

Practically Accurate Floating-Point Math

Neil Toronto and Jay McCarthy | Brigham Young University

With the right tools, floating-point code can be debugged like any other code, drastically improving its accuracy and reliability.

Most computing scientists know something about the dangers of treating floating-point numbers as if they're real numbers. We even know some folk wisdom intended to avoid the dangers. Test for nearness instead of equality. If the standard library has a function to compute a hypotenuse, use it instead of $\sqrt{x^2 + y^2}$. Use $(x - y) \cdot (x + y)$ instead of $x^2 - y^2$. Compute sums in order of decreasing magnitude—or is it *increasing* magnitude?

Sometimes we apply these rules habitually. Sometimes we apply them after tracing a reproducible error to an obvious problem. Sometimes we fire them at our code from a shotgun because it gives obviously wrong answers after many hours of apparently proper functioning. But most of the time, we can't tell whether this folk wisdom actually improves anything.

The typical prescription—error analysis—is time-consuming and requires expertise that most of us can't seem to find the time to obtain. We're apparently stuck between the two extremes of ignorance and certainty. Unfortunately, the initial and continuing costs of certainty often make ignorance more practical.

In almost every other software development task, testing and debugging is a practical middle ground. We gain knowledge and confidence about an algorithm's implementation by its behavior on representative inputs, using informal arguments about path coverage and similarity. With the right tools, testing and debugging floating-point code is no different, and can drastically improve its accuracy and reliability.

These tools and the principles behind them can be used for more than just fixing badly behaved floating-point code: they can also help us determine whether our research questions themselves are buggy. For concreteness, let's first concentrate on debugging the code that answers the questions, and then expand our scope.

You Could Have Invented Floating-Point Math

While easier than error analysis, floating-point debugging still requires some background knowledge. Therefore, we need to review what every computing scientist really needs to know about floating point. This includes

- how floats are distributed on the real line;
- how error is defined and how to measure it; and
- how errors originate and propagate.

For all of these, it helps to know something about how floating-point numbers are represented.

Floating-point numbers are little more than scientific notation encoded as bits. It's instructive, and not that difficult, to formalize the rules we've all learned for manipulating numbers in scientific notation by coding up floating-point math from scratch. Fortunately, we can ignore all those capricious rules about significant figures.

Any real number x that can be put in scientific notation can also be put in the form

$$x = s \cdot m \cdot 10^e, \quad (1)$$

where s is -1 or 1 , m is a natural number, and e is an integer. For example, to put $-1.23456789 \cdot 10^{-10}$ in this form, extract the sign, move the decimal point 8 places to the right, and subtract 8 from the exponent to get $-1 \cdot 123456789 \cdot 10^{-18}$.

Because computers operate more efficiently on powers of two than on powers of 10, floating-point numbers represent real numbers that can be put in the form

$$x = s \cdot m \cdot 2^e, \quad (2)$$

where $s \in \{-1, 1\}$, $m \in \mathbb{N}$, and $e \in \mathbb{Z}$, as before. In a float, s is called the *sign*, m is called the *significand* (an old-school and somewhat confusing synonym is *mantissa*), and e is called the *exponent*.

To code up floats in the Racket programming language (see the sidebar on Racket and its design),¹ we first define a new `struct` type to represent a float:

```
#lang typed/racket

(struct float ([sign : Integer]
              [significand : Natural]
              [exponent : Integer]))
#:transparent)
```

The first line of our program, `#lang typed/racket`, tells Racket that we're using the typed dialect. The next few lines define the `struct` type for floats, containing three fields with types `Integer`, `Natural`, and `Integer`. (We could have used `[sign : (U -1 1)]` to tell Typed Racket that the sign can be only -1 or 1 , but it would make working with the `sign` field more difficult.) It's important to note that `Natural` and `Integer` instances aren't limited to the system's word size or to any other maximum size at all: they're *big integers*. The `#:transparent` flag tells Racket, among other things, to not hide a printed `float`'s fields.

Using $x = s \cdot m \cdot 2^e$ as a guide, we define a function to convert `floats` to exact rationals:

```
(: rat (-> float Exact-Rational))
(define (rat x)
  (match-define (float s m e) x)
  (* s m (expt 2 e)))
```

The first line declares that the function `rat` takes arguments of type `float` and returns `Exact-Rational` instances, which are represented internally by reduced quotients of big integers. The line `(match-define (float s m e) x)` unpacks argument `x`'s fields. The next line, `(* s m (expt 2 e))` simply encodes the formula $s \cdot m \cdot 2^e$ in Racket.

If we run the program we have so far in the DrRacket IDE, an interactions window opens with a friendly welcome and a "`>`" prompt. We should try a few test cases.

One way to represent 80 as a float is $1 \cdot 10 \cdot 2^3$. In the interactions window, we write `(float 1 10 3)`, press Enter, and see

```
> (float 1 10 3)
(float 1 10 3)
```

DrRacket has printed the results of applying the `float` function to arguments 1, 10, and 3: a `float` with fields 1,

The Racket Programming Language

Racket's syntax (see www.racket-lang.org) is similar to Lisp's, so expect nested parentheses and `(notation prefix)`. But it goes far beyond Lisp in its ability to define new languages as libraries. One new language that ships with Racket is Typed Racket, in which all the code in this article is written.

Racket and its libraries are open source (Lesser General Public License, or LGPL). They've been under development for 20 years by PLT, a group of programming language researchers who put correctness first but aim for practical applicability. As a result, Racket's libraries are broad and reliable, and its runtime performance is good. In particular, Racket's floating point is both standards-compliant and fast: within a factor of 3 of C's speed when carefully tuned, which Typed Racket accelerates to within a factor of 1.5.

Racket ships with an integrated development environment (IDE) called DrRacket. Originally developed for teaching programming, it's unobtrusive and useful for both students and experts. DrRacket provides programmers with abilities that most IDEs don't, such as the ability to manipulate pictures and store them in programs as literal values, as easily as if they were simply numbers.

10, and 3. So far, so good. Converting it to an exact rational works as expected:

```
> (rat (float 1 10 3))
80
```

Another correct test case is as follows:

```
> (rat (float -1 23 -6))
- 26
  34.
```

That is, $-1 \cdot 23 \cdot 2^{-6} = -\frac{23}{64}$.

Multiplying `float` instances is easy, using the properties of multiplication and the identity $2^{e_1} \cdot 2^{e_2} = 2^{e_1+e_2}$. If $x_1 = s_1 \cdot m_1 \cdot 2^{e_1}$ and $x_2 = s_2 \cdot m_2 \cdot 2^{e_2}$, then

$$\begin{aligned} x_1 \cdot x_2 &= (s_1 \cdot m_1 \cdot 2^{e_1}) \cdot (s_2 \cdot m_2 \cdot 2^{e_2}) \\ &= (s_1 \cdot s_2) \cdot (m_1 \cdot m_2) \cdot 2^{e_1+e_2}. \end{aligned} \tag{3}$$

```
(: add (-> float float float))
;; Returns the exact sum of two floats
(define (add x1 x2)
  (match-define (float s1 m1 e1) x1)
  (match-define (float s2 m2 e2) x2)
  (define d (- e1 e2))
  (define n
    (if (< d 0)
        (+ (* s1 m1) (* s2 m2 (expt 2 (- d)))))
        (+ (* s2 m2) (* s1 m1 (expt 2 d)))))

  (float (if (< n 0) -1 1) (abs n) (min e1 e2)))
```

Figure 1. An implementation of exact floating-point addition.

Using the last formula as a guide, we define

```
(: mul (-> float float float))
(define (mul x1 x2)
  (match-define (float s1 m1 e1) x1)
  (match-define (float s2 m2 e2) x2)
  (float (* s1 s2) (* m1 m2) (+ e1 e2)))
```

Checking it against Racket's exact rational arithmetic, we get

```
>(* 80 - 23/64)
-28 3
4
>(rat (mul (float 1 10 3)
            (float -1 23 -6)))
-28 3
4
```

as expected.

Implementing addition is more involved, requiring shifting the significand of one float to line up its “decimal point” with the other float's, but it's certainly within the reach of anyone reading this article. Figure 1 gives a Typed Racket implementation.

Floats on the Number Line

Instances of `float` are different from typical floating-point numbers in one crucial way: because a `float`'s significand and exponent are big integers, there's no bound on their size. While this might seem like an improvement, it's actually kind of a curse.

Multiplication can double the number of bits required to store the significand. For example, multiplying two floats with 4-bit significands can result in a float with an 8-bit significand:

```
> (mul (float 1 15 0) (float 1 15 0))
(float 1 255 0)
```

Addition is worse: the number of bits required to store the result's significand depends on the difference in the arguments' exponents. For example, the following sum has $1 + 16 - (-16) = 33$ -bit significand, computed from two 1-bit significands:

```
> (add (float 1 1 16) (float 1 1 -16))
(float 1 4294967297 -16)
```

Division and irrational functions like square root are even worse: most results require infinitely many bits. We clearly need to limit the significand's range.

An unbounded exponent is also a curse. For example, it would take billions of terabytes to store just the exponent field of $\exp(\exp(\exp(100)))$.

At this point, we could review how floats are packed into bit fields to be stored in memory. But it takes a lot of work to turn those details into intuition, which we could get just as easily by visualizing the floats on the number line. An exhaustive visualization is infeasible for standard float sizes—billions of floats can be packed into the smallest standard size, 32 bits—so let's take a look at 6-bit floats.

The 6-bit floats we'll use have one sign bit to represent $s \in \{-1, 1\}$, two significand bits to represent $m \in [4..7]$, and three exponent bits to represent $e \in [-4..1]$. They're encoded as bits in a way that gives them the same properties as standard 32-, 64-, and 128-bit floats. Figure 2 shows their positions on the extended real line, except those that encode the error value not-a-number (NaN).

Because there are finitely many 6-bit floats, they aren't infinitely dense like rational numbers or instances of `float`. Consequently,

- every float has at least one closest neighbor;
- floats can be enumerated in their natural order; and
- every extended real number is either a float or is between two floats.

As an example of the last consequence, 19 is between 14 and ∞ . Also, π is between 3 and 3.5, but is closer to 3, so with 6-bit floats, $\pi = 3$. Many struggling math students, and some historic lawmaking bodies, would heartily approve.

To keep the numeric labels from overlapping, Figure 2 has to “zoom in” to the floats near zero, twice. Because there are the same number of floats between every two neighboring powers of two, floats are quite dense near zero and quite sparse away from it. This has somewhat surprising consequences. For example, every 6-bit float above $2^2 = 4$ is an integer, and every 6-bit float above $2^3 = 8$ is even. Likewise, every 64-bit float above 2^{52} is an integer, and every 64-bit float above 2^{53} is even.

Floats that can be written $s \cdot m \cdot 2^e$, where (for 6-bit floats) $s \in \{-1, 1\}$, $m \in [4..7]$, and $e \in [-4..1]$, are called *normal*. The

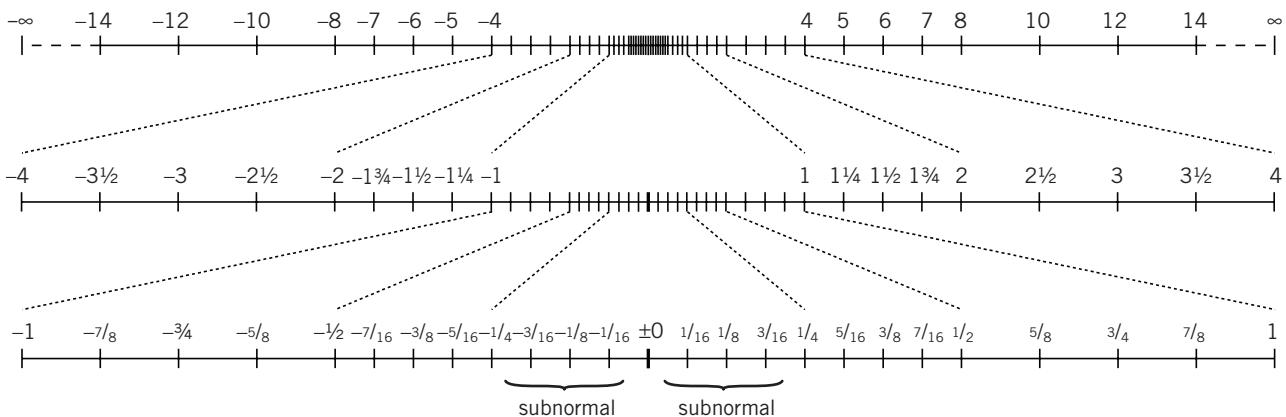


Figure 2. The positions of all 58 non-not-a-number (NaN), 6-bit floats on the extended real line.

strange range of m helps ensure that the encoding of non-NaN floats is unique. If we allowed $m = 0$, then 0 could be represented by both $1 \cdot 0 \cdot 2^0$ and $1 \cdot 0 \cdot 2^1$, among others.

But if so, 0 can't be a normal float. One exponent field bit pattern is thus reserved for specially encoding 0 and the next three uniformly spaced floats, which are called *subnormal*. For standard 64-bit floats, there are $2^{52} - 1$ positive subnormals, uniformly spaced in the open interval $(0, 2^{-1,022})$.

Another exponent field bit pattern is reserved for specially encoding $-\infty$, ∞ , and NaN.

There are two floating-point zeros. It's occasionally useful to distinguish their signs, but we'll regard them as equal.

One fact Figure 2 can't show is that 6-bit and standard-size floats are cleverly encoded to make it easy to enumerate them in their natural order. In particular, the encoding of any non-NaN float x , interpreted as a *signed magnitude* integer, or an integer whose most significant bit determines whether it's positive or negative, is the signed index of x on the number line. For example, the 6-bit encoding of 1 is 001100_2 , or 12. The encoding of -1 is 101100_2 , which, as a 6-bit signed magnitude integer, is -12 .

We'll define $\text{ord}(x)$ to be a float x 's signed ordinal index, so that $\text{ord}(1) = 12$ and $\text{ord}(-1) = -12$ for 6-bit floats. Figure 3 gives C implementations of ord and its inverse ord^{-1} for 64-bit floats. Racket's 64-bit implementations of ord and ord^{-1} are provided by its math library as `f1onum->ordinal` and `ordinal->f1onum`.

We care about the ordinal indexes of floats because they enable us to measure error and to design effective randomized testing strategies. For now, let's focus on measuring error.

Measuring Floating-Point Error

Functions that return approximations can at best be *close* to correct, so we need a natural way to measure the amount of error in their outputs. For a float x that approximates a real number r , the most obvious error measure is the distance

```
int64_t double_to_ordinal(double x) {
    double y = fabs(x);
    int64_t n = *((int64_t*)(&y));
    return x <= 0.0 ? -n : n;
}

double ordinal_to_double(int64_t n) {
    int64_t m = n < 0 ? -n : n;
    double x = *((double*)(&m));
    return n < 0 ? -x : x;
}
```

Figure 3. C implementations of $\text{ord}(x)$ and $\text{ord}^{-1}(x)$ for 64-bit floats. This code should work correctly on any system that stores floats with the same endianness as integers.

MPFR

The Multiple-Precision Float with correct Rounding (MPFR; www.mpfr.org) library is an open source project whose goal is to provide well-defined, standards-compliant, efficient, multi-precision floating-point functions. Its contributors take great pains to prove that the algorithms it uses always produce correctly rounded results.

between them, or $|x - r|$. This is called *absolute error*, and it's usually the wrong measure.

To see why, we'll first define $\text{fl}(r)$ to be the nearest float to a real number r . In case of a tie, $\text{fl}(r)$ is the nearest float with an even significand. If r is too large, $\text{fl}(r) = \pm\infty$.

IEEE 754-2008 Compliance

The first floating-point standard, IEEE 754-1985, was introduced to stem the tide of proliferating floating-point formats and implementations. It requires all basic functions to be *correctly rounded*—essentially, to be as accurate as possible. Consumer chip makers, who didn't have to support legacy floating-point units, embraced the standard with uncharacteristic zeal. The 2008 update addressed ambiguities and added a list of recommended basic functions.

As computing scientists, we should seek and emphatically support IEEE 754-2008 compliance. It makes our computations more accurate and our scientific results repeatable.

It seems most modern desktop hardware is mostly compliant, and system libraries are catching up. Unfortunately, many compilers still aggressively optimize floating-point code in a way that breaks compliance.

The following short Racket program tests hardware and system library compliance with IEEE 754-2008:

```
#lang racket
(require math/utils)
(test-floating-point 10000)
```

This program tests Racket's basic functions, as well as additional `math` library functions whose implementations are very sensitive to noncompliance, on thousands of deterministic and 10,000 random arguments, against MPFR. An automated process runs a similar program on a compliant system every time a Racket developer uploads changes to ensure Racket remains compliant.

According to reports from Racket users who have run the program on their own systems, it appears that

- arithmetic and square root are almost universally correctly rounded;
- on many systems, trigonometric functions are correctly rounded for arguments near zero, but return values with a few ulps error for very large arguments; and
- on a small handful of systems, $\exp(x)$ or trigonometric functions are accurate near zero but have error up to hundreds of ulps when x is large.

IEEE 754-2008 doesn't specify which floating-point functions are implemented in hardware, so it's difficult to make a general statement about whether the errors are caused by hardware floating-point units or system libraries.

With 64-bit floats, $|\text{fl}(\exp(100)) - \exp(100)| \approx 1.61 \cdot 10^{27}$, which is a ludicrous amount of error for the closest approximation of $\exp(100)$ —it's about 1.5 times the diameter of our galaxy in micrometers. The problem here is

the sparseness of 64-bit floats near $\exp(100)$: the floats on either side of it are $2^{92} \approx 4.95 \cdot 10^{27}$ apart!

The standard way to measure floating-point error relativizes absolute error by dividing it by the distance between neighboring floats near r :

$$\text{error}(x, r) = \frac{|x - r|}{\text{ulp}(\text{fl}(r))}. \quad (4)$$

Here, $\text{ulp}(y)$ is y 's *unit in last place*, which is defined as the magnitude of the lowest-order bit in y 's significand, or equivalently the distance from y to the next float from y away from zero. (If the next float from y is an infinity, $\text{ulp}(y)$ is the distance to the next float toward zero.) It's easy to compute $\text{ulp}(y)$ using `ord` and `ord-1`.

There's a nice intuitive interpretation of $\text{error}(x, r)$: it corresponds closely with the number of floating-point numbers from x to r . For example, with 6-bit floats,

$$\text{error}(8, 14) = \frac{|8 - 14|}{\text{ulp}(14)} = \frac{6}{2} = 3. \quad (5)$$

This interpretation admits *fractional* numbers of floats as well. For example,

$$\text{error}(\text{fl}(\exp(100)), \exp(100)) \approx \frac{1.61 \cdot 10^{27}}{2^{92}} \approx 0.325. \quad (6)$$

That is, $\text{fl}(\exp(100))$ and $\exp(100)$ are about 0.325 floats apart. Generally, if $\text{fl}(r)$ is finite, then $\text{error}(\text{fl}(r), r) \leq 0.5$, meaning that the nearest float to a real number r is no more than half a float, or half an ulp, away.

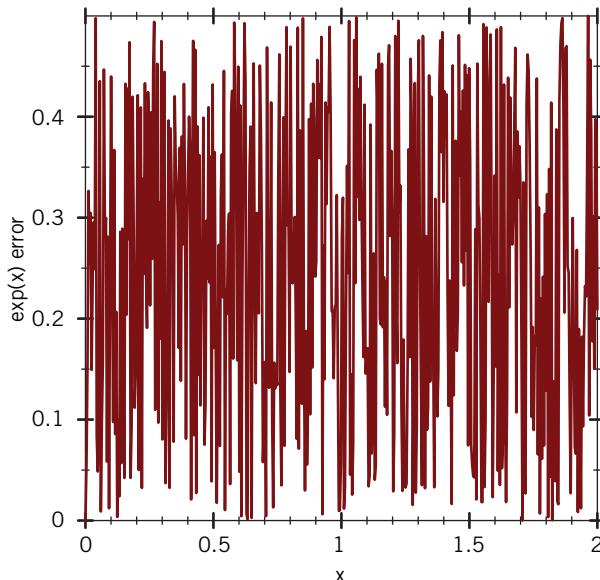
Racket's `math` library provides `ulp` and `error` as `fulp` and `fulp-error`. (Racket's 64-bit floating-point function names usually start with `f1`.) To use `fulp-error` to debug floating-point functions, we need correct values for its second argument. Racket's built-in exact rationals can represent correct values, at least for the results of arithmetic. For example,

```
> (require math)
> (fulp-error (/ 1.0 7.0) (/ 1 7))
0.2857142857142857
```

For irrational functions such as \exp , we can approximate correct values closely with high-precision floats. Racket's `math` library provides such floats by wrapping the C library MPFR,² the most accurate multiprecision floating-point library in existence (see the “MPFR” sidebar). For example,

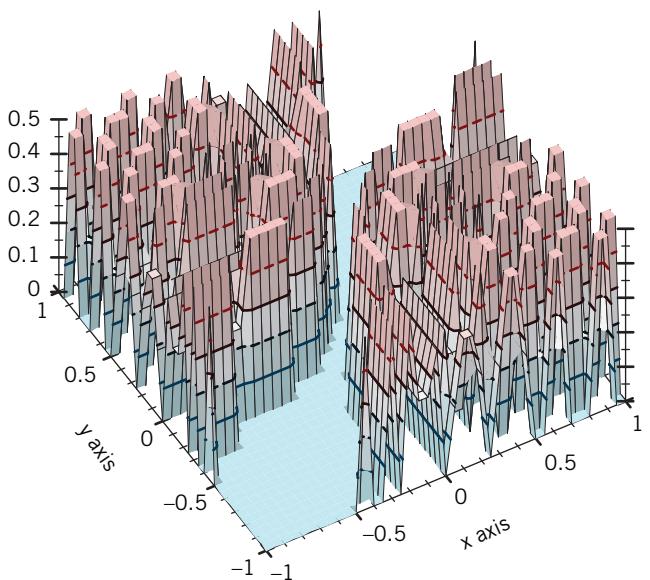
```
> (fulp-error (flexp 100.0)
              (bigfloat->real
               (bfexp (bf 100.0))))
0.32516258740803655
```

```
> (require plot/typed)
> (plot (function (fun-error flexp bfexp) 0 2)
  #:x-label "x"
  #:y-label "exp(x) error")
```



(a)

```
> (plot3d (contour-intervals3d
  (fun2d-error f1- bf-)
  -1 1 -1 1))
```



(b)

Figure 4. Error plots for basic functions that are correctly rounded in accordance with IEEE 754-2008. (a) Exponentiation and (b) subtraction.

The expression `(bf 100.0)` returns an MPFR floating-point number, or a *bigfloat*, which by default has a 128-bit significand. Then `bfexp` computes $\exp(100)$ with high precision, and `bigfloat->real` converts the result to an exact rational.

We'll frequently use bigfloat functions to estimate error in the outputs of 64-bit floating-point functions, so it's helpful to abstract away the details. We define

```
(: fun-error (-> (-> Flonum Flonum)
  (-> Bigfloat Bigfloat)
  (-> Real Flonum)))
(define ((fun-error f f*) r)
  (define x (f1 r))
  (flulp-error (f x)(bigfloat->real (f* (bf x)))))
```

Given a 64-bit floating-point function `f` and a bigfloat function `f*`, `fun-error` produces a function that accepts a real number `r` and returns an estimate of the error in `(f (f1 r))`, in ulps. To use it to measure the error in `flexp`, for example,

```
> ((fun-error flexp bfexp) 100)
0.32516258740803655
```

which is the same as before.

Now that we can get `flexp`'s error as a function, it's easy to plot. Figure 4a is a transcript from DrRacket's interactions window, plotting `(fun-error flexp bfexp)`. (In Racket, plots are values that are defined in a way that causes DrRacket to print them as graphics.) It appears that on this system, for arguments between 0 and 2, `(fun-error flexp bfexp)` never returns a value greater than 0.5. If so, `flexp` is accurate within half an ulp—as accurate as it can possibly be.

According to the IEEE 754-2008 floating-point standard,³ such accuracy is expected. Section 4.3 says

Except where stated otherwise, every operation shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded ...

More precisely, if `f` is a floating-point implementation of f , and `x` is the exact floating-point representation of a real value x , then `(f x)` must be the exact floating-point representation of $f(f(x))$. Operations that are computed this accurately, as mandated by IEEE 754-2008, are called *correctly rounded*.

Let's test another function. First, we'll define an abstraction similar to `fun-error` but for $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ functions:

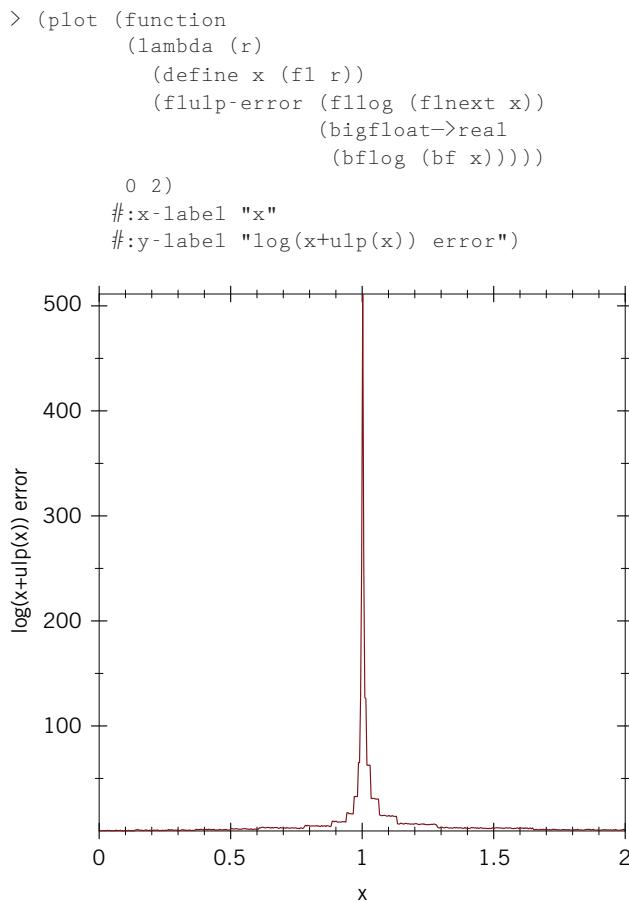


Figure 5. How $\log(x)$ magnifies error in x .

```
(: fun2d-error
  (-> (-> Flonum Flonum Flonum)
    (-> Bigfloat Bigfloat Bigfloat)
    (-> Real Real Flonum)))
(define ((fun2d-error f f*) rx ry)
  (define x (f1 rx))
  (define y (f1 ry))
  (flulp-error (f x y) (bigfloat->real
    (f* (bf x) (bf y)))))
```

The plot of `(fun2d-error f1- bf-)` in Figure 4b suggests subtraction is correctly rounded on this system. The trough in the center demonstrates an additional, useful fact: when x and y are within a factor of two of each other, or $1/2 \leq x/y \leq 2$, then $x - y$ is not only correctly rounded but is exact.

It would be nice if low error in basic functions' outputs meant that every floating-point formula had low error. If that were true, this article could end now with no more than an impassioned plea to hardware makers and compiler

writers to adhere to IEEE 754-2008 (see the related sidebar). Unfortunately, we're not so lucky.

Error Propagation

The most disappointing and difficult fact about floating-point error is that it isn't compositional: low error in a formula's subterms doesn't guarantee low error in the formula itself. One consequence is that correctly rounded functions are guaranteed to have low error only when their arguments are exact. We'll use this consequence to demonstrate the problem.

Suppose we give `f1log` an argument with just one ulp error. We can do that using `f1next`, which returns the next 64-bit float toward positive infinity. We then measure the error of its log:

```
> (flulp-error (f1log (f1next 1.1))
  (bigfloat->real
    (bflog (bf 1.1))))
14.572897331748424
```

This isn't terrible. 64-bit floats have 53-bit significands. Being 14.6 ulps off means that the $\log_2(14.6) \approx 3.87$ low-order significand bits are bad, so the result's 53-bit significand still has about 49 bits precision.

But really, how bad can it possibly get?

Unfortunately, there's effectively no bound on how bad it can get:

```
> (flulp-error (f1log (f1next 1.0001))
  (bigfloat->real
    (bflog (bf 1.0001))))
16381.87167545197
> (flulp-error (f1log (f1next 1.00000000001))
  (bigfloat->real
    (bflog (bf 1.00000000001))))
137438953470.5061
```

In the last example, we have only 16 bits precision. For perspective, this is just a little more precise than $22/7 \approx \pi$.

This sensitivity to argument error seems to get worse as the argument approaches 1. Let's plot how sensitive `log` is to error in arguments between 0 and 2 to find out if that's true. This time, we'll plot a lambda, or an anonymous function, instead of using a function like `fun-error` to create a function to plot.

Figure 5 contains the transcript from DrRacket's interactions window. The spike at $x = 1$ looks perfectly dreadful, but it's actually worse than it looks. If we were to zoom into it by plotting at ever-narrower intervals that contain 1, we'd find that the spike goes all the way up to infinity. There's no limit to how much `log` can magnify its argument's error.

There's a classic joke pertinent to our situation:

[Doctor arrives to find patient bending joint backwards, biting armpit, or other grotesque action.]

Patient: Doctor, it hurts when I do this.

Doctor: Then don't do that!

We can't demand that `f11og` be more accurate if it's already as accurate as it can possibly be. We can't apply `f11og` only to exact arguments because the outputs of other floating-point functions are almost always approximate. All we have left is, "Then don't do that!" Don't apply `f11og` to an approximate result that's near 1.

Is this normal? Unfortunately, yes. Most common real-valued functions, even when their floating-point implementations are correctly rounded, have subdomains on which they magnify argument error in their outputs. These places are often called *ill-conditioned*, but that term doesn't fill anyone with the right amount of dread. Instead, we'll call those subdomains *the badlands*.

High error in floating-point code isn't caused by rounding exact results to the nearest float. It's caused by wandering into the badlands with a rounded result.

Avoiding the Badlands

The main task in debugging floating-point math is figuring out when we've wandered into the badlands with an approximation and then rewriting parts of our code so we don't do that.

It's therefore essential to know where common functions' badlands are. Figures 6 and 7 have plots and formulas for the badlands of the most common floating-point functions. To derive them, we've made the definition of *badlands* more precise: they're the parts of the domain in which error is magnified about 4 or more times, without considering additional rounding error. In other words, the badlands reduce precision by two bits or more, as well as add rounding error.

Perhaps surprisingly, some common functions don't have badlands, such as multiplication and arctangent. We can confidently apply them to approximations.

Figure 6 identifies the badlands for $x - y$ as

$$\frac{1}{2} \leq x/y \leq 2. \quad (7)$$

In other words, the badlands are where x and y are within a factor of two of each other. This is the same subdomain on which floating-point subtraction is exact when x and y are exact, so whether subtraction in the badlands is prudent or deadly depends only on exactness. The universe is clearly messing with us, but at least it's made this important subdomain of $\mathbb{R} \times \mathbb{R}$ easier to remember.

Log-One-Pea? Exp-Em-One?

It might seem like a stretch to say $\text{log1p}(x) = \log(1 + x)$ and $\text{expm1}(x) = \exp(x) - 1$ are common functions. They should be, because they so often provide an escape from \log 's and \exp 's badlands. If the language you use for scientific computing doesn't have them, submit a bug report.

If $f: \mathbb{R} \rightarrow \mathbb{R}$ is differentiable, its badlands, as we've defined the term, can be estimated by solving

$$\left| \frac{x \cdot f'(x)}{f(x)} \right| \geq 4 \quad (8)$$

for x , where f' is the first derivative of f . The left-hand side is called f 's *condition number* at x . There are more general condition number formulas for multivariate functions, but they're more complicated than we need to get into.

We say the badlands can be "estimated" because the condition number formula doesn't account for any floating-point details. But the fact that such a simple formula can estimate how much f magnifies floating-point error is telling: it means that f 's badlands are primarily a property of f itself, and much less a property of its floating-point implementation.

We can use the condition number formula to get some intuition about how to avoid the badlands. When the denominator $f(x)$ is near zero and the numerator isn't, the condition number is large. We should therefore avoid zero crossings that aren't at the origin. A canonical example is $\log(1) = 0$. Another trait to avoid is exponential growth away from the origin, which makes the numerator large.

Figures 6 and 7 plot badlands that can be estimated using condition numbers, which estimate how error is magnified only for *normal* floats. Floating-point implementations of some functions, particularly x/y , \sqrt{x} , and $\log(x)$, additionally magnify error without bound when their arguments are approximate and *subnormal*.

A sometimes convenient, but complicating fact is that high error isn't compositional, either: high error in a formula's subterms doesn't guarantee high error in the formula itself. For example, applying `f11og` far enough from zero reduces error:

```
> (flulp-error (f11og (flstep 1e10 40))
  (bigfloat->real
   (bflog (bf 1e10))))  
2.11101355742103
```

Operation	Badlands
$x + y$	$\frac{1}{2} \leq -x/y \leq 2$
$x - y$	$\frac{1}{2} \leq x/y \leq 2$
$x \cdot y$	<i>None</i>
x/y	x or y subnormal
\sqrt{x}	x subnormal
$\log(x)$	$\exp(-\frac{1}{4}) \leq x \leq \exp(\frac{1}{4})$ or x subnormal
$\exp(x)$	$ x \geq 4$
$\log1p(x)$	$x \leq -\frac{15}{16}$
$\expml(x)$	$x \geq 4$

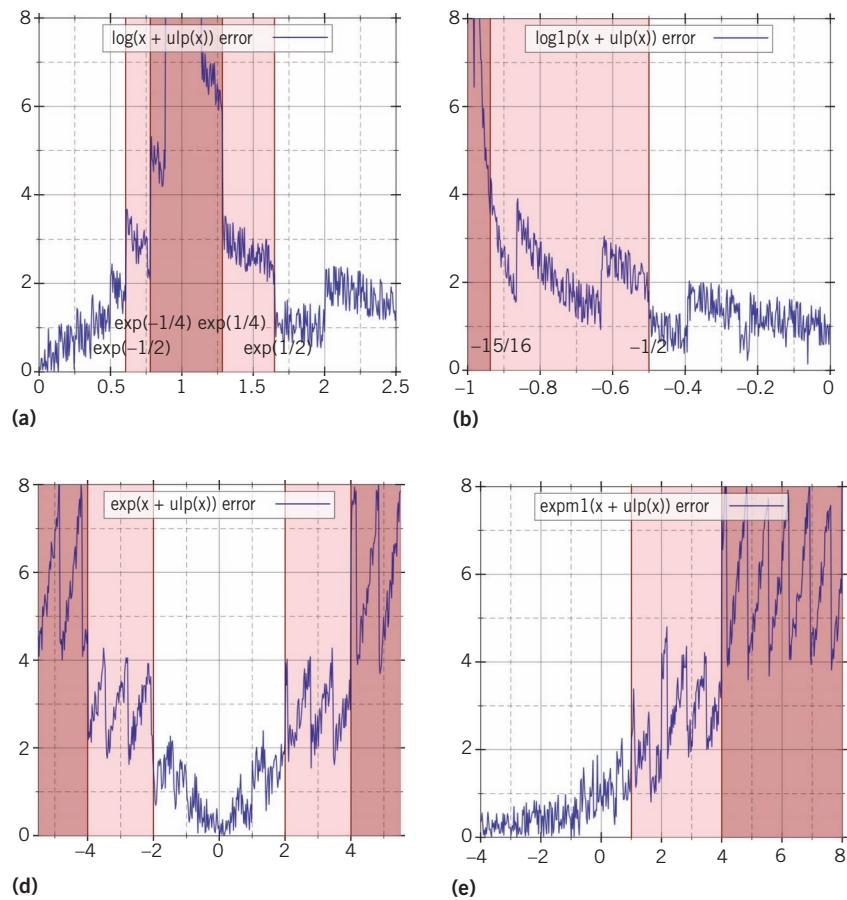


Figure 6. Formulas and plots of the badlands for common 64-bit floating-point functions.

Here, `(f1step 1e10 40)` is equivalent to 40 iterations of `f1next`, starting with `1e10`. Computing the log has reduced error from 40 ulps to 2 ulps.

It's occasionally useful to reformulate floating-point math to reduce error. We'll let error reduction happen by accident, and use testing to find out where it doesn't make up for wandering into the badlands.

Floating-Point Test-Debug Cycle

A good test-debug cycle for a floating-point function f proceeds as follows (also see the sidebar, "Testing and Debugging Support"). First, write a straightforward version f . Next, write a version f^* with higher precision. Then repeat the following steps as necessary:

1. Test f against f^* to find high-error subdomains, using only *exact* arguments to f .
2. Reformulate f to reduce error.
3. Make the same changes to f^* , if necessary.

We'll demonstrate this test-debug cycle on three functions, which will illustrate the answers to the following questions.

Q. Are you seriously advocating writing two versions of every floating-point function?

A. Yes. Admittedly, that answer isn't very motivating, so here's a counterquestion: How certain do you want to be that your good, bad, or unexpected results aren't an artifact of floating-point error?

Q. What kind of numbers should f^* use?

A. For arithmetic functions, use exact rationals if your language or libraries provide them. Otherwise, multiprecision floats with 128-bit significands seem to work well for testing most 64-bit floating-point functions.

Q. Why test f only with exact arguments?

A. How f magnifies error when given approximate arguments is a property of the function it implements. We can't do anything about it.

Q. Why make corresponding changes to f^* ?

A. Unless f^* is written using exact rational arithmetic, if f wanders into the badlands with an approximation, so does f^* .

Operation	Badlands
$\tan(x)$	$ x \geq \arctan(\sqrt{7})$
$\sin(x)$	$ x \geq \frac{5}{6}\pi$
$\cos(x)$	$ x \geq \pi$ or $\frac{1}{3}\pi \leq x \leq \frac{2}{3}\pi$
$\arctan(x)$	<i>None</i>
$\arcsin(x)$	$ x \geq \sqrt{\frac{63}{64}}$
$\arccos(x)$	$ x \geq \cos\left(\frac{1}{2}\right)$ or $x \leq -\sqrt{\frac{255}{256}}$

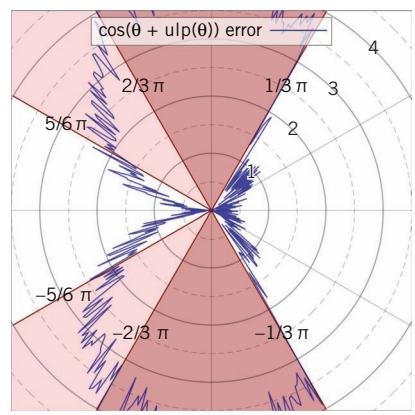
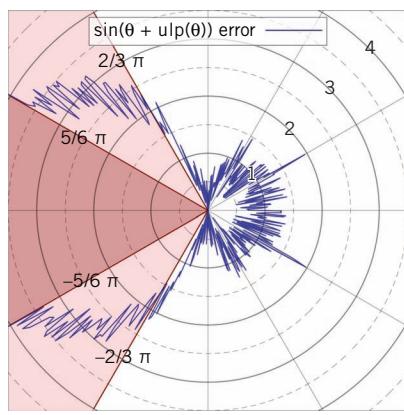
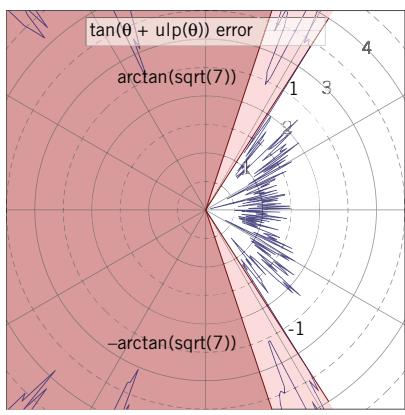
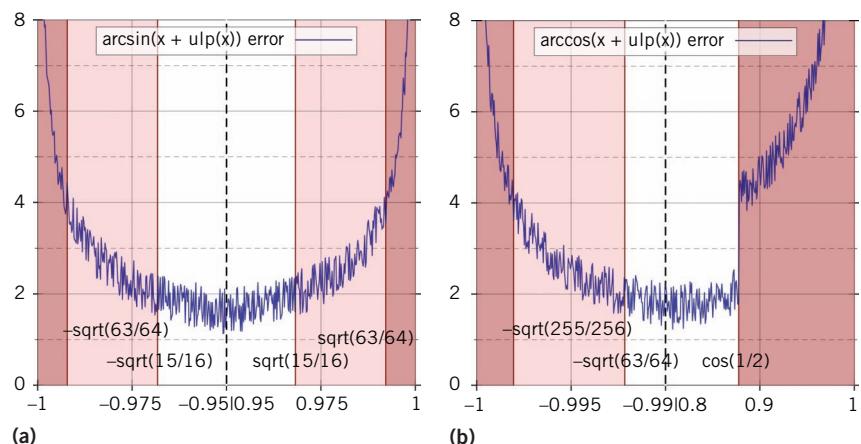


Figure 7. Formulas and plots of the badlands for trigonometric 64-bit floating-point functions.

Q. Why not just use f^* instead of f ?

A. It's almost certainly hundreds to thousands of times slower. Perhaps more importantly, the initial version of f^* , though more accurate than the initial version of f , might not be accurate enough.

One question without an easy answer is "What kind of accuracy should I aim for?" This depends strongly on how f will be used. Racket's `math` library generally aims for less than a few ulps error on the entire domain. But library code is used in many different, unpredictable ways, so such high standards make sense. Your project might tolerate 10,000 ulps error, or even millions, and almost certainly doesn't touch every floating-point function's entire domain.

Whatever your project is, we recommend reducing the subdomain on which error is unbounded, if not eliminating unbounded error altogether.

Debugging the Geometric Distribution

We'll start debugging with a real task from creating Racket's `math` library. The Geometric distribution's

Testing and Debugging Support

For testing and debugging floating-point code as described in this article, a language or library needs

- 64-bit implementations of `ord`, `ord-1`, and `ulp`;
- multiprecision floats (MPFR recommended);
- for each kind of high-precision number, implementations of `fl` (correctly rounded) and `error`;
- plotting on any subinterval of $(-\infty, \infty)$ without additional error, underflow, or overflow; and
- random float and 64-bit integer generation.

Other high-precision numbers, such as exact rationals and double-doubles, are a plus.

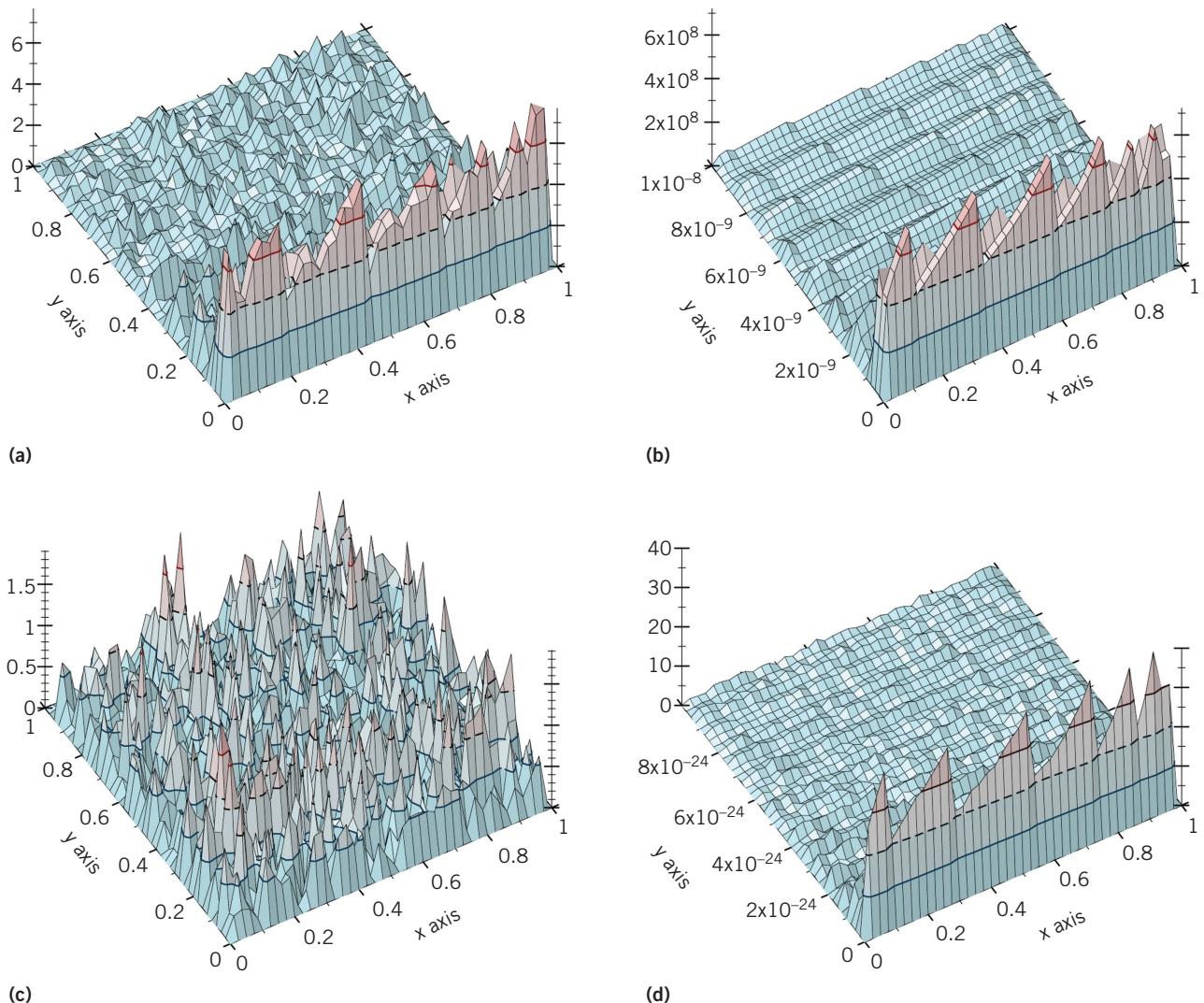


Figure 8. Plots of (`fun2d-error f f*`) at two debugging stages, in different domains. (a) Initial implementation, entire domain. (b) Initial implementation, approaching $y = 0$. (c) Implementation using `log1p`, entire domain. (d) Implementation using `log1p`, without updating the bigfloat implementation to use `log1p`, near $y = 2 \cdot 10^{-24}$.

implementation is partly defined in terms of $f: (0, 1) \times (0, 1) \rightarrow \mathbb{R}$, defined by

$$f(x, y) = \frac{\log(1-y)}{\log(x)}. \quad (9)$$

Following the test-debug procedure, we first define two straightforward implementations: f , for 64-bit floats, and f^* , for arbitrary-precision bigfloats:

```
(: f (-> Flonum Flonum Flonum))
(define (f x y)
  (/ (f1log (- 1.0 y))
    (f1log x)))
```

```
(: f* (-> Bigfloat Bigfloat Bigfloat))
(define (f* x y)
  (bf/ (bflog (bf- (bf 1.0) y))
        (bflog x)))
```

We start testing f by plotting (`fun2d-error f f*`) on the entire domain. Figure 8a shows the results, which aren't terrible so far. We apparently have error less than 4 ulps for $y > 0.1$ or so. The plot only suggests that error increases as y approaches zero.

But really, how bad could it possibly get?

There's effectively no bound on how bad it can get. Figure 8b plots the error for $y \leq 10^{-8}$. Instead of the maximum error at just under 8 ulps, in this plot, error is just

under $8 \cdot 10^8$ ulps. Further plots would show that it only gets worse closer to $y = 0$. Ugh.

Is this normal? Unfortunately, yes. It's utterly typical for a straightforward floating-point implementation of a real function to have low error on most of its domain and unbounded error on a small subdomain.

There must be some badlands-wandering going on. Knowing that f 's error is high when y is near 0.0, it's not hard to find the problem: if y is near 0.0, then $(-1.0/y)$ is approximate and near 1.0, which is in \log 's badlands, so $(\text{f1log } (-1.0/y))$ has high error.

What about the rest of f 's definition? The outer division doesn't magnify error when its arguments are normal. (We'll deal with subnormal arguments only if testing turns up problems.) For a different reason, $(\text{f1log } x)$ should be fine: x is assumed exact because it's a function argument, so it doesn't matter if x is in \log 's badlands.

To fix $(\text{f1log } (-1.0/y))$, we need a log-like function with different badlands, such as $\text{log1p}(x) = \log(1 + x)$, whose badlands are shown in Figure 6. Replacing $\log(1 - y)$ in the definition of the real function f with $\text{log1p}(-y)$ wouldn't change f . We could similarly replace $(\text{f1log } (-1.0/y))$ in the definition of the floating-point function f with $(\text{f1log1p } (-y))$, but to change f in a way that changes only its floating-point error.

Using f1log1p will definitely reduce error: y is exact and negation is exact, so even if $(-y)$ is in log1p 's badlands, it won't matter. So we change f 's definition to

```
(: f (→ Flonum Flonum Flonum))
(define (f x y)
  (/ (f1log1p (- y))
    (f1log x)))
```

and plot its error. Figure 8c shows the result. It looks great—error is apparently bounded by 2 ulps—but we're not done.

We still need to update f^* . If $(\text{bf}- (\text{bf } 1.0) y)$ is in \log 's badlands, then $(\text{bflog } (\text{bf}- (\text{bf } 1.0) y))$ at least quadruples error, even with bigfloats. Because the amount that error can be magnified is unbounded, at some point, \log 's badlands could make f^* less accurate than f ! In this case, it happens near $2e-24$, as shown in Figure 8d.

Instead of updating f^* , we could increase the number of bits in the bigfloats that f^* computes with. In this case, it takes bigfloats with 1,074-bit significands to make f^* more accurate than f again! Clearly, using more precision isn't always the best approach. Changing f^* to avoid \log 's badlands the same way f does is more reliable and results in faster high-precision code, which means faster and more reliable testing.

Whether by using more significand bits or changing f^* , we would find through more testing that our updated implementation of f has less than 2 ulps error everywhere.

Debugging Log of Quotient

The function

$$h(x, y) = \log(x/y) \quad (10)$$

comes up often when dealing with logarithms, and has an annoyingly large subdomain for which a straightforward implementation has high error. Accurate implementation requires a common technique we didn't use in the last example: splitting up the domain. For simplicity, we'll restrict h to the first quadrant; that is, $h : [0, \infty) \times (0, \infty) \rightarrow \mathbb{R}$.

Figure 9a plots the error in the straightforward 64-bit float implementation h against the bigfloat implementation h^* . The ridge in the middle suggests that error gets up to at least 30 ulps near the diagonal $x = y$.

But really, how bad could it possibly get?

This time, we won't dignify that silly question with an answer.

The problem can't be division, which just adds rounding error. The problem must be applying floating-point \log to the rounded quotient. According to Figure 6, we can avoid \log 's badlands by not applying it if $\exp(-1/4) \leq x/y \leq \exp(1/4)$, but we'll need a reformulation on that subdomain.

An obvious reformulation to try is $\log(x/y) = \log(x) - \log(y)$, but this is doomed to fail in the same way. The error plot looks similar, but has a fatter diagonal ridge. Because $(\text{f1log } x)$ and $(\text{f1log } y)$ are approximate, subtracting them results in high error when they're within a factor of two, which happens when x and y are near the diagonal.

Again, we'll turn to log1p . We need to compute

$$\begin{aligned} \log(x/y) &= \log(1 + x/y - 1) \\ &= \text{log1p}(x/y - 1) \\ &= \text{log1p}\left(\frac{x-y}{y}\right). \end{aligned} \quad (11)$$

At this point, we could find the conditions under which $(x - y)/y$ is in log1p 's badlands and determine whether they intersect $\exp(-1/4) \leq x/y \leq \exp(1/4)$. Instead, we'll just code it up:

```
(: h (→ Flonum Flonum Flonum))
(define (h x y)
  (define q (/ x y))
  (cond [(<= (flexp -0.25) q (flexp 0.25))
         (f1log1p (/ (- x y) y))]
        [else
         (f1log q)]))
```

Then we'll test it, as shown in Figure 9b. We apparently have error bounded by about 2 ulps. Of course, we should test it more thoroughly than we can demonstrate in this article.

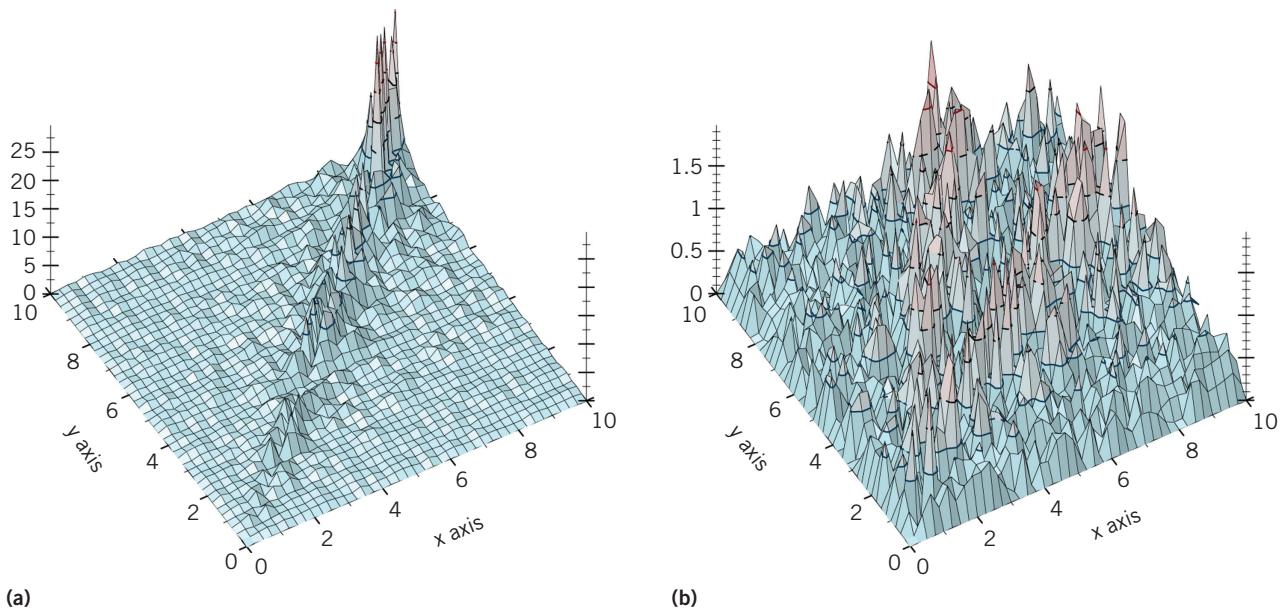


Figure 9. Plots of `((fun2d-error h h*)` before and after debugging. (a) Initial implementation and (b) debugged implementation.

We're not done debugging `h` yet, but we stress that, at this point, it's good enough for most purposes. It's also much more accurate than it was before, with error bounded by about 2 ulps near the diagonal instead of unbounded. It's still fairly simple.

But suppose we're writing library code, or `h`'s callers need it to be extremely solid, or someone using `h` has reported there's occasionally a problem with it. Let's see how robust we can make `h` using just testing and debugging.

The first problem we'll find is that it's hard to find a problem, at least by generating plots. We need a way to create unusual arguments. For this, we turn to random sampling. One effective way to generate unusual combinations of floats is to generate uniformly random ordinal indexes and convert them to floats using `ord-1`. A Typed Racket generator that chooses floats this way between `a` and `b` (excluding `b`) is

```
(: random-flonum (-> Flonum Flonum Flonum))
;; Returns a random flonum in [a,b)
(define (random-flonum a b)
  (ordinal->flonum
    (random-integer (flonum->ordinal a)
      (flonum->ordinal b))))
```

We can now use `(random-flonum -max.0 +inf.0)` to sample from all finite 64-bit floats, where `-max.0` is defined by the `math` library as the largest negative float (approximately $-1.8 \cdot 10^{308}$) and `+inf.0` is the next float

after `+max.0`. This tends to return very large and very small numbers because the exponent field is uniformly distributed.

It takes only a few random samples to find an `x` and `y` for which `((fun2d-error h h*) x y)` is infinite. We'll define them temporarily in the interactions window to keep from having to retype them:

```
> (define x 5.295588382145396e+229)
> (define y 7.503806844818842e-122)
```

We get infinite error because `(h x y)` is infinite but `(h* x y)` isn't:

```
> (h x y)
+inf.0
> (bigfloat->flonum
  (h* (bf x) (bf y)))
807.858831263043
```

Manually computing `(h x y)` reveals what happened:

```
> (/ x y)
+inf.0
> (f1log +inf.0)
+inf.0
```

The exact quotient of `x` and `y` is too large to be approximated by a float. We say that `(/ x y)` *overflows*.

Similarly, when an exact result is too close to zero, it *underflows* to 0.0 or -0.0. That happens, for instance, if we swap `x` and `y`:

```
> (/ y x)
0.0
```

Fortunately, the reformulation $\log(x/y) = \log(x) - \log(y)$ seems to work with low error for these arguments, and would keep `h` from prematurely overflowing or underflowing:

```
> (- (f1log x) (f1log y))
807.8588312630429
```

Randomized testing finds another, less frequent problem: when $(/ x y)$ is subnormal, $(f1log (/ x y))$ has high error. Using $(- (f1log x) (f1log y))$ may solve this problem as well because it doesn't use division. More testing should reveal whether $(f1log x)$ and $(f1log y)$ are in subtraction's badlands.

The definition of `h` is now

```
(: h (-> Flonum Flonum Flonum))
(define (h x y)
  (define q (/ x y))
  (cond [(<= (flexp -0.25) q (flexp 0.25))
         (f1log1p (/ (- x y) y))]
        [(or (<= q +max-subnormal.0)
             (= q +inf.0))
         (- (f1log x) (f1log y))]
        [else
         (f1log q)])))
```

where `+max-subnormal.0` is the largest subnormal 64-bit float, or the float just before 2^{-1022} . Racket takes about a minute to apply this version of `h` to 1 million random argument pairs and compare its outputs to the high-precision outputs of `h*`. The largest reported error is less than 2.1 ulps.

When updating `h*` to match `h`, it's not necessary to test whether `q` overflows or underflows. MPFR's multiprecision floats can have much larger exponents than 64-bit floats, so they don't overflow or underflow as easily.

Debugging 3D Dot Product

What if we can't find an algebraic trick to get out of the badlands? When this happens, sometimes the only good answer is to use more precision. Of course, we can use the high-precision implementations we use for testing, but they're usually quite slow. Fortunately, on systems with correctly rounded arithmetic—which is the lowest possible requirement we can think of for a scientific computing platform—there are faster high-precision options.

Extended-Precision Floats

Some languages provide access to the hardware's 80-bit floats, which system libraries use to implement 64-bit functions such as $\text{pow}(x, y) = x^y$ accurately in software. While an attractive alternative to other higher-precision numbers, 80-bit floats usually don't seem to have enough precision to make up for wandering into the badlands.

Some compilers silently replace intermediate 64-bit results with 80-bit results, which isn't compliant with IEEE 754-2008. In practical terms, these compilers break important error-reducing 64-bit algorithms such as Kahan summation and double-double arithmetic, and ruin portability and scientific repeatability. If the language you use for scientific computing does this, submit a bug report.

We'll debug the dot product to demonstrate some practically accurate, efficient options. We'll use the definition

$$\begin{aligned} \text{dot}(x_1, y_1, z_1, x_2, y_2, z_2) &= (12) \\ &= x_1 \cdot x_2 + y_1 \cdot y_2 + z_1 \cdot z_2. \end{aligned}$$

This might seem perfectly innocuous, but accurate implementation is beastly. The first problem is that if the products $x_1 \cdot x_2$, $y_1 \cdot y_2$ and $z_1 \cdot z_2$ don't all have the same sign, the implementation can wander into addition's badlands. The second problem is premature overflow. For example, even if $\text{fl}(x_1 \cdot x_2 + y_1 \cdot y_2) < \infty$, we still might get $\text{fl}(x_1 \cdot x_2) = \infty$ as an intermediate result, if $y_1 \cdot y_2$ is about as large as $x_1 \cdot x_2$ and has the opposite sign.

As before, we start by coding up straightforward implementations `dot` and `dot*`. For `dot*`, we can use exact rational arithmetic instead of bigfloats. If we do, we won't have to update `dot*` to avoid any badlands.

We can't plot error on a domain with six dimensions, so we'll have to settle for random sampling. To avoid dealing with overflow at first, we'll sample between -1 and 1. We'll have our generator flip a coin to determine whether to sample uniformly from the floats in [-1, 1] to represent typical arguments or to sample uniformly from the *encodings* of floats in [-1, 1] to represent atypical arguments:

```
(: random-dot-flonum (-> Flonum))
(define (random-dot-flonum)
  (if (< (random) 0.5)
      (random-flonum -1.0 (flnext 1.0))
      (* 2.0 (- (random) 0.5))))
```

Using `random-dot-flonum` to generate arguments for `dot` and `dot*`, we occasionally get errors in the thousands of ulps. By coding up the addition badlands formula in Figure 6,

About That Folk Wisdom

Looking to use some floating-point folk wisdom in debugging your project? We now have the tools to evaluate it.

Test nearness instead of equality. Wise, if done right. *Floating-point epsilon* is defined as $\varepsilon = \text{ulp}(1)$, the distance between 1 and the next float. If x is a normal float, then $|x| \cdot \varepsilon$ is between $\text{ulp}(x)$ and $2 \cdot \text{ulp}(x)$. If $|x - y| \leq |x| \cdot (4 \cdot \varepsilon)$, then y is within 4 to 8 ulps of x .

Use the library's hypotenuse function. Often wise. A library function should avoid premature overflow from squaring, and may be correctly rounded.

Use $(x - y) \cdot (x + y)$ instead of $x^2 - y^2$. Wise. Half the time, $x^2 - y^2$ wanders into subtraction's badlands with two approximations. But $(x - y) \cdot (x + y)$ multiplies approximations, which doesn't magnify their error.

Compute sums in some specified order or grouping. Not usually wise. No ordering or grouping strategy is best in all cases. Use Kahan summation (http://en.wikipedia.org/wiki/Kahan_summation_algorithm), double-doubles, or a specialized summation algorithm.¹

Reference

- J. Demmel and Y. Hida, "Accurate and Efficient Floating Point Summation," *SIAM J. Scientific Computing*, vol. 25, no. 4, 2003, pp. 1214–1248.

we start writing a version of dot that avoids wandering into addition's badlands with a rounded result:

```
(define (dot x1 y1 z1 x2 y2 z2)
  (define x (* x1 x2))
  (define y (* y1 y2))
  (cond [(<= 0.5 (- (/ x y)) 2.0)
         <compute-dot-somehow>]
        [else
         (define d (+ x y))
         (define z (* z1 z2))
         (if (<= 0.5 (- (/ d z)) -2.0)
             <compute-dot-somehow>
             (+ d z)))]))
```

We could replace `<compute-dot-somehow>` with code that converts the arguments to exact rationals, applies `dot*` to them, and converts the exact result to a float. This works, and testing the updated `dot` suggests that its error is bounded by about 2.5 ulps. This is great, but it's very slow: about 40 times slower than the initial `dot` on random arguments. If we

try using bigfloats instead of exact rationals, we'll find that the resulting code is hardly faster.

If we could somehow get *exact* multiplication out of 64-bit floats, we'd be set. It turns out that there's a way to do that, using two 64-bit floats to represent one real number. These are often called *double-doubles*. Without going into details, a double-double is a pair of 64-bit floats, that each have a 53-bit significand, whose exact sum has 107 bits precision. Operations on double-doubles work correctly only on systems with correctly rounded arithmetic.

(Of course, $53 + 53 = 106$. The second float's sign bit carries information equivalent to one significand bit.)

Racket's `math` library exports functions that operate on double-doubles. Those that only *return* double-doubles have the suffix `/error` (read "with error"). For example,

```
> (f1*/error 1.1 2.1)
2.3100000000000005
-2.1316282072803005e-16
```

The *exact* sum (not the *floating-point* sum) of these returned float values is the exact product of 1.1 and 2.1. Alternatively, the first float value is an approximation, and the second is its signed rounding error.

Math library functions that accept double-doubles as arguments have the prefix `f12`; for example, `f12+` accepts four arguments and returns two results. We can use `f12+` and `f1*/error` to define a high-precision, but reasonably fast, version of `dot`, as follows:

```
(define (slowish-dot x1 y1 z1 x2 y2 z2)
  (define-values (xhi xlo) (f1*/error x1 x2))
  (define-values (yhi ylo) (f1*/error y1 y2))
  (define-values (zhi zlo) (f1*/error z1 z2))
  (define-values (dhi dlo) (f12+ xhi xlo yhi ylo))
  (define-values (ehi elo) (f12+ dhi dlo zhi zlo))
  ehi)
```

Replacing `<compute-dot-somehow>` with `(slowish-dot x1 y1 z1 x2 y2 z2)` in the definition of `dot` yields an implementation that's only about four times slower than the initial implementation on random arguments, with error again bounded by about 2.5 ulps.

On the system we use for testing, the updated `dot` computes 25 million dot products per second instead of the initial implementation's 100 million per second. This might seem like a bad deal, but there are other considerations. First, in scientific computing, we should worry more about our programs being faithful to our models than about computation speed. Second, so much other code usually surrounds floating-point computations—for example, code that shuffles data around, indexes arrays, or decides what to compute—that taking even a fourfold speed hit hardly matters.

If we're willing to tolerate more error, it's possible to get good performance using the exact-rational `dot*`. We can give ourselves a speed versus accuracy knob to turn by implementing

$$\frac{1}{t} \leq -x / y \leq t \quad (13)$$

instead of the original addition badlands formula. Setting the "knob" t to 2 yields the original formula. With $t = 1.1$, using the exact-rational `dot*` as a fallback makes `dot` about nine times slower than the initial implementation and bounded by about 12 ulps error. With $t = 1.01$, `dot` with exact-rational fallback is only four times slower but is bounded by about 300 ulps error, according to randomized testing.

Our current implementation might prematurely overflow when given arguments outside of about $[-10^{150}, 10^{150}]$. We'll leave debugging premature overflow as an exercise. A simple solution punts to `dot*` when `dot` is given large arguments or returns an infinity. A more complicated solution scales the arguments by powers of two, which is exact as long as it doesn't overflow or underflow.

It's daunting to think of the millions of lines of floating-point code that have never been tested and debugged, simply for lack of tools and a bit of know-how.

It's probably more daunting to consider testing and debugging your own project, which is certainly much larger than the three small examples we've shown here. It can be done, and testing and debugging scales reasonably well. Start small and work toward larger functions. Move troublesome formulas into their own functions to debug them on their own. Document their badlands and avoid them. Build a parallel high-precision implementation from the bottom up. Remember that every small improvement can help a lot.

Also remember to step back and look at the big picture: you should test and debug your research question, too. Floating-point data is an approximation of real data, with up to half an ulp error. This data is often an approximation itself, with potentially much more error, of some real-life quantity. Your functions and algorithms have badlands. If the data is in the badlands and the computed answers are numbers, these two approximations could make all of their digits nonsense.

Fortunately, testing a research question can be done similarly to plotting log's error sensitivity in Figure 5. Repeatedly run the code that answers the question on randomly perturbed, representative data. Are the answers' errors millions of times

Build a parallel high-precision implementation from the bottom up. Remember that every small improvement can help a lot.

larger than the perturbations? Is some of the data nonzero but the answers near zero? Is some of the data far from zero and in an area of exponential growth? If the answer to any of these questions is "yes," your data might be in the badlands. If so, either your data and code need more precision, in which case you'll be very glad to have a parallel high-precision implementation, or you need to ask a different question.

We'll end with a dirty secret: almost everything we presented and demonstrated is error analysis in disguise. The condition number formula, for example, is straight out of any numerical methods textbook. But we replaced the burden of proof with testing, which reduces the burden on you. ■

References

1. M. Flatt, "Reference: Racket," tech. report PLT-TR-2010-1, PLT Inc., 2010; <http://racket-lang.org/tr1>.
2. L. Fousse et al., "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding," *ACM Trans. Mathematical Software*, vol. 33, no. 2, 2007, pp. 13:1–13:15.
3. "IEEE Standard for Floating-Point Arithmetic," IEEE Std 754-2008, Aug. 2008, pp. 1–70.

Neil Toronto recently graduated with a PhD from Brigham Young University, and is a postdoctoral researcher at University of Maryland, College Park. His research focuses on programming language support for mathematical computation, and currently emphasizes Bayesian modeling and inference. Contact him at neil.toronto@gmail.com.

Jay McCarthy is a visiting assistant professor of computer science at Vassar College and formerly an assistant professor at Brigham Young University. His research focuses on the nexus of systems, verification, and security from the perspective of applied programming language techniques. McCarthy has a BS in mathematics from the University of Massachusetts, Lowell, and a PhD in computer science from Brown University. Contact him at jay.mccarthy@gmail.com.