Conditioning your quantile function

Keith Pedersen[*]

(Dated: 25 April 2017)

Random sampling via a quantile function $Q(u)$ is a popular technique, but two very common sources of numerical instability are often overlooked: (i) quantile functions tend to be ill-conditioned when $u \to 1$ and (ii) feeding them uniformly spaced u can make them ill-conditioned as $u \to 0$. These flaws undermine the tails of $Q(u)$'s distribution, and both flaws are present in the polar method for normal sampling (used by GNU's `std::normal_distribution` and `numpy.random.normal`). Furthermore, quantile instability causes the polar method to lose precision *near the mode*; hence, it is not just Monte Carlos which study low-probability events which are sensitive to these effects, but also those with a very large sample size. This paper presents a novel method to mitigate quantile instability with minimal cost — feeding high entropy u into a "quantile flip-flop."

I. INTRODUCTION

A random sample $\{f\}$ from a probability distribution f can be easily generated if one can calculate f's cumulative distribution function F. Inverting F produces the quantile function $Q \equiv F^{-1}$. A sample of f is obtained by feeding $U(0,1)$, a random sample from the uniform distribution over the exclusive[1] unit interval, into Q;

$$\{f\} = Q\left(\{U(0,1)\}\right). \tag{1}$$

This is called inverse transform sampling, and it is powerful because the quantile function is formally exact. Unfortunately, floating point arithmetic is *formally inexact* [1], which breaks the quantile function; but by how much?

Computers do real number arithmetic using binary floating point numbers — scientific notation in base-2 [1]. The mantissa has fixed precision P (the total number of binary digits) and the integer E for the exponent 2^E must exist in a finite range. The finite precision of floating point numbers forces them to sample a discrete, rational subset of the real number line. All possible values of the mantissa evenly span $[1,2)$, with the distance between adjacent values called *machine epsilon* ($\epsilon = 2^{-P+1}$) [2]. Yet the factor of 2^E makes floating point numbers unevenly spaced across scales; every time the exponent decrements, the set of numbers spanned by the mantissa becomes twice as dense. The toy model depicted in Fig. 1 uses $P = 3$, and thus has a comically large $\epsilon = 0.25$. The common choice for floating point formats — "double" precision (officially designated *binary64* by the IEEE 754 standard [3]) — uses $P = 53$, so $\epsilon \approx 2.22 \times 10^{-16}$; this is generally small enough to avoid *most* numerical problems.

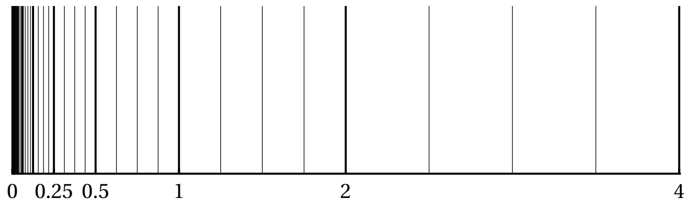

FIG. 1: Digital spacing for a toy binary floating point type with $P = 3$ (two digits after the decimal).

II. THREE WAYS TO LOSE PRECISION

Unfortunately, a minuscule ϵ is not a magic bullet for numerical stability. It is actually rather easy to decrease the effective precision of a floating point calculation far below the ~16 decimal digits one expects from *binary64*. This is especially important for random samples, since they tend to be used at the start of a complicated calculation, and are often treated like black boxes that emit perfect randomness. The latter may be a poor attitude, but it is often difficult to empirically analyze the quality of random samples without exhaustive testing. This section takes the more

[*] Keith.David.Pedersen@gmail.com

[1] The endpoints of the unit interval are excluded because $F(0)$ and $F(1)$ are formally ill-defined.

direct approach; probing the *theory* behind the numerical stability of inverse transform sampling. It turns out there are three common pitfalls which cause quantile functions to lose precision.

A. $Q(u)$ are usually ill-conditioned

Probability distributions tend to have *tails* — large swaths of improbable sample space. For unimodal distributions, the tails are usually mapped by $u \to 0$ and $u \to 1$, since these endpoints represent the min/max values of the cumulative distribution F. Populating the tails with a quantile function requires a thin slice of the unit interval to map out a large range of values, which can be pathological for floating point arithmetic.

1. *The condition number $C(g)$*

When a function $g(x)$ is changing rapidly, small shifts in its input can produce large shifts in its output (e.g. if changing x's mantissa by one unit in the last place causes the mantissa of $g(x)$ to change by 1,000 units in the last place). In such scenarios, $g(x)$ loses *effective* precision — intermediate values of $g(x)$ are representable with floating point numbers, but are not attainable from floating point arithmetic. This is problematic because: (i) if $g(x)$ is continuous, it must pass through the missed values, and (ii) $g(x)$'s diminished precision will propagate to subsequent calculations. This problem is quantified by the *condition number*, the relative change in a function's output versus its relative change in input [4]

$$C(g) \equiv \left| \frac{g(x+\delta x) - g(x)}{g(x)} \Big/ \frac{\delta x}{x} \right| = \left| x\, \frac{g'(x)}{g(x)} \right| + \mathcal{O}(\delta x) \tag{2}$$

$$\approx \left| \frac{\mathrm{d} \log g(x)}{\mathrm{d} \log x} \right| . \tag{3}$$

When $C(g)$ is large, $g(x)$ is an **ill-conditioned** calculation — adjacent inputs cause it to skip over huge swaths of representable numbers, making $g(x)$ extremely sensitive to small errors in the input. Since the accumulation and propagation of *rounding* error is unavoidable in floating point arithmetic, an ill-conditioned function is bad news.

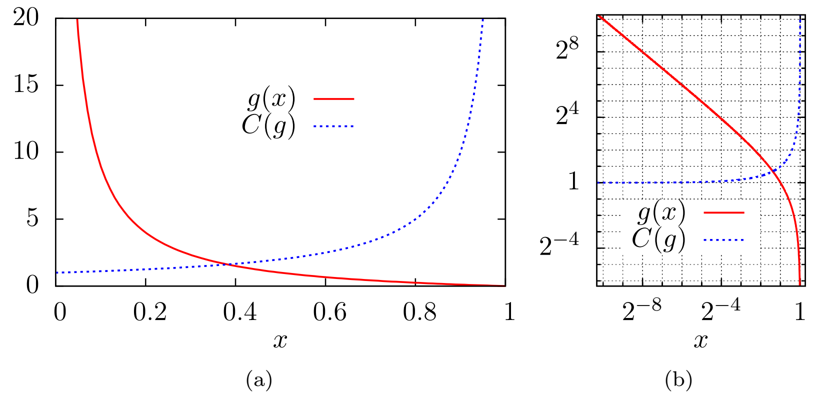

FIG. 2: (a) $g(x) = \frac{1-x}{x}$ and its condition number and (b) on log-log plot.

Figure 2a shows the function $g(x) = \frac{1-x}{x}$, which blows up as $x \to 0$ and smoothly goes to zero as $x \to 1$. Curiously, $g(x)$ is well-conditioned at its singularity, and ill-conditioned at its root. The beginning of an explanation lies in Eq. 3, which says that $g(x)$ is ill-conditioned whenever it changes rapidly on a log-log plot (shooting towards infinity *or* zero). Because floating point numbers are *nearly* evenly distributed in log-space ($\delta x/x = \delta g/g = \mathcal{O}(\epsilon)$ at all scales), adjacent floating-point numbers can be roughly depicted by the ticks on Fig. 2b's log-log scale. Thus, $g(x)$ is visibly well-conditioned as $x \to 0$, because it maps adjacent inputs to adjacent outputs, and visibly ill-conditioned as $x \to 1$. An alternative explanation is that when $g(x)$ experiences a rapid relative change, maintaining a small $\delta g/g$ requires a vanishing δx. This sends $\delta x/x \to 0$, exploding the condition number ... *except* when $x \to 0$. The density of representable x increases nearly arbitrarily as one approaches the origin, so $g(x)$ can be mapped out more slowly there. This extra x-precision near the origin is why Eq. 2 is proportional to x, a term which softens $g'(x)/g(x)$ singularities near the origin, allowing rapidly changing functions to remain well-conditioned there.

2. *The quantile flip-flop*

Recall that quantile functions are almost guaranteed to be ill-conditioned, since they must change rapidly when they map out a distribution's tail. We can see this effect in an exponential distribution with mean $\mu = 1$

$$f(r) = e^{-r} \quad\longrightarrow\quad F(r) = 1 - e^{-r} \quad\longrightarrow\quad Q_1(u) = -\log(1-u) \quad\longrightarrow\quad C(Q_1) = -\frac{u}{(1-u)\log(1-u)}. \tag{4}$$

$Q_1(u)$ is well behaved as $u \to 0$, but loses precision as $u \to 1$, where it maps the large-value tail (see Fig. 3a). This problem is paired with a floating point cancellation[2] $(1-u)$ that discards precision when $u \to 1$. Mapping $u \mapsto 1-u$ removes the cancellation *and* gives the same sample, since $U(0,1)$ is perfectly symmetric across $u = 1/2$. This creates a second version of the quantile function

$$Q_2(u) = -\log(u) \quad\longrightarrow\quad C(Q_2) = -\frac{1}{\log(u)}. \tag{5}$$

Unfortunately, the condition number of $Q_2(u)$ still blows up as $u \to 1$, this time while sampling the small-value tail (this is still a tail since *very tiny* values near the mode are just as improbable as very large values). Note that the large-value tail has moved to $u \to 0$ — removing the cancellation has swapped the tails' locations.

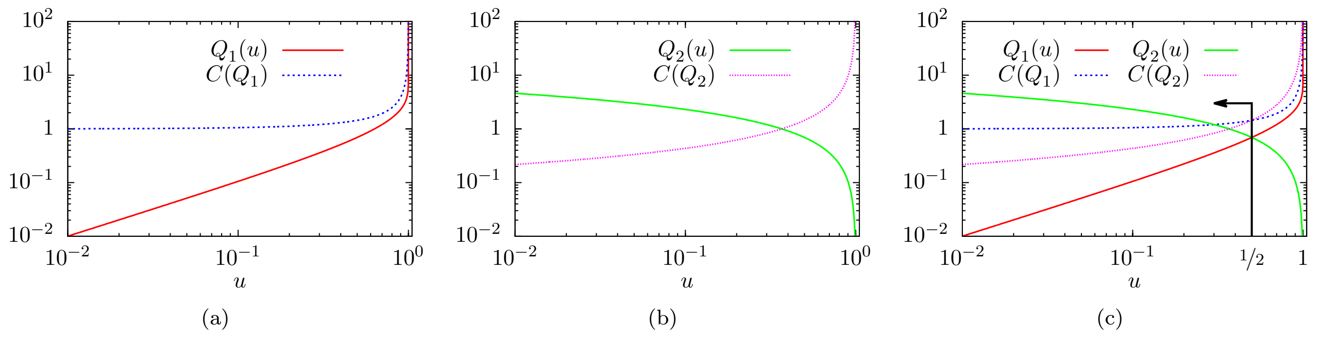

FIG. 3: The quantile functions Q_1 (a) and Q_2 (b) for $f(r) = e^{-r}$. In the quantile flip-flop (c), each version of Q maps out half of f over $0 < u \leq 1/2$, where each is well conditioned.

Each version of the quantile function stably maps one tail ($u \to 0$), and unstably maps the other ($u \to 1$). This is no accident; $Q'(u)/Q(u)$ is singular for both tails, in both versions, but this singular component of the condition number is softened by the arbitrarily small floating point precision as $u \to 0$ (the factor of x in Eq. 2). Now examine $Q_1(u)$ and $Q_2(u)$ in Fig. 3c. Each maps out half of the sample range in $0 < u \leq 1/2$ — sampling one of the tails at $u \to 0$ and meeting at the median at $u = 1/2$ — and each is well conditioned over the entire domain. This leads us to a novel method to condition a quantile function; use *both* versions of the quantile function, plugging $\{U(0, 1/2]\}$ into each,[3] and randomly choosing which Q to use for each instance (since simply alternating between them would introduce a cyclic bias in the sample). The only drawback to this **"quantile flip-flop"** is an extra, unpredictable branch in the code.

B. $Q(u)$ can be further ill-conditioned by low-entropy $\{U(0,1)\}$

Inverse transform sampling is a stochastic process; the quantile function is purely deterministic, and $\{U(0,1)\}$ supplies *all* the randomness. Since sources of true randomness are hard to obtain (at least at GHz frequencies), Monte Carlos tend to rely on pseudorandom number generators (PRNG), deterministic state machines which mimic randomness. PRNGs are also beneficial for debugging or auditing, since they can be re-seeded to produce the same sequence.[4] A good PRNG should have a very long period before its sequence resets,[5] should lack n-dimensional

[2] The importance of cancellation cannot be over-emphasized, but it is beyond the scope of this paper. See Ref. [1] for a thorough review.
[3] Note that we must now draw u from a semi-inclusive, half-unit interval. This allows the median to be sampled, but requires care during implementation, so that the median has the same probability as adjacent values. This will be covered in the next section.
[4] Deterministic PRNG usage is naïvely broken in multi-threaded applications, since thread scheduling is non-deterministic.
[5] If you need N random numbers, the period of your PRNG should be much greater than N^2 [5].

correlations between adjacent values, and should generate values which are uniformly distributed (the most randomness you can get from a given interval).

Some PRNGs are designed to supply $\{U(0,1)\}$ directly [6, 7], but this approach conflates the deep magic of generating pseudorandom bits with the important math responsible for turning those bits into $\{U(0,1)\}$. The philosophy of this paper is, "Let someone else design an excellent PRNG that draws N-bit unsigned integers independently and uniformly from $[0, 2^N)$. Instead, concern yourself with transforming those integers into $\{U(0,1)\}$." We will therefore leave the determination of an excellent PRNG as an exercise for the reader.

1. *Superuniform $\{U(0,1)\}$ have unpleasant side effects*

Algorithm 1 is the canonical method for sampling $U(0,1)$ in a computer;[6] generate a random integer in $(0, 2^P)$, convert it to floating point,[7] then divide by 2^P. This is how $\{U(0,1)\}$ is produced in Python's `random.random` [8] and GNU's `<random.h>` (via `std::generate_canonical` [9]). Algorithm 1 selects $\{U(0,1)\}$ from $(2^P - 1)$ possible values, evenly spaced in $(0,1)$ with a separation of exactly ϵ, creating a **"superuniform"** $\{U(0,1)\}$ [5]. Since this is only a discrete slice of the idealized $\{U(0,1)\}$, we should refer to the sample as $\{U^*_\mathrm{S}(0,1)\}$ (with the star denoting the rational, floating point subset of $\mathbb{R}$).

```
def randU_S() : return float(random.randint(1, 2**P - 1)) * 2.**-P
```

Algorithm 1: Generating superuniform $\{U^*_\mathrm{S}(0,1)\}$ in Python 2.7.

The condition number implicitly assumes that the input-space gets denser as $x \to 0$. This fails for superuniform $\{U^*_\mathrm{S}(0,1)\}$, because $\delta x = \epsilon$ everywhere. This suggests that the effective precision, the relative change in $g(x)$

$$P^*(g) = \left|\frac{g(x+\delta x) - g(x)}{g(x)}\right| = \delta x \left|\frac{g'(x)}{g(x)}\right| + \mathcal{O}(\delta x^2), \tag{6}$$

is a better metric for quantile stability. A large effective precision is *bad*, and will infect any subsequent calculations with the same imprecision. Unrestricted floating point input uses $\delta x = \mathcal{O}(\epsilon\, x)$, so the effective precision for generic floating point calculations is $P^*(g) \approx \epsilon\, C(g)$. This is why the condition number is normally a good metric.[8] But superuniform $\{U^*_\mathrm{S}(0,1)\}$ use $\delta u = \epsilon$, so that

$$P^*_\mathrm{S}(Q) \approx \epsilon \left|\frac{Q'(u)}{Q(u)}\right| + \mathcal{O}(\epsilon^2). \tag{7}$$

The effective precision of a quantile function now lacks the factor of u that previously suppressed singularities in $|Q'(u)/Q(u)|$ near the origin. This is because superuniform u do not utilize the full precision available to the floating point data type; as $u \to 0$, they do not grow denser. This additionally manifests as a loss of entropy in the mantissa; given some superuniform $u < 2^{-n}$, its mantissa must have at least n trailing zeroes.

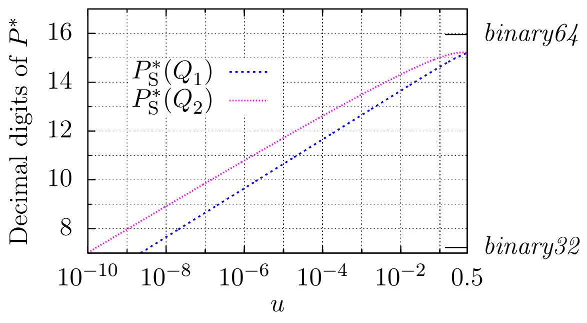

FIG. 4: Number of decimal digits of effective precision for Eq. 4's quantile flip-flop, given superuniform $\{U^*_\mathrm{S}(0,1)\}$.

[6] We can study the algorithms used to produce $\{U(0,1)\}$ in Python, since it looks like pseudocode, but is actually functional. To keep the algorithms generic, we will treat the floating point precision P and unsigned integer bit-length N like parameters (which must be defined for the Python code to work). The Python `float` type is *binary64* ($P = 53$), and though Python doesn't have an unsigned integer type, you can mimic one using the positive values of the signed integer (i.e. $N = 63$).

[7] An IEEE 754 floating point can exactly store any integer in $(-2^P, 2^P)$, with an exact conversion, by placing the integer in the mantissa.

[8] When $g'(x) = 0$, so is the effective precision, even though it should never dip below the minimum precision of the data type ($\epsilon/2$). This is an artifact of Eq. 6's assumption that $g(x)$ is continuous, not quantized.

Superuniform $\{U^*_{\mathrm{S}}(0,1)\}$ swiftly un-conditions a quantile flip-flop developed for $f(r)=e^{-r}$. Both $Q_1(u)$ and $Q_2(u)$ now lose precision as $u\to 0$ (although it is significantly worse for $Q_1(u)$, which maps the small-value tail),

$$P^*_{\mathrm{S}}(Q_1) = -\frac{\epsilon}{(1-u)\log(1-u)} \underset{u\to 0}{\approx} \frac{\epsilon}{u}, \qquad P^*_{\mathrm{S}}(Q_2) = -\frac{\epsilon}{u\log(u)}. \tag{8}$$

This is evident in Fig. 4, which plots the number of decimal digits of effective precision for this quantile flip-flop. The entire point of using *binary64* is to bury numerical instabilities deep enough that you *never* have to worry about them rising from the grave, but by the time $u \lessapprox 1\times 10^{-9}$, the sample has essentially become *binary32*. This is quite troubling, because a one-in-a-billion event is not very rare in a modern Monte Carlo.

We can visualize the bias of Alg. 1 empirically. Since an IEEE 754 floating point mantissa is always normalized (there is always a 1 before the decimal), there are 2^{P-1} possible mantissae. An idealized $\{U(0,1)\}$ rounded to the nearest floating point should fall with equal probability into each mantissa, simply preferring exponents with a probability of 2^{-E} (for $E>0$). Given a sample $\{U^*_{\mathrm{S}}(0,1)\}$ of size of 2^L, we can count how many times each mantissa appears. Binning these counts into a normalized histogram gives us the probability mass function (PMF) for the number of mantissa collisions (e.g. what's the probability a mantissa is drawn eight times?). This provides a strong constraint on the uniformity of mantissa sampling, because if all mantissae are equally probable, then the PMF for the number of collisions should follow a Poisson distribution with mean $\lambda = 2^{L-P+1}$.

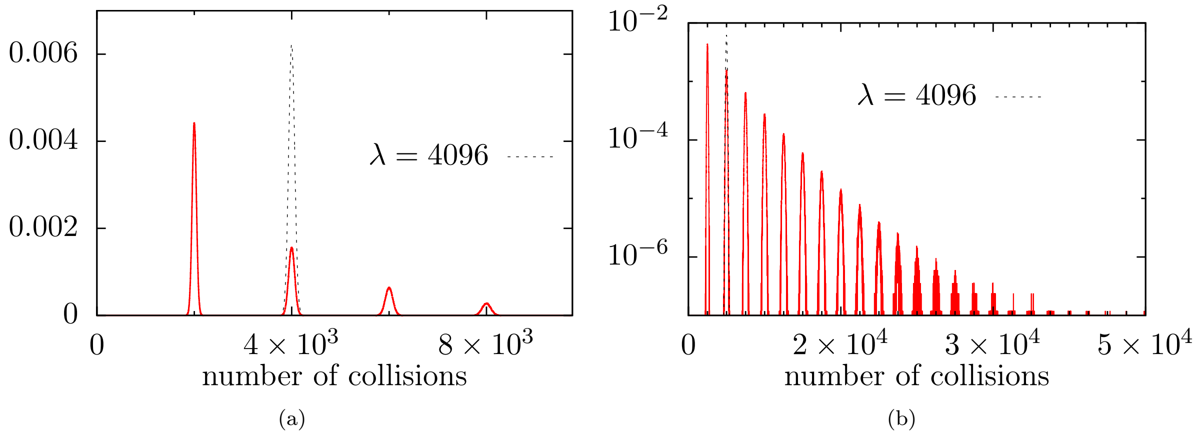

FIG. 5: (a) The probability of mantissa collision for Alg. 1 using *binary32* with a sample size of 3.44×10^{10} ($L=35$) and (b) on a semi-log scale. Overlayed is the Poisson distribution expected if all mantissae have equal probability.

Fig. 5 shows the PMF for mantissa collision when Alg. 1 is implemented in C++, using the MT19937 PRNG ($N=32$, via `std::mt19937`) and *binary32* ($P=24$).[9] The multiple Poisson distributions demonstrate that the mantissae are not uniform; some are much more popular than others. This is to be expected; the even spacing of the superuniform algorithm restricts low-probability u to a subset of mantissae, each separated by long runs of prohibited mantissae. These underutilized mantissae create the quantile instability as $u\to 0$.

2. *Quasiuniform* $\{U(0,1)\}$ *at the superuniform price*

The numerical stability of the quantile flip-flop was ruined by the rigidly even spacing of superuniform $\{U^*_{\mathrm{S}}(0,1)\}$. Hence, the obvious fix is to enable uneven spacing. We can visualize this problem in terms of entropy. With a superuniform $\{U^*_{\mathrm{S}}(0,1)\}$, any $u<2^{-n}$ always has at least n trailing zeroes in its mantissa — clearly less than optimal. Refilling the always-zero bits with new entropy unlocks rational numbers that aren't available in $\{U^*_{\mathrm{S}}(0,1)\}$, permitting the **"quasiuniform"** $\{U^*_{\mathrm{Q}}\}$ to span all representable floating point u, while still being uniformly distributed.[10] A quasiuniform $\{U^*_{\mathrm{Q}}\}$ should therefore be equivalent to a true sample of $\{U\}$ from $\mathbb{R}$, rounded to the nearest floating

[9] The lower precision of *binary32* permits a tractable investigation, but we expect equivalent results for *binray64*, since both types are regulated by the same IEEE 754 floating point standard.

[10] My apologies to mathematicians if I am butchering the term "quasiuniform"; it seemed appropriate.

```python
def randUNSIGNEDint() : return random.randint(0, 2**N - 1)

def randHalfU_Q() :
   #  randInt needs (P+2) bits of entropy: P for the mantissa and 2 extra for rounding
   randInt = randUNSIGNEDint()
   #  Scale randInt from [0, 2**N) to (0, 0.5] (with implicit rounding in the return statement)
   scaleToU = 0.5 * 2.**-N
   # Top up the entropy when there's not (P+2) bits
   if randInt < 2**(P+1) :
      shiftLeft = 0 # The size of randInt's leftward bit-shift
      while(randInt == 0) : # There's no leading bit; keep shifting N bits left until one is found
         scaleToU *= 2.**-N
         randInt = randUNSIGNEDint()
      while randInt < 2**(P+1) : # Keep shifting left till the leading bit is in the correct place
         randInt <<= 1
         shiftLeft += 1
      #  Get new bits and insert them into randInt with BINRAY-OR
      randInt |= (randUNSIGNEDint() >> (N - shiftLeft))
      #  Keep u in its original location, just with more entropy
      scaleToU *= 2.**(-shiftLeft)
   #  Set the "sticky" bit to 1, round the nearest floating point, then scale
   return float(randInt | 1) * scaleToU
```

Algorithm 2: Generating quasiuniform $\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$ in Python 2.7.

point number. Algorithm 2 generates $\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$ from the semi-inclusive, half-unit interval (as required by a quantile flip-flop), ensuring that its entropic refilling does not spoil the uniformity of U.

Algorithm 2 starts by drawing a random, unsigned integer. It must be greater than 2^{P+1} because we need $(P+2)$ bits of entropy; P bits to fill the mantissa, and two extra bits to ensure proper rounding (explained in the next paragraph). If we don't have $(P+2)$ bits, we shift the bits left until the leftmost (most significant) bit slides into position. Then we fill the vacated space on the right with new random bits, and factor the bit shift into `scaleToU`; this ensures that u's coarse location is unchanged, we've merely altered its fine location (and thus preserved uniformity). Finally, we set the "sticky" bit (a trick learned from Ref. [10]), round the integer to floating point, then scale it into $[0, {}^1\!/\!_2)$. Algorithm 2 is in fact quite similar to the one presented in Ref. [10], although Alg. 2 is faster because it only calls the PRNG to top up the entropy when absolutely necessary (instead of every time).

Any integer larger than 2^P cannot be exactly converted to floating point because there are not enough bits in the mantissa (unless the integer has sufficient trailing zeroes). A naïve conversion would simply truncate the integer to P digits, but IEEE 754 requires the integer to be rounded. Most computers default to the most numerically stable rounding mode — round-to-nearest, ties-to-even (R2N-T2E) — but this mode is problematic for Algorithm 2 because it makes *even* mantissae slightly more probable than odd ones. We actually want naïve truncation, and to force this behavior the last two (least significant) bits of `randInt` are used to defeat R2N-T2E during floating point conversion.[11] The last bit is the "sticky" bit, and always setting it to 1 creates systematic rounding resolution; if every integer is odd, it is always closer to only one of the truncated options, but without giving preference to the even option. The sticky bit only fails when the integer has exactly $(P+1)$ digits, so that only the sticky bits needs to be truncated. In this case, the integer is again equidistant from its two truncated options, and T2E kicks in. Thus, at least one "buffer" bit is required between the sticky bit and the last bit of the mantissa — `randInt` needs $(P+2)$ bits.

```python
def randHalfU() : return 0.5 * float(randUNSIGNEDint()) * 2.**-N
```

Algorithm 3: Generating $\{U^*(0, {}^1\!/\!_2]\}$ simply, with extra precision, in Python 2.7.

While testing Alg. 2, we can demonstrate the necessity of its two rounding bits by testing a "bad" version that doesn't use them. Additionally, we can test Alg. 3, which quickly generates $\{U^*(0, {}^1\!/\!_2]\}$ with extra precision. The test results are shown in Fig. 6. Clearly, Alg. 2 samples all floating point mantissae with perfect uniformity, since it is indistinguishable from the theoretical Poisson distribution. Conversely, while Alg. 3 and the "bad" Alg. 2 do a much better job than Alg. 1, they look like the sum of two slightly offset Poisson distributions, indicating their slight distaste for odd mantissae.

[11] A true hacker would use the IEEE 754 bit format to cook up U^* via bit manipulation, without rounding bits, but Alg. 2 is more portable.

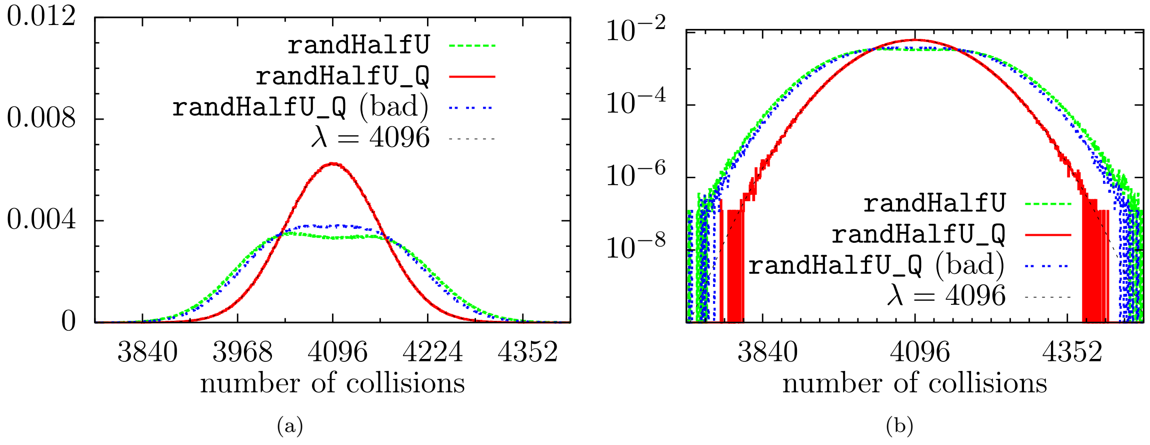

FIG. 6: (a) The probability of mantissa collision for Algs. 2 and 3 using *binary32* with a sample size of 3.44×10^{10} ($L = 35$) and (b) on a semi-log scale. The Poisson distribution for equi-probable mantissae overlaps Alg. 2.

An important feature of Alg. 2's rounding behavior is that $u = {}^1\!/\!_2$ is *half as probable* as its next-door neighbor, $u = \frac{1}{2}(1 - {}^{\epsilon}\!/\!_2)$. Imagine dividing the domain $[{}^1\!/\!_4, {}^1\!/\!_2]$ into 2^{P-1} bins, with the bin edges depicting the representable u. Each u absorbs a full bin via rounding (half a bin to its left, half a bin to its right) *except* $u = {}^1\!/\!_2$, which can only absorb half a bin from the left. But recall that $\{U^*_\mathrm{Q}(0, {}^1\!/\!_2)\}$ is intended for use in a quantile flip-flop — a normal quantile function folded in half at the median ($u = {}^1\!/\!_2$). Since *both* quantiles map to the median when they are fed $u = {}^1\!/\!_2$, the median will be double-counted *unless* $u = {}^1\!/\!_2$ is half as probable.

Quasiuniform $\{U^*_\mathrm{Q}(0, {}^1\!/\!_2]\}$ were motivated by $P^*_\mathrm{S}(Q)$ blowing up as $u \to 0$. An example where this does not occur is the distribution that is evenly spread in log-space between $a \le x \le b$,

$$f(x) = \frac{1}{x}\frac{1}{\log(b/a)} \quad \longrightarrow \quad F(x) = \frac{\log(x/a)}{\log(b/a)} \quad \longrightarrow \quad Q_1(u) = a\,(b/a)^u\ ,\ Q_2(u) = b\,(b/a)^{-u}. \tag{9}$$

We have implemented a quantile flip-flop by solving for $Q(u)$, then transforming $u \mapsto 1 - u$, and simplifying. When both quantile functions are supplied superuniform $\{U^*_\mathrm{S}(0, 1)\}$, their effective precision is constant;

$$P^*_\mathrm{S}(Q_{1,2}) = \epsilon \log(b/a). \tag{10}$$

This makes sense, because the quantiles are trying to evenly distribute numbers across log-space. To generate evenly spaced output, we actually need evenly spaced input. So although this distribution benefits from a quantile flip-flop, feeding the flip-flop $\{U^*_\mathrm{Q}(0, {}^1\!/\!_2]\}$ isn't neccessary. As $u \to 0$, both quantiles asymptotically flatten to a and b (respectively), so any extra precision is rounded away during exponentiation. Thus, there is no benefit to using quasiuniform $\{U^*_\mathrm{Q}(0, {}^1\!/\!_2]\}$ for this distribution. However, one must still ensure that $u = {}^1\!/\!_2$ is half as probable to use a quantile flip-flop. This is accomplished by Alg. 4.

```python
def randBool() : return (not random.getrandbits(1))

def randHalfU_S() :
   randInt = random.randint(0, 2**N - 1)
   if(randInt != 0) :
      return float(randInt) * 2.**-N
   else : # map 0 to 0.5
      if randBool() : # But only return 0.5 with half the probability
         return 0.5
      else : # The probability of drawing another zero is vanishingly small; no endless recursion
         return randHalfU_S()
```

Algorithm 4: Generating $\{U^*_\mathrm{S}(0, {}^1\!/\!_2]\}$ for a quantile flip-flop in Python 2.7.

Most $Q(u)$ have an effective precision that blows up as $u \to 0$m, requiring quasiuniform $\{U^*_\mathrm{Q}\}$. So what is the cost of its algorithm versus the "one-liner" for superuniform $\{U^*_\mathrm{S}(0, 1)\}$? Luckily, Alg. 2 is only slightly slower than Alg. 1, because line 9 is rarely true (1 in 1024 when $N = 64$ and $P = 53$, as in a real implementation). The code to tops up the entropy is rarely needed, making the main conditional branch quite predictable. The expected extra overhead comes from verifying that `randInt` has enough bits, then setting the sticky bit.[12] Armed with a high-entropy/high-precision $\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$, with minimal cost, we can *almost* restore Eq. 4's quantile flip-flop to the precision promised by the condition number ... there is one last fly in the ointment.

C. Cancellation ruins high-entropy $\{U(0, 1)\}$

$\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$ is very susceptible to floating point cancellation. For example, Eq. 4 uses $Q_1(u) = -\log(1-u)$. But the $(1-u)$ immediately strips away all the extra entropy from $\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$, converting it back into $\{U^*_{\mathrm{S}}(0, {}^1\!/\!_2]\}$. Luckily, there is a more numerically stable version of expressions that looks like $\log(1 + x)$ [11]

$$\log(1 + x) \approx \texttt{log1p(x)} \equiv \begin{cases} x & x \le \epsilon/2 \\ \frac{x\log(1+x)}{(1+x)-1} & x > \epsilon/2 \end{cases}. \tag{11}$$

The denominator in the second case looks rather silly, but when implemented in floating point arithmetic, the seemingly unnecessary parenthesis intentionally create rounding error to compensate for the rounding error in the numerator. Since `log1p` should only be used for positive arguments,[13] we can convert the first quantile function as

$$Q_1(u) = -\log(1-u) = \log\left(\frac{1}{1-u}\right) = \log\left(1 + \frac{u}{1-u}\right) = \texttt{log1p}\left(\frac{u}{1-u}\right). \tag{12}$$

While the final expression still has a cancellation in the denominator, its output is dominated by the numerator.

Although this paper is about conditioning quantile functions, the world's most popular statistical shape — the normal/Gaussian distribution — doesn't possess an analytic $Q(u)$. While this is true prima facie, one of the most popular ways to draw from the normal distribution (used by both GNU's `std::normal_distribution` [12] and NumPy's `numpy.random.normal` [13]) is the Marsaglia polar method (a form of Box-Muller),[14] which internally uses

$$f(r) = re^{-r^2/2} \quad\longrightarrow\quad F(r) = 1 - e^{-r^2/2} \quad\longrightarrow\quad Q_1(u) = \sqrt{-2\log(1-u)} \quad\longrightarrow\quad Q_2(u) = \sqrt{-2\log(u)}. \tag{13}$$

Therefore, a very common method to generate a normal sample *does indeed* use a quantile function, which should be well-conditioned. Since Eq. 13 is very similar to Eq. 4, their quantile functions possess almost identical condition numbers, so we can use the same prescription: (i) a flip-flop for $u \to 1$ and (ii) high-entropy $\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$ for $u \to 0$.

```
  # Return a pair of random, independent, normally distributed numbers (mu = 0, sigma = 1)
2 def stdNormalPair() : # To convert to another normal distribution: y = mu + sigma * x
     r2 = 2. # dummy value, too large
4    while r2 >= 1. : # Require points within the unit circle
        x = 1. - 2.*random.random()
6       y = 1. - 2.*random.random()
        r2 = x*x + y*y
8    scale = math.sqrt(-2.*math.log(r2)/r2) # Only Q2 is used, as Q2(r2)/sqrt(r2)
     return [scale*x, scale*y]
```

Algorithm 5: Normal sampling via the Marsaglia polar method in Python 2.7

To correctly condition Eq. 13, we need more details about how it is used. Algorithm 5 is the polar method as implemented by GNU and NumPy. Inserting the flip-flop at line 8 should be trivial, but using $\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$ for line 4 and 5 is simply not going to work. The cancellation will converts quasiuniform $\{U^*_{\mathrm{Q}}(0, 1]\}$ back to superuniform $\{U^*_{\mathrm{S}}(0, 1]\}$, keeping the quantile functions ill-conditioned for $u \to 0$. To remove the cancellation, we must understand the purpose of the subtraction. All it does is uniformly map $[0, 1] \mapsto [-1, 1]$; it's a cheap way to get a random sign.

The solution to high-precision normal sampling is Alg. 6, which begins on line 1 with a slow (but simple) way to sample $\{U^*_{\mathrm{Q}}(0, 1]\}$ with a random sign (scaling $\{U^*_{\mathrm{Q}}(0, {}^1\!/\!_2]\}$ from $(0, {}^1\!/\!_2]$ to $(0, 1]$). To sample the median, we must

[12] Since the entropy is rarely topped up, there's no real benefit in eluding the while loop (e.g. using the x86 instruction `BSR`).
[13] `log1p`'s floating point tricks cannot alter the fact that $\log(1 + x)$ becomes ill-conditioned as $x \to -1^+$.
[14] Marsaglia polar avoids trigonometric functions, which are ill-conditioned outside of $\pi/4 \le \phi \le \pi/4$.

```
   def randSignedU_Q() : return 2.*(1.-2.*random.getrandbits(1)) * randHalfU_Q()
 2
   def stdNormalPair_highPrecision() :
 4    r2 = 2.
      # Allow u = 1., which maps to the median, but only 1/3 of the time
 6    while (r2 > 1.) or ((r2 == 1.) and (float(randBool())*2. < 3.)) :
         x = randSignedU_Q()
 8       y = randSignedU_Q()
         r2 = x*x + y*y
10    if randBool() : # implement the quantile flip-flop
         scale = math.sqrt(-2.*math.log(0.5*r2)/r2)
12    else:
         scale = math.sqrt(2.*math.log1p(r2/(2.-r2))/r2)
14    return [x*scale, y*scale]
```

Algorithm 6: High-precision normal sampling in Python 2.7

allow $r^2 = 1$ (the u of the quantile function), but this time r^2 can round to unity from the left *and* the right. To prevent *triple*-counting the median, we can only accept rounding from the left. This requires rejecting $^2/_3$ of $r^2 = 1$, since the rounding regions have a width of $w_L = {}^{\epsilon}/_4$ to the left and $w_R = {}^{\epsilon}/_2$ to the right. We then implement the flip-flop on line 10 by randomly choosing one quantile and using $u = \frac{1}{2}r^2$ (note that this conversion does not affect the denominator of the square root, since that r^2 is not part of the quantile, but scales x and y to the unit circle).

Algorithm 6 is easy to read in Python, and covers the meat-and-potatoes of high-precision quantile sampling. However, it is far too inefficient for general use, which is why we have written the (P)recise (Q)uantile flip-flop (Rand)om package (`pqRand`) [14], a free implementation of normal, log-normal, exponential, weibull and pareto distributions in C++, and the tools for manually implementing custom distributions (of which there are many).

III. CONCLUSION

Double precision (*binary64*) has an excellent intrinsic precision that is generally small enough to suppress relevant numerical errors. However, in a large Monte Carlo, with many non-linear connections, it can be difficult to validate that numerical errors are acceptably small. The easiest strategy is to ensure that each component is as reliable as possible, so that the resulting apparatus is optimally well-conditioned. Since random numbers are the backbone of MonteCarlo, are often used near the start of long analysis chains, and their generators are often treated like a "black box" of perfect randomness, it is very important to ensure that random sampling algorithms are numerically stable. This is especially true when the cost of optimal performance is minimal.

This paper lays out the road map for conditioning a quantile sampling algorithm, which we will summarize here.

1. Given some quantile function $Q(u)$, use its condition number $C(Q)$ to check for ill-conditioned results. If $C(Q) \to \infty$ as $u \to 1$, implement a quantile flip-flop:

   (a) Create a second $Q(u)$ by mapping $u \mapsto 1 - u$. Each $Q(u)$ draws half of the sample (split at the median).

   (b) Supply each $Q(u)$ with $\{U^*(0, {}^1/_2]\}$ (ensuring that $u = {}^1/_2$ is half as probable as its neighbors).

   (c) During sampling, randomly switch between $Q_1(u)$ and $Q_2(u)$, to prevent bias in the sample.

2. Given a quantile function, use $P^*_{\rm S}(Q)$ to test whether superuniform $\{U^*_{\rm S}\}$ will create an ill-conditioned sample as $u \to 0$. If so, draw **quasiuniform** $\{U^*_{\rm Q}\}$ (Alg. 2). If not, superuniform $\{U^*_{\rm S}\}$ (Alg. 1) will suffice.

3. Eliminate floating point cancellation from the implementation, especially since $(1 - \{U^*_{\rm Q}\}) \overset{\text{cancellation}}{\longrightarrow} \{U^*_{\rm S}\}$.

These steps are relatively straightforward, and are demonstrated here for the exponential and normal distribution (which gives them direct application to the log-normal distribution). These and other distributions are implemented in the `pqRand` package [14], free C++ code with a minimal license.

IV. ACKNOWLEDGEMENTS

Special thanks to Zack Sullivan and Andrew Webster. This work was supported by the U.S. Department of Energy under award No. DE-SC0008347 and by Validate Health LLC.

[1] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[2] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[3] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic.* August 2008.

[4] Wikipedia. Condition number — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Condition%20number&oldid=767574928`, 2017. [Online; accessed 28-February-2017].

[5] Pierre L'Ecuyer and Jerry Banks (editor). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice (Ch. 4: Random number generation).* A Wiley-Interscience publication. Wiley, 1998.

[6] George Marsaglia, Arif Zaman, and Wai Wan Tsang. Toward a universal random number generator. *Statistics & Probability Letters*, 9(1):35–39, 1990.

[7] Mutsuo Saito and Makoto Matsumoto. A prng specialized in double precision floating point numbers using an affine transition. In Pierre L' Ecuyer and Art B. Owen, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2008*, pages 589–602, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] Python Software Foundation. random.py (`RECIP_BPF` uses 53 bits per float). `https://svn.python.org/projects/python/tags/r32/Lib/random.py`, 2017. [Online; accessed 28-February-2017].

[9] Free Software Foundation. random.h (line 165, `__detail::_Adaptor`, used by the generators in random.tcc). `https://gcc.gnu.org/onlinedocs/gcc-4.8.2/libstdc++/api/a01452_source.html`, 2017. [Online; accessed 28-February-2017].

[10] Taylor R. Campbell. Uniform random floats: How to generate a double-precision floating-point number in [0, 1] uniformly at random given a uniform random source of bits. `http://mumble.net/~campbell/2014/04/28/random_real.c`, 2014. [Online; accessed 5-April-2017].

[11] The Open Group. log1p, log1pf, log1pl - compute a natural logarithm. `http://pubs.opengroup.org/onlinepubs/009695399/functions/log1p.html`, 2004. [Online; accessed 5-April-2017].

[12] Free Software Foundation. random.tcc (line 1934, `normal_distribution::operator()`). `https://gcc.gnu.org/onlinedocs/gcc-4.8.2/libstdc++/api/a01453_source.html`, 2017. [Online; accessed 28-February-2017].

[13] Jean-Sebastien Roy. randomkit.c (line 599, `rk_gauss()`). `https://github.com/numpy/numpy/blob/master/numpy/random/mtrand/randomkit.c`, 2017. [Online; accessed 28-February-2017].

[14] Keith Pedersen. `pqRand`: The precise quantile flip-flop random package. `https://github.com/keith-pedersen/pqRand`, 2017.