Reconditioning your quantile function

Keith Pedersen*
Illinois Institute of Technology, Chicago, IL 60616

Monte Carlo simulations are an important tool for modern science because they can model highly nonlinear systems (like particle colliders and cellular membranes). Precision Monte Carlo results rely on precise random samples, which commonly employ the inversion method via a quantile function $Q(u)$ (either directly, or as the proposal distribution for rejection sampling). Yet the rapidly growing *scale* of these simulations makes the inversion method vulnerable to a loss of precision in a distribution's tails, where $Q(u)$ becomes *ill-conditioned*. This problem stems from limitations in floating point arithmetic which are often overlooked during implementation (e.g. in common C++ and Python libraries). Reconditioning $Q(u)$ requires extra care both when drawing a uniform variate and when using it inside $Q(u)$. An improved implementation is provided by pqRand, a free C++ and Python package.

INTRODUCTION

The inversion method samples from a probability distribution f via its quantile function $Q \equiv F^{-1}$, the inverse of f's cumulative distribution F [1, 2]. Q is used to transform a random sample from $U(0,1)$, the uniform distribution over the unit interval, into a random sample $\{f\}$;

$$\{f\} = Q\big(\{U(0,1)\}\big) \; . \tag{1}$$

This scheme is powerful because quantile functions are formally exact. But any real-world implementation will be *formally inexact* because: (i) a source of true randomness is generally not practical (or even desirable, if one seeks repeatability), while an algorithm supplying pseudo-randomness is never perfect and (ii) both u and $Q(u)$ will use finite-precision machine arithmetic. While much attention has been paid to the first defect, the second has been largely ignored in the literature. As a result, common implementations of inversion sampling lose precision in the tails of f.

The fact that no one has patched this leak means that the loss of precision must be subtle. Nevertheless, if follows from a striking disparity. Using a pseudo-random number generator (PRNG) with a ridiculously long period of $2^{19,937}$, one operates a sampling engine which can only output 2^{53} unique values — nowhere close to the theoretical maximum of 2^{1074} unique values. This is like firing a sniper rifle (the PRNG) from the hip. As Monte Carlo simulations continue to grow more complex, with ever increasing sample sizes, they *may* becoming sensitive to such poor, repetitive samples. Proving that they *are not* is incredibly difficult, so the best alternative is to use the most numerically stable sampling engine possible — provided that it is not prohibitively slow. Luckily, the scheme proposed in this paper is almost as fast as the one it replaces.

To isolate the loss of precision, we can examine the three independent steps of inversion sampling:

1. Generate random bits (i.i.d. coin flips) using a PRNG.
2. Convert those random bits into a uniform variate u from $U(0,1)$.
3. Plug u into $Q(u)$ to sample from f.

A major virtue of this method is that only step 3 depends on the actual distribution; the first two steps are totally generic. Of those, step 1 has been exhaustively studied [3–5], and is essentially a solved problem — when in doubt, use the Mersenne twister [5, 6]. On the other side of the method,

* kpeders1@hawk.iit.edu

step 3 has been validated using real analysis [1, 7], so that known quantile functions need only be translated into the high-quality math functions available on modern computers.

This leaves step 2, the oft neglected middle child. Perhaps it receives such little attention because it naïvely resembles a trivial coding task to bridge the gap between random bits and a real-valued Q. Yet computers cannot use real numbers, and neglecting this fact is dangerous. With this maxim in mind, this paper conducts a careful investigation of the inversion method from step 2 onward, the point at which machine arithmetic takes over. As a case study we use the exponential distribution, which describes the time between events in a Poisson process with rate λ (e.g. radioactive decay). The lessons learned there can then be applied to the inversion method in general.

Starting with step 3 of the inversion method, we find that Q is often numerically unstable in a distribution's tails. Setion I illustrates how to use the condition number to fix this problem. This provides a sound framework to understand where step 2 can go wrong, so that Sec. II can demonstrate why the standard algorithm for drawing uniform variates is incomplete. A new algorithm is proposed, then empirically validated in Sec. III by comparing the near-perfect sample from the PQRAND package to the sub-optimal samples obtained via standard C++ and Python.

I. Q ARE ILL-CONDITIONED, BUT THEY DON'T HAVE TO BE

Real numbers are not countable, so computers cannot represent them. Machine arithmetic is limited to a countable set like rational numbers $\mathbb{Q}$. The most versatile rational approximation of $\mathbb{R}$ are *floating point* numbers, or "floats" — scientific notation in base-two ($m \times 2^E$). The precision of floats is limited by P, the number of binary digits in their mantissa m, which means that every floating point operation adds relative rounding errors of order $\epsilon \equiv 2^{-P}$ [8]. The propagation of such errors makes floating point arithmetic formally inexact. In the worst case, subtle effects like cancellation can degrade the *effective* (or de facto) precision to just a handful of digits. Using floats with arbitrarily high P mitigates such problems, but is usually emulated in software — an expensive cure. Prudence usually restricts calculations to the largest precision widely supported in hardware, *binary64* ($P = 53$), commonly called "double" precision.

Limited P makes the intrinsic stability of a computation an important consideration; a result should not change dramatically when its input suffers from a pinch of rounding error. The numerical stability of a function $g(x)$ can be quantified via its condition number $C(g)$ — the relative change in $g(x)$ per the relative change in x [9]

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

When an $\mathcal{O}(\epsilon)$ rounding error causes x to increment to the next representable value, $g(x)$ will increment by $C(g)$ representable values. So when $C(g)$ is large (i.e. $\log_2 C(g) \to P$), $g(x)$ is *ill-conditioned* and imprecise; the tiniest shift in x will cause $g(x)$ to hop over an *enormous* number of values — values through which the real-valued function passes, and which are representable with floats of precision P, but which cannot be attained via the floating point calculation $g(x)$. The condition number should be used to avoid such catastrophes.

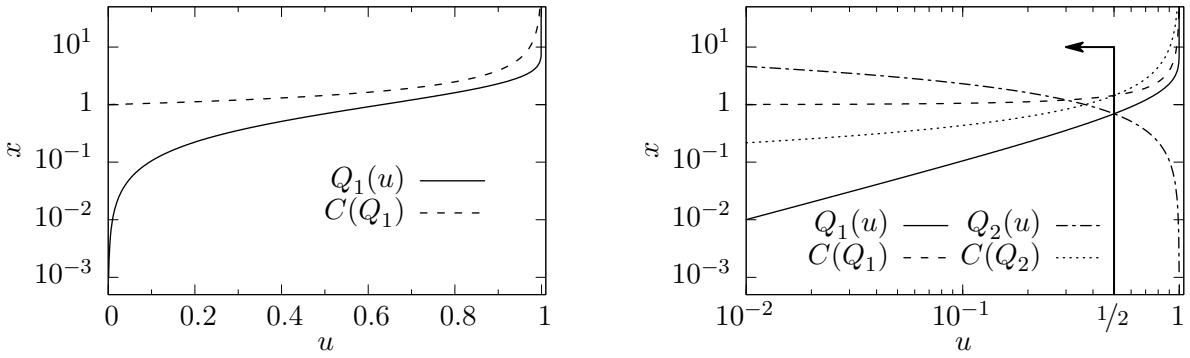
To determine if step 3 of the inversion method can stably sample the exponential distribution, we calculate the condition number of its quantile function;[1]

$$f(x) = \lambda \, e^{-\lambda x} \quad \longrightarrow \quad F(x) = 1 - e^{-\lambda x} \; ; \tag{3}$$

$$Q_1(u) = -\frac{1}{\lambda} \log(1-u) = -\frac{1}{\lambda} \texttt{log1p}(-u) \quad \longrightarrow \quad C(Q_1) = -\frac{u}{(1-u)\texttt{log1p}(-u)} \; . \tag{4}$$

[1] $\texttt{log1p}(x)$ is an implementation of $\log(1+x)$ which sidesteps an unnecessary floating point cancellation [10].

A high quality sample requires $C(Q_1) \leq \mathcal{O}(1)$ everywhere, but Fig. 1a clearly reveals that $C(Q)$ (dashed) becomes large as $u \to 1$. Why is Q_1 ill-condition there? According to Eq. 1, a function can become ill-conditioned when it is *steep* ($|g'/g| \gg 1$), and Q_1 (solid) is clearly steep at both $u = 0$ and $u = 1$. These are f's "tails" — a large range of sample space mapped by a thin, low probability slice of the unit interval. Yet in spite of its steepness, Q_1 remains well-conditioned throughout its small-value tail ($u \to 0$) because floats are denser near the origin — reusing the same set of mantissae, but with smaller exponents — and a denser set of u allows a more continuous sampling of a rapidly changing $Q_1(u)$. This extra density manifests as the singularity-softening factor of x in Eq. 1. Unfortunately, the same relief cannot occur as $u \to 1$, where representable u are not dense enough to accommodate Q_1's massive slope.



(a) (b)

FIG. 1: The $\lambda = 1$ exponential distribution $f(x) = e^{-x}$; (a) the quantile function Q_1 (solid) and its condition number (dashed) and (b) the "quantile flip-flop" — in $0 < u \leq 1/2$ each Q maps out half of f's sample space, a domain in which each remains well-conditioned.

Because Q_1 is ill-conditioned near $u = 1$, the large-x portion of its sample $\{f\}$ will be imprecise; many large-x floats which should be drawn will never be sampled by Q_1. This problem is not unique to the exponential distribution; it will occur whenever f with two tails, because one of those tails will be located near $u = 1$. Luckily, $U(0, 1)$ is perfectly symmetric across the unit interval, so transforming $u \mapsto 1 - u$ produces an equally valid quantile function;

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

The virtue of two symmetric Q is evident in Fig. 1b; for $u \leq 1/2$, each version is well-conditioned, with Q_1 handling the small-value tail ($x \leq$ median) and Q_2 the large-value tail ($x \geq$ median). Since the pair collectively and stably spans f's entire sample space, f can be sampled via the composition method; for each variate, randomly choose one version of the quantile function (to avoid a high/low pattern), then feed that Q a random u from $U(0, 1/2]$.

This "quantile flip-flop" — a randomized, two-Q composition split at the median — is a simple, general scheme to condition a quantile function which becomes unstable as $u \to 1$. It is also immediately portable to antithetic variance reduction, a useful technique in Monte Carlo integration where, for every $x = Q(u)$ one also includes the opposite choice $x' = Q(u')$ [11]. A common convention is $u' \equiv 1-u$, which has the useful property that the $\mathrm{cov}(x, x')$ can be negative, decreasing the variance of the integral estimate. Applying this technique to a quantile flip-flop is trivial; instead of randomly choosing Q_1 or Q_2 for each variate, always use both.

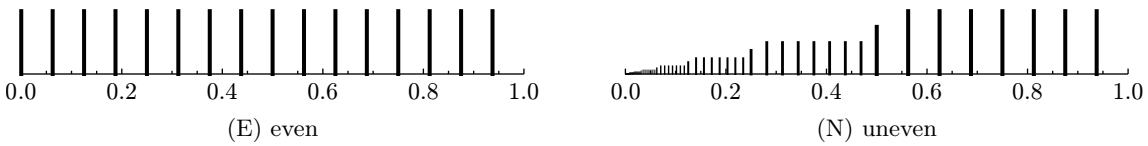
II. AN OPTIMALLY UNIFORM VARIATE IS MAXIMALLY *UNEVEN*

The condition number guided the development of the quantile flip-flop, a rather simple way to stabilize step 3 of the inversion method during machine implementation. Our investigation now proceeds to step 2 — sampling uniform variates. While steps 2 and 3 seem independent, we will find that there is an important interplay between them; a quantile function can be destabilized by sub-optimal uniform variates, but it can also wreck itself by mishandling *optimal* uniform variates.

The canonical method for generating $\{U(0,1)\}$ is Alg. 1; an integer is randomly drawn from $[0, 2^B)$, then scaled into the half-open unit interval $[0, 1)$ [2–4, 10, 12]. Using $B \leq P$ produces a sample space that is the pinnacle of uniformity — each u has the same probability, with a rigidly even spacing of 2^{-B} between each. Using $B = P$ gives the ultimate *even* sample $\{U_{\mathrm{E}}[0,1)\}$, as depicted in Fig. 2E (which uses a ridiculously small $B = P = 4$ to aide the eye). When $B > P$, line 4 will be forced to *round* many large j, as the mantissa of a is not large enough to store every j with full precision. As $B \to \infty$, this rounding saturates the floats available in $U[0,1)$, creating the *uneven* $\{U_{\mathrm{N}}[0,1)\}$ depicted Fig. 2N. This uneven sample space is still uniform because large u are more probable, absorbing more j from rounding (due to their coarser spacing).

Algorithm 1 Canonically draw a random FLOAT (with precision P) uniformly from $U[0,1)$

Require: $B \in \mathbb{Z}^+$ ▷ B must be a positive integer
1: $A \leftarrow \mathrm{FLOAT}(2^B)$ ▷ Convert $(j_{\max} + 1)$ to a float (an exact conversion for a power-of-two).
2: **repeat**
3: $\quad j \leftarrow \mathrm{RNG}(B)$ ▷ Draw B random bits and convert them into a integer from $U[0, 2^B)$.
4: $\quad a \leftarrow \mathrm{FLOAT}(j)$ ▷ Convert j to a FLOAT with precision P. Rounding may occur if $j > 2^P$.
5: **until** $a < A$ ▷ If $B > P$ and j rounds to A, the algorithm should not return 1.
6: **return** a/A



(E) even (N) uneven

FIG. 2: A visual depiction (using floats with $P = 4$ for clarity) of each possible u for (E) even $\{U_{\mathrm{E}}[0,1)\}$ ($B = P$) and (N) uneven $\{U_{\mathrm{N}}[0,1)\}$ ($B \to \infty$). The height of each tic indicates its relative probability, which is proportional to the length of the number line which rounds to it.

Algorithm 1 can generate uniform variates with are both even and uneven, but which is superior? What is the correct choice of B? Surprisingly, there seems to be no definitive answer in the existing literature, so we will have to work it out. We start by choosing the *even* uniform variate $\{U_{\mathrm{E}}[0,1)\}$ as the null hypothesis, for two obvious reasons: (i) Fig. 2E certainly *looks* more uniform and (ii) taking $B \to \infty$ does not seem practical. However, we will instantly find that this perfect evenness creates a subtle side effect — it forces all quantile functions to become ill-conditioned as $u \to 0$, even if they have an excellent condition number!

The condition number implicitly assumes that δx is vanishingly small. This is true enough for floats in general, for which $\delta x = \mathcal{O}(\epsilon\, x)$. But the even uniform variates have an *absolute* spacing of $\delta u = \epsilon$. To account for a non-vanishing δx, we define a function's *effective* precision

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

Like $C(g)$, a large effective precision $P^*(g)$ indicates an ill-conditioned calculation. For a generic floating point calculation, $\delta x = \mathcal{O}(\epsilon\, x)$, so P^* reverts back to the condition number; $P^*(g) \approx \epsilon\, C(g)$. But feeding even uniform variates into a quantile function gives $\delta u = \epsilon$, so

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

Calculating $P^*_{\mathrm{E}}(Q)$ for the quantile flip-flop of Fig. 1b reveals that both Q become imprecise as $u \to 0$, in stark contrast to the predictions of their condition numbers. This problem is not unique to the exponential distribution; it occurs whenever f has a tail (so that $|Q'/Q| \to \infty$ as $u \to 0$).

This reduced effective precision $P^*_{\mathrm{E}}(Q)$ makes the tails sparsely populated; there are many extreme values which $\{f\}$ will never contain, and those which it does will be sampled too often. $\{U_{\mathrm{E}}[0,1)\}$ is simply *too finite*; 2^P even uniform variates can supply no more than 2^P unique values from f. This implies that the *uneven* sample $\{U_{\mathrm{N}}[0,1)\}$ will restore quantile stability; its denser sample space near the origin will stabilize $P^*_{\mathrm{E}}(Q)$, and its small u will greatly expand the sample space of $\{f\}$, making the tails *far less* repetitive. Furthermore, recall that the uneven sample corresponds to the limit where $B \to \infty$ in Alg. 1. This is equivalent to sampling $U[0,1)$ from $\mathbb{R}$ and rounding to the nearest float, which is clearly the optimal floating point realization of a true $U[0,1)$.

The appeal of the uneven uniform variate also follows from information theory. The Shannon entropy of a sample space X counts the bits of information contained in each variate x;

$$H(X) = -\sum_i \Pr(x_i) \log_2 \Pr(x_i)\,. \tag{8}$$

The space of the even uniform variate ($B = P$) has $n \equiv \epsilon^{-1}$ members, each with probability ϵ, so

$$H_{\mathrm{E}} = -\sum_{i=1}^{n} \epsilon \log_2(\epsilon) = -\log_2 \epsilon = P\,. \tag{9}$$

This makes sense, since each even uniform variate emanates from a P-bit pseudo-random integer.

The sample space of the uneven $\{U_{\mathrm{N}}[0,1)\}$ contains every float in $[0,1)$, which is naturally partitioned into sub-domains $[2^{-k}, 2^{-k+1})$ with common exponent $-k$. Each domain comprises a fraction 2^{-k} of the unit interval, and the minimum exponent $-K$ depends on the floating point type (although $K \gg 1$ for *binary32* and *binary64*). The uneven entropy is then the sum of over sub-domains, each of which sums over the $n/2$ equiprobable mantissae[2]

$$H_{\mathrm{N}} = -\sum_{k=1}^{K} \left(\sum_{i=1}^{n/2} 2^{-k} (2\,\epsilon) \log_2 \left(2^{-k} (2\,\epsilon) \right) \right) = \sum_{k=1}^{K} 2^{-k}\,(P - 1 + k) \approx P + 1 \quad (\text{for } K \gg 1)\,. \tag{10}$$

One extra bit of information is clearly not a windfall. But this is the entropy of the *bulk* sample $\{U[0,1)\}$. What is the entropy of the sub-space $U[0, 2^{-k})$?

Rejecting all $u \geq 2^{-k}$ in the even sample $\{U_{\mathrm{E}}[0,1)\}$, we find that smaller u have less information

$$H_{\mathrm{E}}(k) = P - k \quad (\text{for } u < 2^{-k})\,. \tag{11}$$

This lack of information is inevitably mapped to the sample, consistent with our previous results. But for the *uneven* uniform variate, the sample space is *fractal*; each sub-space looks the same as the whole unit interval, so that $H_{\mathrm{N}}(k) = P + 1$ as before! Every u has maximal information, and a high-entropy input should give a high-precision sample.

---

[2] Ignoring that exact powers of two are 3/4 as probable, which makes no difference once $P \gtrsim 10$.

The effective precision $P^*_{\mathrm{E}}(Q)$ and Shannon entropy H both demonstrate that using an even uniform variate will decondition a quantile function, while switching to an uneven uniform variate will *recondition* it. But be warned — uneven variates are *very* delicate. Subtracting them from one mutates them back into *even* variates (with opposite boundary conditions);

$$1 - \{U_{\mathrm{N}}[0,1)\} \mapsto \{U_{\mathrm{E}}(0,1]\} \,. \tag{12}$$

This is floating point *cancellation*. The subtraction maps the very dense region of the uneven sample (near zero) to a region where floats are intrinsically sparse (near one). This requires rounding, which destroys the extra density of the uneven sample. Conversely, the sparse region of the uneven sample (near one) has no extra information to give when it is mapped near zero, and so remains sparse. This makes the use of `log1p` in Eq. 4 absolutely *essential* for obtaining a high-precision sample $\{f\}$, as we will empirically demonstrate in the next section.

III. PRECISION: LOST AND FOUND

In Sec. I we conditioned an intrinsically imprecise quantile function using a two-Q composition. Then in Sec. II we determined that an uneven uniform variate is required to *keep* Q well-conditioned. But how does one actually generate uneven uniform variates, since taking $B \to \infty$ is clearly not possible? It turns out to be rather simple, but we will save the technical details for the Appendix. First, we must prove that the pair of remedies proposed actually make a material difference. If indiscreet sampling decimates the precision of $\{f\}$, it should be quite evident in an experiment!

The quality of a real-world sample $\{f\}$ can be assessed via its Kullback-Leibler divergence [13]

$$D_{\mathrm{KL}}(\widehat{P}||\widehat{Q}) = \sum_i \widehat{P}(x_i) \log_2 \frac{\widehat{P}(x_i)}{\widehat{Q}(x_i)} \,. \tag{13}$$

D_{KL} quantifies the *relative* entropy between posterior distribution $\widehat{P}$ and prior distribution $\widehat{Q}$ (c.f. Eq. 8). The empirical $\widehat{P}$ is based on the count c_i, the number of times x_i appears in $\{f\}$

$$\widehat{P}(x_i) = c_i/N \tag{14}$$

(where N is the sample size). The *ideal* density $\widehat{Q}$ is obtained by mapping f onto floats, using the domain of real numbers $(x_{i,L}, x_{i,R})$ that round to x_i;

$$\widehat{Q}(x_i) = \int_{x_{i,L}}^{x_{i,R}} f(x)\,\mathrm{d}x = F(x_{i,R}) - F(x_{i,L}) \,. \tag{15}$$

D_{KL} does not sum terms where $\widehat{P}(x_i) = 0$ (i.e. x_i was not drawn), because $\lim_{x\to 0} x \log x = 0$.

The D_{KL} divergence is not a *metric* because it is not symmetric under exchange of $\widehat{P}$ and $\widehat{Q}$ [13]. Furthermore, while D_{KL} is frequently interpreted as the information *gained* when using distribution $\widehat{P}$ instead of $\widehat{Q}$, this is not true here. Consider a PRNG which samples from $\widehat{Q} = U(0,1)$ using floats with precision P. Now imagine that this generator is *so bad* that it always outputs $x = 0.5$. In this scenario, $D_{\mathrm{KL}} \approx P$, which is clearly the bits of precision *lost* by $\widehat{P}$ (the generator), since each x has zero information. In a less extreme case, since $\widehat{Q}$ is the most precise distribution possible given floats of precision P, any divergence denotes how many bits of precision were *lost*.

We sample from the $\lambda = 1$ exponential distribution via the inversion method. We use GNU's `std::mt19937` for our PRNG ($B = 32$), fully seeding its state from the computer's environmental noise (using `std::random_device`, which accesses `/dev/urandom`). Calculating D_{KL} requires

recording the count for each unique float, and an accurate D_{KL} requires a very large sample size ($N \gg P$, so that $\widehat{P} \to \widehat{Q}$ in the case of perfect agreement). To keep the experiment both exhaustive and tractable, and with no loss of generality, we use *binary32* ($P = 24$, or single precision). Since double precision is governed by the same IEEE 754 standard [14], and both types use library math functions with $\mathcal{O}(\epsilon)$ errors, the results for *binary64* will be identical.[3]

We test two standard implementations of exponential inversion sampling, one in C++ and one in Python. First we test GNU's `std::exponential_distribution`, a member of C++11's `<random>` suite which obtains its uniform variates from `std::generate_canonical` [15, 16]. Given our choice of PRNG, this is equivalent to calling Alg. 1 with $B = 32$ and $P = 24$. This results in a *partially* uneven sample $\{U_{\mathrm{P}}[0, 1)\}$, with $B - P = 8$ more bits of entropy than the totally even sample. These partially uneven variates are then fed into Q_1 (Eq. 4), *but only the version which does not use* `log1p`. As predicted by Eq. 12, this strips all extra entropy and converts a partially uneven sample ($B > P$) into a totally even one ($B = P$). As a result, the $\{f\}$ from GNU's C++11 should be exactly equivalent to the one obtained with Python using `numpy.random.exponential` [12], which feeds the totally even uniform variate ($B = P$) into Q_2.

Figure 3 depicts the precision lost by three samples using the same seed. The median is depicted at the center of the plot, with the tails (small u) near the edges. This expresses all results *as if* they were generated by a quantile flip-flop, even if the implementation used only one quantile function. This format becomes easier to understand by referring to the top axis, which shows the $x = Q(u)$ sampled by the various u. When sampling via `std::exponential_distribution` (▼), there is a clear and dramatic loss of precision the farther one gets from the median. The results agree precisely

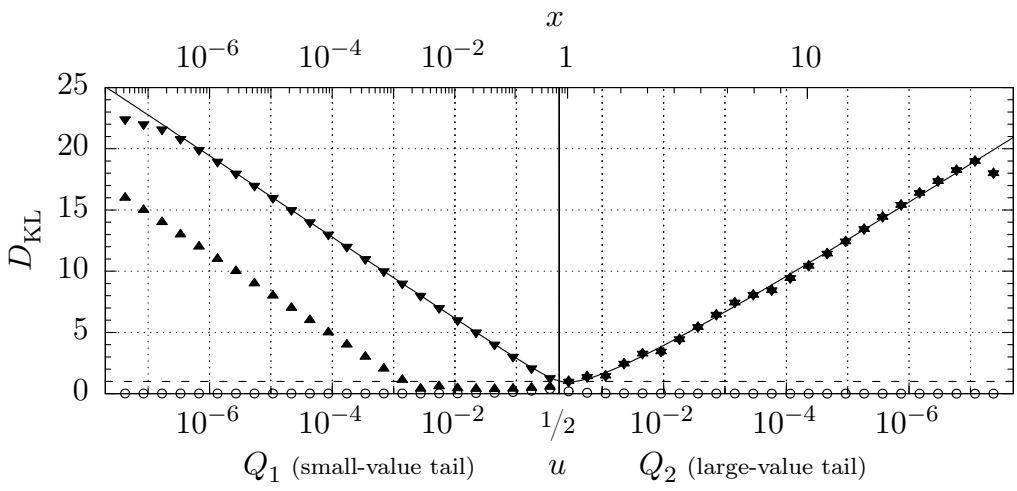


FIG. 3: Counting the bits of *precision lost* when sampling from the $\lambda = 1$ exponential distribution using (▼) `std::exponential_distribution`, (▲) `std::exponential_distribution` modified to use `log1p`, and (ↄ) the PQRAND package, which feeds an uneven $U_{\mathrm{N}}(0, 1/2]$ into the quantile composition of Fig. 1b. This "quantile flip-flop" — where Q_1 samples $x <$ median and Q_2 samples $x >$ median — is depicted with the median at the center ($u = 1/2$) and the tails towards the edges ($u \to 0$). The value of $x = Q(u)$ sampled at every u is shown on the top scale. Each point calculates D_{KL} for a domain $u \in [2^{-k}, 2^{-k+1})$, with a sample size of $N \approx 10^9$ for each point. The solid line is *not a fit*, but the loss of information predicted by Eq. 11 (scaled by $C(Q)$ because precision is lost at a slower pace when $Q'/Q < 1$). The dotted line marks a 1 bit threshold.

[3] A *binary64* experiment is tractable, just not exhaustive. Memory constraints require intricate simulation of tiny sub-spaces of the unit interval, to act as a representative sample of the whole.

with the entropy of an even variate H_{E} (solid line, Eq. 11); every time u becomes half as small (so that x is half as probable), one more bit of precision is lost. The exact same loss of precision occurs when $\{f\}$ is generated from `numpy.random.exponential`, since it stems from the lack of information in the even uniform variates $\{U_{\mathrm{E}}[0,1)\}$.

However, `std::exponential_distribution` could have done better. Recall that it drew a *partially uneven* uniform variate ($B = 32$, $P = 24$), then proceeded to spoil it by using Q_1 without `log1p`. Even though Q_1 is intrinsically well-conditioned as $u \to 0$, this real-valued property can only be realized with floats by using `log1p`; otherwise the $(1 - u)$ cancellation induces $\delta u = \epsilon$ everywhere, creating an unstable effective precision $P^*(Q)$ (see Eq. 7). To fix this issue, we enable `log1p` and regenerate the sample (▲). This permits Fig. 3 to isolate the two very different sources of imprecision identified in this paper. (i) Using `log1p`, Q_1 is allowed to be well-conditioned as $u \to 0$, so only the uniform variates themselves can spoil the low-value tail. Starting at the median and moving left, the *partially* uneven variate from `std::generate_canonical` maintains maximal precision for a time (adding eight extra bits, as expected). Yet since it obtains a fixed number of bits from the PRNG, it eventually runs out of entropy and loses precision. (ii) Conversely, Q_1 is intrinsically ill-conditioned as $u \to 1$, so it doesn't matter how good the uniform variate is; in the high-value tail, an ill-conditioned quantile function causes an immediate loss of precision.

The sample obtained from pqRand (○) corrects both of these mistakes by pairing the high-entropy, uneven uniform variates with a quantile flip-flop that is always well-conditioned. In stark contrast to the existing implementations, no precision is lost. Well almost none; 1/2 bit is lost near the median — where the composite Q is a tad unstable ($C(Q) \gtrsim 1$, see Fig. 1b) — but the performance in the tails is perfect. This result clearly demonstrates that the two remedies outlined in Secs. I and II fulfill their existential purpose, delivering the best sample possible with floats of precision P. Furthermore, this massive boost in quality arrives at ∼80% the speed of `std::exponential_distribution` (∼25 ns per variate on an Intel i7 @ 2.9 GHz). Surprisingly, it is *not* the uneven uniform variates which cause this slowdown (see the Appendix); it is the unpredictable branch when choosing Q, and the fact that `log1p` is a tad slower than `log`.

Equivalent results for any λ, as well as many other distributions (uniform, normal, log-normal, Weibull, Pareto) are available with pqRand, a free C++ and Python package available on GitHub [17]. pqRand uses optimized C++ to implement Alg. 2 (see the Appendix), with Python wrappers for fast scripting. Yet the usefulness of pqRand is not restricted to the rarefied set of distributions with analytic quantile functions. Rejection sampling gives access to *any* distribution $f(x)$, provided that one can more easily sample from the proposal distribution $g(x) \geq f(x)$. This implies that $g(x)$ is sampled via the inversion method. Since the eventual sample $\{f\}$ is merely a subset of the proposed sample $\{g\}$, a high-precision $\{f\}$ requires a high-precision $\{g\}$. Thus, pqRand supplies the tools neccessary to build a high-quality rejection sampling algorithm.

IV. CONCLUSION

Using the exponential distribution as a case study, we find two general sources of imprecision when sampling from f via the inversion method: (i) When f has two tails (two places where $Q'/Q \gg 1$), the quantile function Q becomes ill-conditioned as $u \to 1$. (ii) Drawing uniform random variates using the canonical algorithm (Alg. 1) gives too finite a sample space, forcing Q to become ill-conditioned as $u \to 0$. Both problems cause a huge loss of precision in the sample's tails, but the latter problem is especially nefarious since it is concealed by a good condition number $C(Q)$. Additionally, both problems exist in popular implementations of the inversion method (e.g. GNU's implementation of of C++11's `<random>` suite [15] and the `numpy.random` module for Python [12]), which eventually lose all precision, creating a unnecessarily repetitive sample.

To recondition Q, one must combine *uneven* uniform variates (Alg. 2) with a quantile flip-flop (a Q composition split at the median). One must also take care not to spoil the uneven variates through cancellation. These steps restore the sample $\{f\}$ to the full precision available with floats of precision P. Patching this leak is especially important in large, non-linear Monte Carlo simulations, which can draw *so many* numbers that they may be sensitive to this vulnerability. Since it is difficult to prove that a loss of precision in the tails is insignficant, one should use the most numerically stable components at every step in the simulation chain, provided they are not prohibitively slow. Fortunately, a fast implementation of reconditioned inversion sampling is provided by PQRAND, a free C++ and Python package available on GitHub [17].

V. ACKNOWLEDGEMENTS

Thanks to Zack Sullivan for his invaluable help and editorial suggestions, and to Andrew Webster for lowering the activation energy. This work was supported by the U.S. Department of Energy under award No. DE-SC0008347 and by Validate Health LLC.

Appendix: Drawing random, uneven $U_{\mathrm{N}}(0, 1/2]$

In Sec. II we saw that the best uniform variate is the uneven $\{U_{\mathrm{N}}[0, 1)\}$, obtained by taking $B \to \infty$ in Alg. 1. Since this will take *forever*, we must find an alternate scheme. A clue lies in the bitwise representation of the *even* uniform variate from Alg. 1, for which every $u < 2^{-k}$ has a reduced entropy $H_{\mathrm{E}} < P - k$ (see Eq. 11). When $B = P$, Alg. 1 draws an integer M from $(0, 2^P)$, then converts it to floating point. Inside the resulting float, the mantissa is stored as the integer M^*, which is just the original integer M with its bits shifted left until $M^* \geq 2^{P-1}$. This shift ensures that any $u < 2^{-k}$ always has at least k trailing zeroes in M^*; zeroes which contain no information. Filling this always-zero hole with new random bits restores maximal entropy.

Algorithm 2 Draw an *uneven* random FLOAT (with precision P) uniformly from $U(0, 1/2]$

Require: $B \geq P$
$n \leftarrow 1$ ▷ We return $j/2^n$. Starting at $n = 1$ ensures final scaling into $(0, 1/2]$.
repeat
  $j \leftarrow \mathrm{RNG}(B)$
  $n \leftarrow n + B$
until $j > 0$ ▷ Draw B bits from the infinite bit stream until we find at least one set bit.
if $j < 2^{P+1}$ **then** ▷ Require $S = P + 2$ significant bits.
  $k \leftarrow 0$
  repeat
    $j \leftarrow 2j$
    $k \leftarrow k + 1$
  until $j \geq 2^{P+1}$ ▷ Shift j's bits left until $S = P + 2$.
  $j \leftarrow j + \mathrm{RNG}(k)$ ▷ The leftward bit shift created a k-bit hole; fill it with new entropy.
  $n \leftarrow n + k$ ▷ Ensure that the leftward shift doesn't change u's course location.
end if
if j is even **then** $j \leftarrow j + 1$ **else** do nothing ▷ Make j odd to force proper rounding.
return $\mathrm{FLOAT}(j)/\mathrm{FLOAT}(2^n)$8 ▷ Round j to a FLOAT using R2N-T2E.

Given the domain required by a quantile flip-flop, Alg. 2 samples uneven $\{U_{\mathrm{N}}(0, 1/2]\}$ from the half-open, half-unit interval. It works by taking $B \to \infty$, but knowing that floating point arithmetic will truncate the infinite stream to P bits of precision. So as soon as the RNG returns the first 1

(however many bits that takes), only the next $P+1$ bits are needed to convert to floating point; P bits to fill the mantissa, and two extra bits for proper rounding. To fix u's coarse location, the first loop (line 2) finds the first non-zero bit. The next loop (line 8) requires $S = P + 2$ significant bits. If S is too small, j's bits are shifted left until the most significant (leftmost) bit slides into the $P + 2$ position. Then the vacated space on the right is filled with new random bits, and the leftward shift is factored into n, so that only u's fine location changes (enhancing precision while preserving uniformity). Finally, the integer is rounded into $(0, 1/2]$.[4]

Algorithm 2 needs two extra bits to maintain uniformity when j is converted to a float. With few exceptions, exact conversion of integers larger than 2^P is not possible because the mantissa lacks the necessary precision. And truncation won't work because $j < 2^n$, so Alg. 2 would never return $u = 1/2$, whereas a quantile flip-flop needs $u = 1/2$ to sample the exact median. Since Alg. 2 must be able to round up, it uses round-to-nearest, ties-to-even (R2N-T2E). Being the most numerically stable IEEE 754 rounding mode, R2N-T2E is the default choice for most operating systems.

Yet R2N-T2E is slightly problematic because Alg. 2 is truncating a theoretically infinite stream to finite significance S. There are going to be rounding ties, and when T2E kicks in, it will make *even* mantissae slightly more probable than odd ones, breaking uniformity. To defeat this bias, j is made odd. This creates a systematic tie-breaker, because an odd j is always closer to only one of the truncated options, without giving preference to the even option. This system only fails when $S = P+1$, and only the final bit needs removal. In this case, j *is* equidistant from the two options, and T2E kicks in. Adding a random buffer bit (requiring $S = P + 2$) precludes this failure.

An important property of Alg. 2 is that $u = 1/2$ is *half as probable* as its next-door neighbor, $u = \frac{1}{2}(1 - \epsilon)$. Imagine dividing the domain $[1/4, 1/2]$ into 2^{P-1} bins, with the bin edges depicting the representable u in that domain. Uniformly filling the domain with $\mathbb{R}$, each u absorbs a full bin of real numbers via rounding (a half bin to its left, a half bin to its right). The only exception is $u = 1/2$, which can only absorb a half bin from the left, which makes it half as probable. But recall that $\{U_{\mathrm{N}}(0, 1/2]\}$ is intended for use in a quantile flip-flop — a regular quantile function folded in half at the median ($u = 1/2$). Since *both* Q 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.

Finally, we must address the question of computational speed. Not only can Alg. 2 produce a sample which is unequivocally superior to `std::generate_canonical` (see Fig. 3), it does so at equivalent speed (∼5 ns per variate using MT19937 on an Intel i7 @ 2.9 GHz). This is possible because line 8 is rarely true (∼0.4% of the time when $N = 32$ and $P = 24$). Hence, the code to top-up entropy is rarely needed, and the main conditional branch is quite predictable. For most variates, the only extra overhead is verifying that $S \geq P + 2$, then making j odd; these steps take no time relative to the call to the PRNG and the R2N-T2E operation.

[1] L. Devroye, *Non-Uniform Random Variate Generation* (Springer-Verlag, 1986).

[2] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2nd Ed.): The Art of Scientific Computing* (Cambridge University Press, New York, NY, USA, 1992).

[3] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms* (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997).

[4] P. L'Ecuyer, in *Proceedings of the 29th Conference on Winter Simulation*, WSC '97 (IEEE Computer Society, Washington, DC, USA, 1997) pp. 127–134.

[5] P. L'Ecuyer and R. Simard, ACM Trans. Math. Softw. **33**, 22:1 (2007).

[6] M. Matsumoto and T. Nishimura, ACM Trans. Model. Comput. Simul. **8**, 3 (1998).

[4] We exclude zero from the output domain of Alg. 2 because, while theoretically possible, it will *never happen* (given a reliable RNG). Returning zero in *binary32* (single precision) would require drawing more than 128 all-zero bits in the first loop. Given a million cores drawing $B = 64$ every nanosecond, that would take $\mathcal{O}(10^{14})$ years.

[7] G. Steinbrecher and W. T. Shaw, European Journal of Applied Mathematics **19**, 87–112 (2008).
[8] D. Goldberg, ACM Comput. Surv. **23**, 5 (1991).
[9] B. Einarsson, *Accuracy and Reliability in Scientific Computing* (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2005).
[10] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++* (International Organization for Standardization, Geneva, Switzerland, 2012).
[11] D. Kroese, T. Taimre, and Z. Botev, *Handbook of Monte Carlo Methods* (Wiley, 2013).
[12] J. S. Roy, "randomkit.c," `https://github.com/numpy/numpy/blob/master/numpy/random/mtrand/randomkit.c` (2017), [Online; accessed 28-FEB-2017].
[13] A. Ben-David, H. Liu, and A. D. Jackson, JCAP **1506**, 051 (2015), arXiv:1506.07724 [astro-ph.CO].
[14] IEEE Task P754, *IEEE 754-2008, Standard for Floating-Point Arithmetic* (IEEE, 2008) p. 58.
[15] F. S. Foundation, "random.h," `https://gcc.gnu.org/onlinedocs/gcc-6.3.0/libstdc++/api/a01509_source.html` (2017), [Online; accessed 11-DEC-2017].
[16] Free Software Foundation, "random.tcc," `https://gcc.gnu.org/onlinedocs/gcc-6.3.0/libstdc++/api/a01510.html` (2017), [Online; accessed 11-DEC-2017].
[17] K. Pedersen, "pqRand," `https://github.com/keith-pedersen/pqRand` (2017).