

Noise Generation Notes

Christian Covington

February 5, 2020

1 OVERVIEW

This document is a write-up of extra notes regarding the ways in which we sample noise in yarrow.

2 RANDOM NUMBER GENERATION

All of our random number generation involves uniform random sampling of bits via OpenSSL. We will take as given that OpenSSL is cryptographically secure, and talk about how it forms the basis for various functions in the library. When we refer to floating-point numbers, we specifically mean the IEEE 754 floating-point standard.

2.1 Uniform Number Generation

2.1.1 *sample_uniform(min : f64, max : f64)*

This is our basic uniform random number generator. We get 8 bytes (64 bits) from OpenSSL and interpret this as an unsigned 64-bit integer m . We divide m by $2^{63} - 1$ to get $m' \in [0, 1]$, and then rescale with both a multiplicative and additive factor to generate a value within the desired range, that is $m'' = m' \cdot (max - min) + min$, where min and max are user-provided data bounds. We then return m'' .

Note that not every floating-point number between min and max is able to be generated by this method. We will always get something of the form

$$m'' = \frac{m}{2^{63} - 1} * (max - min) + min,$$

where $m \in \{0, 1, \dots, 2^{63} - 1\}$. Each possible m'' occurs with probability $\frac{1}{2^{64}}$.

2.1.2 *sample_uniform_snapping()*

This is the uniform random number generator described in [Mir12]. Unlike *sample_uniform*, which takes in arbitrary min and max, this function always generates numbers in $[0, 1)$.¹

In this method, all normal floating-point numbers are representable, though they are not uniformly generated over the set of possible values. Instead, each is generated with probability relative to its unit of least precision.² That is, we generate numbers in $[\frac{1}{2}, 1)$ with probability $\frac{1}{2}$, numbers in $[\frac{1}{4}, \frac{1}{2})$ with probability $\frac{1}{4}$, etc. Within each precision band (the set of numbers with the same unit of least precision), numbers are sampled uniformly. We achieve this sample our exponent from a geometric distribution with parameter $p = 0.5$ and a mantissa uniformly from $\{0, 1\}^{52}$. Let e be a draw from *Geom*(0.5) (truncated such that $e \leq 1023$) and m_1, m_2, \dots, m_{52} be the bits of our mantissa. Then our function outputs u , where

$$u = (1.m_1m_2 \dots m_{52})_2 * 2^{-e}.$$

2.2 Biased Bit Sampling

Recall that we are taking as given that we are able to sample uniform bits from OpenSSL. For many applications, however, we want to be able to sample bits non-uniformly, i.e. where $\Pr(\text{bit} = 1) \neq \frac{1}{2}$. To do so, we use the *sample_bit* function.

2.2.1 *sample_bit(prob : f64)*

This function uses the unbiased bit generation from OpenSSL to return a single bit, where $\Pr(\text{bit} = 1) = \text{prob}$. I was introduced to the method for biasing an unbiased coin from a homework assignment given by Michael Mitzenmacher, and I later found a write-up online [here](#). We will give a general form of the algorithm, and then talk about implementation details.

Algorithm 1 Biasing an unbiased coin

- 1: $p \leftarrow \Pr(\text{bit} = 1)$
 - 2: Find the infinite binary expansion of p , which we call $b = (b_1, b_2, \dots)_2$. Note that $p = \sum_{i=1}^{\infty} \frac{b_i}{2^i}$.
 - 3: Toss an unbiased coin until the first instance of “heads”. Call the (1-based) index where this occurred k .
 - 4: **if** $b_k = 1$ **then**
 - 5: return 1
 - 6: **else**
 - 7: return 0
 - 8: **end if**
-

¹Of course, it could be scaled to arbitrary ranges through the same multiplicative and additive scaling as *sample_uniform*.

²The ULP is the value represented by the least significant bit of the mantissa if that bit is a 1.

Let's first show that this procedure gives the correct expectation:

$$\begin{aligned}
prob &= \Pr(bit = 1) \\
&= \sum_{i=1}^{\infty} \Pr(bit = 1 | k = i) \Pr(k = i) \\
&= \sum_{i=1}^{\infty} b_i \cdot \frac{1}{2^i} \\
&= \sum_{i=1}^{\infty} \frac{b_i}{2^i}.
\end{aligned}$$

This is consistent with the statement in Algorithm 1, so we know that the process returns bits with the correct bias. In terms of efficiency, we know that we can stop coin flipping once we get a heads, so that part of the algorithm has $\mathbb{E}(\#flips) = 2$.

The part that is a bit more difficult is constructing the infinite binary expansion of p . We start by noting that, for our purposes, we do not actually need an infinite binary expansion. Because p will always be a 64-bit floating-point number, we need only get a binary expansion that covers all representable numbers in our floating-point standard that are also valid probabilities. Luckily, the underlying structure of floating-point numbers makes this quite easy.

In the 64-bit standard, floating-point numbers in $[0, 1]$ are represented as

$$(1.m_1m_2 \dots m_{52})_2 * 2^{(e_1e_2 \dots e_{11})_2 - 1023}.$$

Then, our binary expansion is just our mantissa $(1.m_1m_2 \dots m_{52})_2$, with the radix point shifted based on the value of the exponent. We can then index into the properly shifted mantissa and check the value of the k th element.

3 TRUNCATION VS. CENSORING

Throughout our noise functions, we use the terms *truncated* and *censored*. Both are means of bounding the support of the noise distribution, but they are distinct.

Truncating a distribution simply ignores events outside of the given bounds, so all probabilities within the given bounds are scaled up by a constant factor. One way to generate a truncated distribution is via rejection sampling. You can generate samples from a probability distribution as you normally would (without any bounding), and reject any sample that falls outside of your bounds.

Censoring a distribution, rather than ignoring events outside of the given bounds, pushes the probabilities of said events to the closest event within the given bounds. One way to generate a censored distribution would be to generate samples from a probability distribution as you typically would, and then clamp samples that fall outside of your bounds to the closest element inside your bounds.

REFERENCES

- [Mir12] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 650–661, 2012.