Grapheme: An Online Graphing Calculator

Timothy Herchen

October 2020

Contents

1 Floating-point Operations

Grapheme uses double-precision floating-point arithmetic for most calculations, since this functionality is provided by JS directly and is highly optimized. When the calculator is directed to evaluate $3 \cdot 4$, it uses the JS $*$ operator which maps directly to a machine instruction. There is no point in using single-precision arithmetic, as these are the same speed on modern processors and JS has no facilities besides asm.js to use this format.

There are some important limitations in double-precision FP. Some of the most obvious are the inability to express integers greater than $2^{53} \approx 9.007 \cdot 10^{15}$, numbers greater than about $2^{1023} \approx 1.798 \cdot 10^{308}$, and positive numbers smaller than $2^{-1074} \approx 4.941 \cdot 10^{-324}$. While arbitrary-precision arithmetic may be eventually implemented, this is difficult and thus we will try to do our best using the existing system.

Some conventions:

1. $\pm\infty$ and `NaN` are known as *special numbers.*

2. Floating-point numbers that are not special numbers are *finite numbers.*

3. Denormal numbers and normal numbers are named as usual.

4. `NaN` $\neq$ `NaN`, contrary to the mathematical definition of equality. However, `NaN` $\simeq$ `NaN`. For all other purposes, $=$ and $\simeq$ are equivalent.

5. There is only one `NaN` value, because the standard does not specify the existence of qNaNs, sNaNs and the like.

6. The set of all double-precision floating-point numbers, including the special numbers, is denoted $\mathbb{F}_{\text{all}}$.

7. $\mathbb{F}_{\text{all}}$ without the special numbers is denoted $\mathbb{F}$. Without only `NaN`, it is denoted $\mathbb{F}_\infty$. Thus, $\mathbb{F} \subset \mathbb{F}_\infty \subset \mathbb{F}_{\text{all}}$.

1.1 Directed Rounding

Per the ECMAScript standard, JS operations all use round-to-nearest, ties-to-even. That means that if the mathematical result of an operation is, say, 3.261, and the nearest permitted floats are 3.26 and 3.27, the operation will return 3.26. Unfortunately, JS does not provide facilities to set the rounding mode, which is understandable given the niche use of these modes. Grapheme does provide the functions `FP.roundUp` and `FP.roundDown`.

1.1.1 FP.roundUp and FP.roundDown

The basic idea is to treat a floating-point number as a 64-bit integer, which we can do via typed arrays. Incrementing this integer moves us to the next floating-point value, and decrementing it moves us to the previous one. This works for all values except special values, 0, and `-Number.MIN_VALUE` (the last is only an issue for `roundUp`). It even works for denormals!

1.2 Other Helpful Functions

`pow2(x)` computes 2^x exactly for integers $-1074 \leq x \leq 1023$, which is done via bit manipulation. `getExponent` and `getMantissa` return the non-biased exponent and mantissa, respectively, of a given float. `mantissaCtz` and `mantissaClz` count the number of trailing and leading zeros, respectively, of a given float. `frExp(x)`, for $x \in \mathbb{F}$, returns a floating-point number $0.5 \leq |f| < 1$ and integer e and guarantees that $f \cdot \texttt{pow2}(e) = x$. This guarantee is preserved for $x \in \mathbb{F}_{\text{all}}$, but f in that case may be `NaN` or $\pm\infty$. It handles denormalized numbers correctly but uses a special algorithm for them.

`rationalExp(x)` returns a reduced fraction n/d and exponent e such that $n/d \cdot \texttt{pow2}(e) = x$ in JS arithmetic. It does this by first using `frExp` to find f and e such that $f \cdot 2^e = x$, then representing f as a fraction $(f \cdot 2^{53})/2^{53}$; since $0.5 \leq |f| < 1$, the numerator is positive. It finally cancels out as many powers as two as it can to ensure it is reduced.

1.3 Intelligent Pow

Among the real numbers, exponentiation is straightforward for positive bases, but rather complicated for negative bases. Negative bases raised to a rational power are variously positive, negative or undefined, depending on the fraction. Among $\mathbb{F}$, however, negative bases raised to any power $p \in \mathbb{F} \setminus \mathbb{Z}$ are undefined. This can be logically seen from the fact that all floating-point numbers, besides the integers, are rational numbers with a power-of-two base. But the number $0.3333333333333333 \in \mathbb{F}$ likely refers to $\frac{1}{3}$, and so does 0.33333333333333326, but probably not 0.3333333333001.

1.3.1 Doubles to Rationals

We hence describe the function `doubleToRational`. There are two competing interests: one, to correctly recognize mathematical rational numbers that would be reasonably encountered in a graphing session, and two, to make coincidences that lead to the recognition of irrational numbers as rational unlikely. We restrict our further work to the positive domain, since this makes our life far easier. In other words, we want to find a reasonable function $d : \mathbb{F}_{>0} \to \mathbb{Z}^2$, so that the resulting numerator–denominator pair corresponds closely to the floating-point number argument.

To do this, we make some stipulations.

1. At most approximately $\frac{1}{10000}$ of floating-point numbers in any range $\left(2^n, 2^{n+1}\right)$ are classified as rational. This makes it unlikely that a randomly-generated real number will be considered rational.

2. The floating-point numbers corresponding to a rational number $\frac{p}{q}$ are at most those inside `RealInterval.from(p/q)`; in other words, they are either `p/q` as evaluated by JS, or the preceding or succeeding float.

3. There must be no intersections between these intervals.

4. We assert that the numerator and denominator be less than or equal in magnitude to $2^{53}-1$, which makes our life easier (and such numbers are the majority of rational numbers we'd encounter).

5. If we are to recognize any floats in a range, we must recognize up to at least denominator 100.

For example, `1/3` evaluates to the floating-point number 0.3333333333333333. The preceding and succeeding members of $\mathbb{F}$, namely 0.33333333333333337 and 0.33333333333333326 should also correspond to the same number, but no other numbers should correspond to the rational number $\frac{1}{3}$. Because the spacing between floating-point numbers varies, we split up our recognition algorithm over each of the 2046 sets $\left(2^n, 2^{n+1}\right)$ for $-1022 \leq n \leq 1023$ available to floating-point numbers. (Denormalized numbers are ignored, since they are too small.)

The minimum returnable rational number is $\frac{1}{2^{53}-1}$, which means that all numbers smaller than $\frac{1}{2^{53}}$ may be considered irrational. The minimum possible distance between two floating-point numbers in a given range is 2^{n-52}. The maximum error of `p/q` from its mathematical value, since it is round-to-nearest, is 2^{n-53}. Thus, the minimum distance between two allowed fractions must be greater than

$$\underbrace{2 \cdot 2^{n-53}}_{\text{rounding}} + \underbrace{3 \cdot 2^{n-52}}_{\text{interval widths}} = 2^{n-50}$$

to comply with stipulations 2 and 3. We wish to support all reduced rational numbers p/q with $q \leq d_n$ in a range $[2^n, 2^{n+1})$, where we choose d_n intelligently. The minimum distance between any two supported rational numbers is at least $\frac{1}{d_n^2}$, so

$$\frac{1}{d_n^2} \geq 2^{n-50} \implies d_n^2 \leq 2^{50-n} \implies d_n \leq 2^{25-n/2}. \tag{1}$$

Finally, to comply with the first stipulation, we wish to compute the number of rational numbers p/q with $q \leq d_n$ in the given range. This is approximately the length of the Farey sequence of order d_n (the number of such rationals between 0 and 1) times 2^n, which asymptotically is $\frac{3d_n^2 \cdot 2^n}{\pi^2}$. The number of floats classified as rational is three times this. The number of floats in the entire range is 2^{52}. Thus, to comply with the first requirement,

$$\frac{9d_n^2 2^n}{\pi^2} \leq \frac{1}{10000} \cdot 2^{52} \implies d_n^2 \leq \frac{\pi^2 \cdot 2^{52} \cdot 2^{-n}}{9 \cdot 10000} \implies d_n \leq \frac{\pi \cdot 2^{26-n/2}}{300}. \tag{2}$$

We see that (2) is always stricter than (1), and we know that $100 \leq d_n < 2^{53}$, so the final expression is now

$$\boxed{100 \leq d_n \leq \min\left\{\frac{\pi \cdot 2^{26-n/2}}{300}, 2^{53}-1\right\}}.$$

Such a d_n only exists for $n \leq 25$, so we can consider numbers above 2^{26} irrational. Our procedure for numbers $x \in [1/2^{53}, 2^{26}]$ is now as follows:

1. Get the exponent n via any reliable method (presumably the `getExponent` function).

2. Look up the corresponding maximum value of d_n.

3. Find the nearest rational number p/q whose denominator is less than or equal to d_n.

4. If $x - \frac{p}{q} \leq 2^{n-52}$, return p/q; otherwise, return nothing.

Step 1 can be done via looking at the binary representation of the function, which is what `getExponent` does. Step 3 is the tricky part to do rigorously and quickly. A bit more detail is given in the implementation, but the gist of it is below:

(a) Given a floating-point number x and integers `maxDenominator` and `maxNumerator`:

(b) If $x < 0$, return the negation of this function evaluated for $-x$.

(c) If x is not finite, return `NaN`, `NaN`, `NaN`.

(d) If x is an integer, return the minimum of `maxNumerator` and x for the numerator, 1 for the denominator, and the error.

(e) Compute the integer and fractional part of x, `flr` and `frac` respectively.

(f) Using `rationalExp`, find floats p and q in reduced form such that $p/q =$ `flr` and the equation is mathematically exact. q might overflow to ∞, but that only happens when x is smaller than 2^{1000} or so, in which case we return 0 as the nearest rational. The reason we operate on `flr` is because we can guarantee that p is a safe integer (though q, as mentioned, will be exact but not necessarily safe).

(g) If p/q satisfies the bounds given, return it with error 0.

(h) The continued fraction expansion of x is

$$\texttt{flr} + \texttt{frac} = \texttt{flr} + \frac{1}{\frac{q}{p}}.$$

Compute the integer part of $\frac{q}{p}$, `inv_flr`, and the remainder, `inv_rem`. `inv_flr` must be nudged so that rounding errors don't cause `Math.floor` to fail; see the code for how this happens. As an example, consider $x =$ `1/5`, which leads to calculating `inv_flr` for $\lfloor \frac{q}{p} \rfloor = \lfloor \frac{18014398509481984}{3602879701896397} \rfloor$. The JS result is 5 due to the fraction being closer to 5 than 5's predecessor in $\mathbb{F}$, but the mathematically exact result (which we need) is 4. `inv_rem` is exact per the ECMAScript standard, since the result is exactly representable.

(i) Define c_i as the exact, terminating continued fraction expansion of x, computed as

$$\texttt{flr} + \frac{1}{\texttt{inv\_flr} + \frac{1}{\frac{\texttt{inv\_rem}}{p}} \ddots}.$$

Note that $\texttt{inv\_rem}, p \leq \texttt{Number.MAX\_SAFE\_INTEGER}$, so we don't have to do any special handling here on out. Indexing: $c_1 = \texttt{flr}$ and $c_2 = \texttt{inv\_flr}$.

(j) We define additional recurrence relations. These provide the convergent numerators and denominators at each step.

   (a) Convergent numerators: $n_0 = 1, n_1 = \texttt{flr}, n_{i+1} = c_{i+1}n_i + n_{i-1}$.

   (b) Convergent denominators: $d_0 = 1, d_1 = \texttt{flr}, n_{i+1} = c_{i+1}n_i + n_{i-1}$.

(k) Define variables `bestN` = `Math.round`(x) and `bestD` = 1. We begin at $i = 2$.

(l) Compute c_i, n_i, and d_i via the recurrence relations. The computation of c_i from previous values is done simply by storing the numerator and denominator of the unresolved part of the continued fraction; the relevant variables are `contFracGeneratorNum` and `contFracGeneratorDen`.

(m) If n_i/d_i satisfies our bounds, record it in `bestN` and `bestD`, then go to (l).

(n) If one of the variables is too big, compute the maximum reduction r of c_i which would give n_i and d_i small enough values.

(o) If $r < \frac{c_i}{2}$, return `bestN/bestD` with its computed error.

(p) If $r > \frac{c_i}{2}$, compute the corresponding new value n_i'/d_i' and return that with its computed error.

(q) If $r = \frac{c_i}{2}$, compute n_i'/d_i', its error and the error of `bestN/bestD`. If the latter error is worse than the former, return the new convergent with its error; otherwise, return the old convergent with its error.

(r) If c_{i+1} does not exist, return the `bestN/bestD` with its computed error.

(s) Go to step (l)

This is implemented in `closestRational`. Together, this gives a complete implementation of rational-guessing for `pow`. An important optimization is that arguments to `doubleToRational` are cached, since often the exponent of `pow` is constant. We only cache the previous float; caching multiple floats is a rather finnicky process, since it would be expensive to store it in an associative array with strings as keys.

Once tagged real numbers are implemented, this issue should be less important, but will still likely be used as a fallback.

2 Interval Arithmetic