Bisecting Floating Point Numbers

22 Feb 2014

Bisection is about the simplest algorithm there is for isolating a root of a continuous function:

1. Start with an interval such that the function takes on oppositely signed values on the endpoints.
2. Split the interval at its midpoint.
3. Recurse into whichever half has endpoints on which the function takes on oppositely signed values.

After each step, the new interval is half as large as the previous interval and still contains at least one zero (by the Intermediate Value Theorem).

I want to highlight a couple of interesting issues that arise when implementing bisection in floating point arithmetic that you might miss if you just looked at the definition of the algorithm.

In Julia code, a single step of bisection looks like this:

Julia treats floating point arithmetic the same way all modern programming environments do: according to the IEEE 754 standard. The examples here are in Julia.
because I plan to talk more about the language in the future, but everything in this post could as easily be written in any other language.

```plaintext
function bisect_step(fn, x1, x2)
    xm = (x1 + x2)/2

    # Return the sub-interval with oppositely-signed endpoints
    if sign(fn(x1)) != sign(fn(xm))
        return x1, xm
    else
        return xm, x2
    end
end
```

For example,
The first interesting question when implementing bisection is \textit{when should I stop bisecting}? In pure mathematics, you can think of carrying the process on indefinitely, but a computer program should halt.

Here's a little puzzle. I claim that one of the following functions always halts, and the other can loop forever. The functions differ only in their stopping criteria. Which one is which?

```julia
function bisect1(fn, x1, x2)
    @assert sign(fn(x1)) != sign(fn(x2))
    tol = 1e-13
    # Stop when function values are below
    # a set tolerance
    while abs(fn(x1)) > tol || abs(fn(x2))
        x1, x2 = bisect_step(x1, x2, fn)
    end
    return x1, x2
end
```

```julia
function bisect2(fn, x1, x2)
```

Bisecting Floating Point Numbers

http://squishythinking.com/2014/02/22/bisecting...
\begin{verbatim}
@assert sign(fn(x1)) != sign(fn(x2))
# Stop when the mean of the endpoints
# is equal to one of the endpoints
while x1 < (x1 + x2)/2 < x2
    x1, x2 = bisect_step(x1, x2, fn)
end
return x1, x2
end
\end{verbatim}

Let's try them out:

```
julia> bisect1(x -> 1000*\sin(x), 2.0, 4.1
# loops forever...
julia> bisect2(x -> 1000*\sin(x), 2.0, 4.1
(3.141592653589793, 3.1415926535897936)
```

This is the opposite of what would have happened if we ran these algorithms using mathematical real numbers instead of computer floating point numbers.

Over the reals, the first algorithm terminates for continuous functions by the definition of continuity, and the second algorithm doesn’t terminate because for any two non-equal real numbers $x_1 < x_2$ it’s always true that $x_1 < (x_1 + x_2)/2 < x_2$.

Over floating point doubles, the first algorithm doesn’t terminate because there is no floating point number $2.0 < x < 4.0$ such that $1000 \sin (x) < 10^{-13}$, and the second algorithm does terminate because for any finite floating point number $x_1$ (except the largest finite float),
there exists a strictly larger floating point number \( x_2 \) such that \((x_1 + x_2)/2 = x_1\) or \((x_1 + x_2)/2 = x_2\).

Both of these results might be surprising if you aren’t familiar with the details of floating point numbers. They arise due to the granularity of floats. There is a finite gap between any float and the next largest float. The size of the gap depends (proportionally) on the size of the number. In Julia, you can find the size of the gap using `eps`:

```
 julia> eps(1.0)
 2.220446049250313e-16
```

If \( x_1 \) is a float and \( x_2 \) is the next largest float, it is always true that their mean is either \( x_1 \) or \( x_2 \), because there are no other values between them.

For example,

```
 julia> let x=3.0, y=x+eps(x)
       (x + y)/2
    end
 3.0
```

The floating point representation of \( \pi \) is

```
 julia> fpi = convert(Float64, pi)
 3.141592653589793
```

Its sine is positive:
The next largest floating point number is

\[
\text{julia} > \text{fpi} + \text{eps}(\text{fpi}) = 3.1415926535897936
\]

and its sine is negative:

\[
\text{julia} > \text{fpi} + \text{eps}(\text{fpi}) = -3.216245299353273e-16
\]

Neither of these outputs is within $10^{-16}$ of 0.0, which is why bisect1 fails to terminate. On the other hand, bisect2 managed to find exactly these inputs as lower and upper bounds of a root of $\sin$. It didn’t need any explicit tolerances at all.

In this sense, bisect2 is an exemplary floating point algorithm. The answer to our present question, when should I stop bisecting?, is when there are no more floating point numbers between your lower and upper bound, whenever this is practical. Checking whether the mean of the endpoints is equal to one of the endpoints is a convenient way to check this condition.

Choosing a different arbitrary tolerance in a general purpose floating point algorithm is impolite. Absolute tolerances like the $1e-13$ in bisect1 are inappropriate in general purpose
algorithms because floating point numbers don’t come with units attached, so an algorithm with an absolute tolerance will behave differently depending on whether your user measures, e.g., lengths in millimeters or meters. Relative tolerances are better but fail when the answer is supposed to be 0.0. The spacing between floating point numbers cleanly elides these two limits, being relative for finite numbers, and finite for 0.0.

If you write your algorithms to compute to full precision, you save your users from having to think about tolerance conventions specific to your library. It can be tempting to think of floating point numbers as broken real numbers, but it’s much more productive to think of floating point numbers as a carefully thought out set of conventions for rounding the output of one algorithm to be appropriate as an input to another algorithm. Floating point numbers help with the hard work of making our programs composable.

Now that we know when to stop bisecting, the next interesting question is how many iterations will it take to bisect a root to full precision? As we’ve just discussed, floating point numbers are a finite set. There are $2^{64}$ floating point doubles (actually somewhat less because a whole group of them are NaN). Each step of bisection should halve the number of floats contained in the interval. This means it should always take less than 64 steps to reach full precision.

Let’s see some examples:
function count_bisect2_steps(fn, x1, x2)
    i = 0
    while x1 < (x1 + x2)/2 < x2
        x1, x2 = bisect_step(fn, x1, x2)
        i += 1
    end
    return i
end

# Isolate root at pi
julia> count_bisect2_steps(sin, 3.0, 4.0)
51  # good
# Isolate root at 0.0
julia> count_bisect2_steps(sin, -1.0, 1.0)
1075  # bad

What happened?

Earlier, I said “each step of bisection should halve the number of floats contained in the interval,” but as written, bisect_step doesn’t actually do this. The problem is that floats aren’t evenly distributed. They are much denser near 0.0 than far from it. This means that bisecting toward a root at 0.0 using the arithmetic mean eliminates fewer than half of the floats in the interval at each step.

Instead of bisecting the values of floating point numbers, what we really want to do is bisect a function that counts them. That would make it easy to eliminate exactly half of them at each step. Conveniently, the underlying binary representation of floating point numbers is
exactly a function that counts them. If we reinterpret the binary representation of a float as an integer, we can find the mean of the two integers that represent the endpoints, instead of the mean of the values of the two endpoints. Here's a function that does just that:

```julia
function _middle(x1::Float64, x2::Float64)
    # Use the usual float rules for combining non-finite numbers
    if !isfinite(x1) || !isfinite(x2)
        return x1 + x2
    end

    # Always return 0.0 when inputs have opposite signs
    if sign(x1) != sign(x2) && x1 != 0.0 &&
        return 0.0
    end

    negate = x1 < 0.0 || x2 < 0.0

    x1_int = reinterpret(UInt64, abs(x1))
    x2_int = reinterpret(UInt64, abs(x2))
    unsigned = reinterpret(Float64, (x1_int
    
    return negate ? -unsigned : unsigned
    end
```

There are some minor complications for negative numbers. The wikipedia page on the floating point standard has more information on exactly how the binary encoding of floats works. I also recommend “What every computer scientist should know about floating point numbers”. In
_middle, I avoid these complications by returning 0.0 for intervals with one negative endpoint and one positive endpoint, and by casting negative intervals to positive intervals and back. The >> 1 is right shift, a fast way to divide integers by 2.

Using this function, we can define our final version of bisect_root:

```python
function bisect_root(fn, x1, x2)
    xm = _middle(x1, x2)
    s1 = sign(fn(x1))
    s2 = sign(fn(x2))

    while x1 < xm < x2
        sm = sign(fn(xm))

        if s1 != sm
            x2 = xm
            s2 = sm
        else
            x1 = xm
            s1 = sm
        end

        xm = _middle(x1, x2)
    end

    return x1, x2
end
```

This version also inlines `bisect_step` in a way that avoids unnecessary reevaluations of the function. It has the nice property that it will
always converge to full precision in 65 function evaluations or fewer (two initial evaluations, and one more per step for up to 63 additional steps).

There are root bracketing algorithms that converge to full precision in fewer function evaluations for smooth functions (e.g. Brent’s method), but well-implemented bisection has the advantages of simplicity (fewer places for bugs to hide), and very low overhead, because there is so little arithmetic aside from the objective function evaluations.

I recently worked with John Verzani to get code very much like this final version of bisect_root into the Roots.jl package. As of this writing, the pull request is currently awaiting review.