Reconditioning your quantile function

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

Monte Carlo simulations are an important tool for modeling highly nonlinear systems (like particle colliders and cellular membranes), and high-quality random samples are their fuel. These samples are often generated via the inversion method (either directly, or to get the proposal variates for rejection sampling), which harnessing the mapping of the quantile function $Q(u)$. Yet the increasingly large samples size of these simulations makes them vulnerable to a flaw in the inversion method; $Q(u)$ is *ill-conditioned* in a distribution's tails, stripping precision from its sample. This flaw stems from limitations in machine arithmetic which are often overlooked during implementation (e.g. in popular C++ and Python libraries). The *robust* inversion method reconditions $Q(u)$ by carefully drawing and using the uniform variates u. This novel method is implemented 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 a deterministic pseudo-random number generator (PRNG) is never perfect and (ii) both u and $Q(u)$ use finite-precision machine arithmetic. The first defect has received the lion's share of attention, leaving the second largely ignored. As a result, common implementations of inversion sampling lose precision in the tails of f.

This leak must be subtle if no one has patched it. Nonetheless, the loss of precision commonly exceeds *dozens* of ULP (units in the last place) in a distribution's tails. Contrast this to library math functions (`sin`, `exp`), which are painstakingly crafted to deliver no more than *one* ULP of systematic error. And when the inversion method loses precision, it produces inferior, repetitive samples, to which Monte Carlo simulations *may* become sensitive as they grow more complex, drawing ever more random numbers. Proving they are *not vulnerable* is incredibly difficult, so the best alternative is to use the most numerically stable sampling scheme possible — provided that it is not too slow. Luckily, the improved method proposed here is nearly as fast as the original.

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 the distribution f.

The first two steps do not depend on f, so they are totally generic. Of them, 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, step 3 has been validated using real analysis [1, 7], so that known quantile functions need only be translated into computer math functions.

[*] kpeders1@hawk.iit.edu

This leaves step 2, the oft neglected middle child which, at first glance, looks like a trivial coding task to port random bits into a real-valued Q. Yet computers cannot use real numbers, and neglecting this fact is dangerous — using this as its central maxim, this paper conducts a careful investigation of the inversion method from step 2 onward. Section I begins by using the condition number to probe step 3, finding that a distribution's quantile function is numerically unstable in its tails. This provides a sound framework for Sec. II to uncover the flaw in the canonical algorithm for drawing uniform variates (step 2). The *robust* inversion method fixes both problems, and is empirically validated in Sec. III by comparing the near-perfect sample obtained from the pqRand package to the deficient samples obtained from standard C++ and Python tools.

I. Q ARE ILL-CONDITIONED, BUT THEY DO NOT 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 to P, the number of binary digits in their mantissa m, which forces relative rounding errors of order $\epsilon \equiv 2^{-P}$ upon every floating point operation [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 numerical catastrophes.

We now have a tool to uncover possible instability in the inversion method, specifically in its quantile function Q (step 3). As a case study, we can examine the exponential distribution (the time between events in a Poisson process with rate λ, like radioactive decay);[1]

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

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

A well-conditioned sample from the exponential distribution will require $C(Q_1) \le \mathcal{O}(1)$ everywhere, but Fig. 1a clearly reveals that $C(Q_1)$ (dashed) becomes large as $u \to 1$. Why is Q_1 ill-conditioned there? According to Eq. 1, a function can become ill-conditioned when it is *steep* ($|g'/g| \gg 1$), and

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

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.

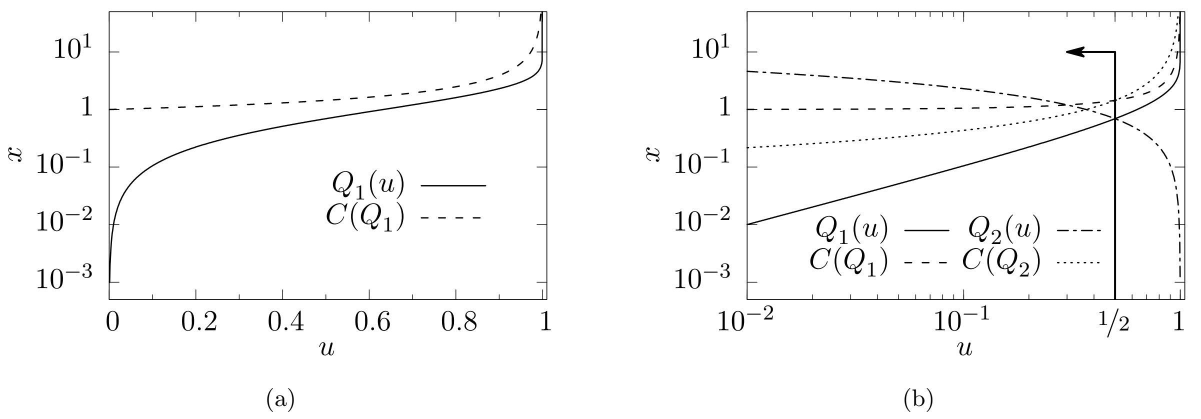

FIG. 1: The $\lambda = 1$ exponential distribution $f(x) = e^{-x}$; (a) the quantile function Q_1 (solid) and its condition number (dashed) and (b) the "quantile flip-flop" — in the domain $0 < u \le 1/2$, each Q maps out half of f's sample space while remaining well-conditioned.

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 sampled are skipped-over 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;

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

The virtue of two valid Q is evident in Fig. 1b; for $u \le 1/2$, each version is well-conditioned, with Q_1 sampling the small-value tail ($x \le$ median) and Q_2 the large-value tail ($x \ge$ 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 recondition 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 can create a negative covariance $\mathrm{cov}(x, x')$ that decreases the overall variance of the integral estimate. Generating antithetic variates with 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 uniform variates is Alg. 1 [2–4, 10, 12]; an integer is randomly drawn from $[0, 2^B)$, then scaled to a float in the half-open unit interval $[0, 1)$. Using $B \leq P$ produces a completely uniform sample space — each possible 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: $A \leftarrow$ FLOAT($2^B$)      ▷ Convert $2^B = (j_\mathrm{max} + 1)$ to a float (a power-of-two gets an exact conversion).
2: repeat
3:     $j \leftarrow$ RNG($B$)      ▷ Draw $B$ random bits and convert them into a integer from $U[0, 2^B)$.
4:     $a \leftarrow$ 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 $a/A$
```

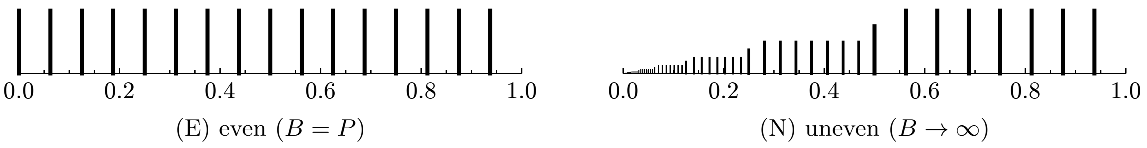

FIG. 2: A visual depiction (using floats with $P = 4$ for clarity) of each possible u for (E) even $\{U_\mathrm{E}[0, 1)\}$ and (N) uneven $\{U_\mathrm{N}[0, 1)\}$. The height of each tic indicates its relative probability, which is proportional to the width of the number-line segment which rounds to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

Depending on the choice of B, Alg. 1 can generate uniform variates which are either even or uneven, but which is better? There seems to be no definitive answers in the literature — which is likely why different implementations choose different B — so we will have to work one out here. 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 soon find that perfect evenness has 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 a generic float, whose $\delta x = \mathcal{O}(\epsilon\, x)$ is much small than x. But the even uniform variates have an *absolute* spacing of $\delta u = \epsilon$. To account for a finite 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

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

Calculating $P^*_\mathrm{E}(Q)$ for the quantile flip-flop of Fig. 1b says that *both* Q become ill-conditioned as $u \to 0$ and Q becomes steep, in stark opposition to their excellent condition numbers. That using even uniform variates will break a quantile flip-flop is a problem that is not unique to the exponential distribution; it occurs whenever f has a tail (so that $|Q'/Q| \to \infty$ as $u \to 0$).

The reduced effective precision $P^*_\mathrm{E}(Q)$ caused by even uniform variates creates sparsely populated tails; 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. This implies that the *uneven* sample $\{U_\mathrm{N}[0,1)\}$ will restore quantile stability, since its denser input space ($\delta u = \mathcal{O}(\epsilon u)$) will stabilize $P^*_\mathrm{N}(Q)$ near the origin. These small u expand the sample space of $\{f\}$ many times over, making its tails *far less* repetitive. And since uneven variates corresponds to the limit where $B \to \infty$ in Alg. 1, they are equivalent to sampling $U[0, 1-\epsilon)$ from $\mathbb{R}$ and rounding to the nearest float — the next best thing to a real-valued input for Q.

The virtue of using uneven uniform variates also follows from information theory. The Shannon entropy of a sample space X counts how many bits of information are conveyed by each variate x;

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

The sample space of the even uniform variates ($B = P$) has $n \equiv \epsilon^{-1}$ equiprobable members, 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 originates 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 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 more bit of information than H_E is clearly not a windfall. But H_E and H_N are the entropies of the *bulk* sample $\{U[0,1)\}$. What is the entropy of the tail-sampling 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 in even variates is inevitably mapped to the sample $\{f\}$, consistent with the exploding effective precision as $u \to 0$. But for *uneven* uniform variates, 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.

Both the effective precision $P^*(Q)$ and Shannon entropy H predict that using even uniform variates will force a well-conditioned quantile function to become ill-conditioned. Switching to uneven uniform variates will *recondition* it. But there is an important caveat; uneven variates are *very* delicate. Subtracting them from one mutates them back into *even* variates (with opposite boundary conditions);

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

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

This is floating point *cancellation.* The subtraction erases any extra density in the uneven sample, because it maps the very dense region (near zero) to a region where floats are intrinsically sparse (near one). Conversely, the sparse region of the uneven sample (near one) has no extra information to convey when it is mapped near zero, and remains sparse. This is why Q_1 (Eq. 4) *must* use `log1p`; cancellation will coerce *any* uniform variates into evenness, precluding a high-precision sample.

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 uneven uniform variates are required to *keep* Q well-conditioned. These two practices comprise the *robust* inversion method, whose technical details we have deliberately left for the Appendix because we have yet to prove that its changes 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]. And 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 $\widehat{Q} = U(0,1)$, but samples so poorly that it always outputs $x = 0.5$ (and thus emits zero information). Its $D_{\mathrm{KL}} \approx P$ is clearly the precision *lost* by $\widehat{P}$ (the generator). In less extreme cases, since $\widehat{Q}$ is the most precise distribution possible given floats of precision P, any divergence denotes how many bits of precision were *lost.*

Our experiments calculate D_{KL} for samples of the $\lambda = 1$ exponential distribution generated 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 GNU's `std::random_device`). 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 experiments 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]

The first implementation we test is GNU's `std::exponential_distribution`, a member of the C++11 `<random>` suite, which gets its uniform variates from `std::generate_canonical` [15,

[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.

16]. Given our PRNG, these uniform variates are equivalent to calling Alg. 1 with $B = 32$ and $P = 24$. This creates a *partially* uneven sample $\{U_\mathrm{P}[0,1)\}$, with an entropy buffer of $B - P = 8$ bits. GNU's implementation feeds these uniform variates into Q_1 (Eq. 4), *but without removing its cancellation* by using `log1p`. As predicted by Eq. 12, the cancellation strips any extra entropy from the partially uneven variates ($B > P$), converting then into even ones ($B = P$). Since both Python's `random.expovariate` [12] and Numpy's `numpy.random.exponential` [17] also uses Q_1 without `log1p`, they exhibit equivalent performance to GNU's `std::exponential_distribution`.

Figure 3 shows the precision lost by three samples using the same PRNG seed. It depicts all results *as if* they were generated by a quantile flip-flop (even if they were not), with the median at the center and the improbable tails (small u) near the edges. This format allows both tails to be probed in the same way, and becomes easier to understand by referring to the top axis, which shows the $x = Q(u)$ sampled by the various u. GNU's `std::exponential_distribution` (▼) exhibits a clear and dramatic loss of precision as variates gets farther from the median (and more rare). This imprecision agrees exactly with the limited information H_E contained in the even uniform variates it uses (Eq. 11, solid line); every time u becomes half as small (so that x is half as probable), one more bit of precision is lost. Importantly, since GNU's implementation uses only Q_1, the right half of the plot (large-value tail) actually depicts Q_1 with input $1/2 \le u < 1$. Nevertheless, since the loss of precision clearly stems from the lack of information in even uniform variates, an equally imprecise sample is generated by Python's `random.expovariate` and `numpy.random.exponential`.

But GNU's `std::exponential_distribution` could have done better; it drew *partially uneven* uniform variates ($B = 32$, $P = 24$), then spoiled them via cancellation. Enabling `log1p` in Q_1 and regenerating GNU's sample (▲) permits Fig. 3 to isolate the two sources of imprecision identified in Secs. I & II. (i) Using `log1p`, Q_1 is allowed to be well-conditioned as $u \to 0$, so only the uniform variates themselves can spoil the small-value tail. Moving left from the median, the *partially* uneven

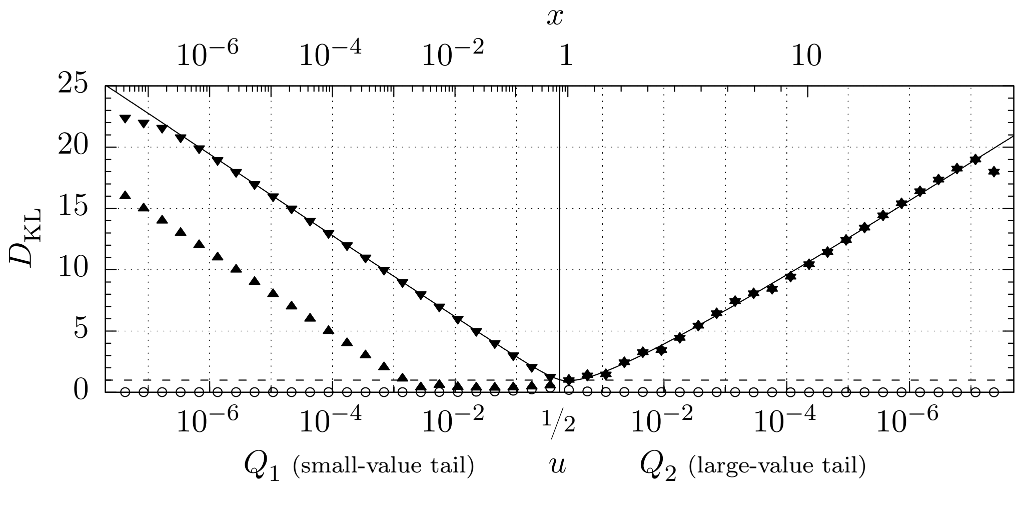

FIG. 3: The bits of *precision lost* (D_KL) when sampling the $\lambda = 1$ exponential distribution via (▼) GNU's `std::exponential_distribution`, (▲) GNU's implementation modified to use `log1p`, and (○) the robust inversion method (PQRAND), whose "quantile flip-flop" — where Q_1 samples $x <$ median 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 top axis is the variate $x = Q(u)$ sampled by every u. 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 precision predicted by Eq. 11 (scaled by $C(Q)$, because precision is lost at a slower pace when $Q'/Q < 1$). The dotted line is a 1 bit threshold.

variates maintain maximal precision . . . until their 8-bit entropy buffer runs dry. (ii) Conversely, Q_1 is intrinsically ill-conditioned for $u > 1/2$ in the large-value tail, so the quality of the uniform variates is irrelevant; an ill-conditioned quantile function causes an immediate loss of precision.

pqRand generates its sample (○) via the robust inversion method, pairing high-entropy, uneven uniform variates with a quantile flip-flop that is always well-conditioned. In stark contrast to the standard inversion method, almost no precision is lost. Half a 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 ideal. This result clearly demonstrates that the robust inversion method fulfills its existential purpose, delivering the best sample possible with floats of precision P. Furthermore, this massive boost in quality arrives at ∼80/100% the speed of GNU's `std::exponential_distribution` for *binary32*/*64* (∼30/40 ns per variate on an Intel i7 @ 2.9 GHz with GCC 6.3, optimization O2).

Similar samples for any rate λ, as well as many other distributions (uniform, normal, log-normal, Weibull, gamma) are available with pqRand, a free C++ and Python package hosted on GitHub [18]. pqRand uses optimized C++ to generate uneven uniform variates (see the Appendix), with Cython wrappers for fast scripting. Yet the usefulness of pqRand is not restricted to the rarefied set of distributions with analytic quantile functions; pqRand uses rejection sampling for its own normal and gamma distributions. Rejection sampling gives access to *any* distribution $f(x)$, provided that one can more easily sample from the proposal distribution $g(x) \geq f(x)$. Since the final sample $\{f\}$ is merely a subset of the proposed sample $\{g\}$, a high-precision $\{f\}$ requires a high-precision $\{g\}$ (i.e. obtained via the robust inversion method).

IV. CONCLUSION

Using the exponential distribution as a case study, we find two general sources of imprecision when sampling f via the inversion method: (i) When f has two tails (two places where $Q'/Q \gg 1$), the quantile function $Q(u)$ 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 $Q(u)$ to become ill-conditioned as $u \to 0$. Both problems can lose dozens of ULP of precision in a sample's tails, but the latter problem is especially nefarious since it is masked by a good condition number $C(Q)$. Both problems exist in popular implementations of the inversion method (e.g. GNU's implementation of C++11's `<random>` suite [15] and the `python.random` [12] and `numpy.random` [17] modules for Python), which produce unnecessarily repetitive samples that eventually lose all precision.

This paper introduces the *robust* inversion method, which reconditions Q by combining *uneven* uniform variates (Alg. 2, see Appendix) with a quantile flip-flop (a two-Q composition split at the median). These steps restore the sample $\{f\}$ to the full precision available with floats of precision P, but require extra care to not spoil the uneven uniform variates via cancellation. Switching to the robust inversion method is especially important for 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 has no side effects, 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 robust inversion sampling is provided by pqRand, a free C++ and Python package [18].

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 uneven uniform variates

In Sec. II we saw that the best uniform variates are uneven, obtained by taking $B \to \infty$ in Alg. 1. Since this will take *forever*, we must devlop 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 $H_{\mathrm{E}} = P - k$ (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 bit-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 will restore maximal entropy.

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

```
Require: $B \geq P$
 1: $n \leftarrow 1$          ▷ We return $j/2^n$. Starting at $n = 1$ ensures final scaling into $(0, 1/2]$.
 2: repeat
 3:     $j \leftarrow \mathrm{RNG}(B)$          ▷ Draw $B$ random bits and convert them into a integer from $U[0, 2^B)$.
 4:     $n \leftarrow n + B$
 5: until $j > 0$          ▷ Draw random bits from the infinite stream until we find at least one non-zero bit.
 6: if $j < 2^{P+1}$ then          ▷ Require $S \geq P + 2$ significant bits.
 7:     $k \leftarrow 0$
 8:     repeat
 9:         $j \leftarrow 2j$
10:         $k \leftarrow k + 1$
11:     until $j \geq 2^{P+1}$          ▷ Shift $j$'s bits left until $S = P + 2$.
12:     $j \leftarrow j + \mathrm{RNG}(k)$          ▷ The leftward bit shift created a $k$-bit hole; fill it with $k$ fresh bits of entropy.
13:     $n \leftarrow n + k$          ▷ Ensure that the leftward shift doesn't change $u$'s course location.
14: end if
15: if $j$ is even then $j \leftarrow j + 1$          ▷ Make $j$ odd to force proper rounding.
16: return FLOAT($j$)/FLOAT($2^n$)          ▷ 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 $B \to \infty$, yet knowing that floating point arithmetic will truncate the infinite bit-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 5) finds the first significant bit. The following conditional (line 6) requires $S \geq P + 2$ 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 (line 11). 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. Simple truncation won't work because $j < 2^{n-1}$, so Alg. 2 would never return $u = 1/2$, a value needed by a quantile flip-flop 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 bit stream to finite significance S. There are going to be rounding ties, and when T2E kicks in, it will

[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.

pick *even* mantissae over 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 $S \geq P + 2$) 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, making it half as probable. But recall that $\{U_{\text{N}}(0, 1/2]\}$ 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.

Not only can Alg. 2 produce better uniform variates than `std::generate_canonical` (see Fig. 3), it does so at equivalent computational speed. This is possible because line 6 is rarely true ($\sim$0.1% when $N = 64$ and $P = 53$), so 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, which take no time compared to the RNG and R2N-T2E operations.

[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).

[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] Python Software Foundation, "random.py," `https://github.com/python/cpython/blob/master/Lib/random.py` (2017), [Online; accessed 6-JAN-2018].

[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] Free Software 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] NumPy Developers, "distributions.c," `https://github.com/numpy/numpy/blob/master/numpy/random/mtrand/distributions.c` (2017), [Online; accessed 6-JAN-2018].

[18] K. Pedersen, "pqRand," `https://github.com/keith-pedersen/pqRand` (2017).