press, San Diego, CA, 1988, pp. 109–126.

Wolfram Research, Inc.
100 Trade Center Drive
Champaign, IL 61820
USA
email: `keiper@wri.com`