Harvard Privacy Tools Project

Snapping Mechanism Implementation Notes

Christian Covington

August 6, 2019

1 Introduction

The implementation of the snapping mechanism utilizes a number of ideas described in Mironov (2012)[1] that, as far as I can tell, are not commonly seen in implementations of other DP algorithms. This document provides an overview of these ideas.

2 Mechanism Definitions

2.1 Laplace Mechanism

Let f be a function computed on a dataset D with sensitivity Δ_f and ϵ be the desired privacy parameter. Further, let $\lambda = \frac{\Delta_f}{\epsilon}$ the Laplace Mechanism is defined as:

$$M_L(D, f(\cdot), \lambda) = f(D) + Y$$

where $Y \sim Laplace(\lambda)$

2.2 Snapping Mechanism

Let B be a user-chosen quantity that reflects beliefs about reasonable bounds on $f(D)$ and Λ be the smallest power of two at least as large as λ. Using the same notation as above, the snapping mechanism is defined as:

$$M_S(D, f(\cdot), \lambda, B) = clamp_B\left(\lfloor clamp_B\left(f(D)\right) \oplus Y \rceil_\Lambda\right).$$

where $clamp_B(\cdot)$ restricts output to the interval $[-B, B]$ and $\lfloor \cdot \rceil_\Lambda$ rounds to the nearest multiple of Λ, with ties resolved toward $+\infty$. We will also make use of $\lfloor \cdot \rceil^*_\Lambda$, which rounds to the nearest multiple of Λ, with ties resolved toward $-\infty$.

[1] specifically in section 5.2

3 Sampling from the Laplace

Mironov presents the Laplace noise as

$$S \otimes \lambda \otimes LN(U^*)$$

where S is a uniform draw over $\{\pm 1\}$, $\lambda = \frac{\Delta_f}{\epsilon}$, $LN(\cdot)$ is the floating-point implementation of the natural logarithm with exact rounding, and U^* is the uniform distribution over $\mathbb{D} \cap (0,1)$[2] such that each double is output with proportion relative to its unit of least precision. $\oplus$ and $\otimes$ are the floating-point implementations of addition and multiplication, respectively.

Drawing S should be done with whichever source of randomness we end up wanting to use. The current Python implementation uses system-level randomness by utilizing an instance of the **random.SystemRandom** class.

We get U^* as Mironov suggests, by sampling 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 $Geom(0.5)$ and $m_1, m_2 \dots, m_{52}$ be the bits of our mantissa. Then we have

$$U^* = (1.m_1 m_2 \dots m_{52})_2 * 2^{1023-e}.$$

$LN(\cdot)$ must be implemented with exact rounding, which we define below.[3] Consider that for an arbitrary $x \in \mathbb{D}$ the natural log of x is not necessarily $\in \mathbb{D}$. Let $a < ln(x) < b$ where $a, b \in \mathbb{D}$ and $\nexists c \in \mathbb{D} : a < c < b$. Without loss of generality, assume that $|a - x| < |b - x|$, so that if we had infinite precision in calculating $ln(x)$ (but still had to output an element $\in \mathbb{D}$), we would output a. Many mathematical libraries do what is called *accurate-faithful* rounding, which means that in the scenario above our algorithm would output a with high probability. In an *exact rounding* paradigm, the algorithm outputs a with probability 1. You can read more about exact rounding in section 1.1 here. Section paper 2.1 of the paper just linked appeals to proofs from a set of papers that say you need 118 bits of precision, in the worst case, to calculate the logarithm with exact rounding. So, the current implementation calculates $LN(U^*)$ with 118 bits of precision.

Finally, we can choose to perform $\oplus$ and $\otimes$ with greater than normal precision if we so choose. For now, we are using the same 118 bits of precision for all the basic floating-point operations, using the assumption that exact rounding of basic arithmetic operations should require less precision than calculating the log. This should probably be made more rigorous at some point.

4 Implementation of $\lfloor \cdot \rceil_\Lambda$

The $\lfloor \cdot \rceil_\Lambda$ function takes an input an rounds it to the nearest multiple of Λ, where Λ is the smallest power of two greater than or equal to λ. There are multiple steps to this implementation that are worth explaining.

[2]the set of doubles $\in (0,1)$

[3]For all rounding, we assume that our goal is to round to the nearest number we are able to represent.

4.1 Finding Λ

The algorithm receives λ as input, but must find Λ itself. First, represent λ in its IEEE-754 64-bit floating-point format:

$$\lambda = (-1)^S (1.m_1 \dots m_{52})_2 * 2^{(e_1 \dots e_{11})_2 - 1023}.$$

We know $\lambda > 0$, so we know $S = 0$. Now, note that powers of two correspond exactly to the IEEE representations with $m_1 = \dots = m_{52} = 0$. If λ is already a power of two, then we simply return λ. Otherwise, we get the smallest power of two greater than λ by changing to mantissa to $(0 \dots 0)_2$ and increasing the exponent by 1. So, we have

$$\Lambda = \begin{cases} \lambda, & \text{if } m_1 = \dots = m_{52} = 0 \\ (1.0 \dots 0)_2 * 2^{(e_1 \dots e_{11})_2 - 1022}, & \text{if } \exists i : m_i \neq 0 \end{cases} \tag{4.1}$$

4.2 Rounding to nearest multiple of Λ

We now want to round our input x to the nearest multiple of Λ. We do so via a three-step process:

1. $x' = \frac{x}{\Lambda}$
2. Round x' to nearest integer, yielding x''
3. $\lfloor x \rceil_\Lambda = \Lambda x''$

We split the process into three steps because each step can be performed exactly (with no introduction of floating-point error) via manipulation of the IEEE floating-point representation.

4.2.1 Calculate $x' = \frac{x}{\Lambda}$

We can perform this division exactly because Λ is a power of two. Let $\Lambda = 2^m$ for some $m \in \mathbb{Z}$ and let x have the IEEE representation

$$x = (-1)^S (1.m_1 \dots m_{52})_2 * 2^{(e_1 \dots e_{11})_2 - 1023}.$$

Then we know that

$$x' = (-1)^S (1.m_1 \dots m_{52})_2 * 2^{(e_1 \dots e_{11})_2 - 1023 - m}.$$

We can rewrite $(e_1 \dots e_{11})_2 - m$ as $(f_1 \dots f_{11})_2$ and represent x' directly as its IEEE implementation:

$$x' = (-1)^S (1.m_1 \dots m_{52})_2 * 2^{(f_1 \dots f_{11})_2 - 1023}.$$

4.2.2 Round x' to nearest integer

Let $y = (f_1 \dots f_{11})_2 - 1023$. We present slightly different rounding algorithms based on the value of y. Note that we abuse notation a bit below; the repeating element notation $\bar{0}$ means to repeat the element (in this case, 0) until the rest of the larger section has been filled. For example, if the mantissa is 52 bits, then $101\bar{0}$ represents 101 followed by 49 trailing zeros.

Case 1: $y \geq 52$ If $y \geq 52$, then we know that only integers are representable at this scale, so there is no need to round.[4]

[4]See here for an explanation.

Case 2: $y \in \{0, 1, \ldots, 51\}$
We can think of multiplying by 2^y as shifting the radix point to the right y times. We can then write

$$x' = (-1)^S(1m_1 \ldots m_y.m_{y+1} \ldots m_{52})_2.$$

We know $m_1 \ldots m_y$ represent powers of two $\in \mathbb{Z}$ and $m_{y+1} \ldots m_{52}$ are powers of two $\notin \mathbb{Z}$. Specifically, m_{y+1} corresponds to $\frac{1}{2}$. So, we know to round up if $m_{y+1} = 1$ and down if $m_{y+1} = 0$.

Rounding up requires incrementing up by 1 the integral part of the mantissa $(m_1 \ldots m_y)$, and changing the fractional part $(m_{y+1}, \ldots, m_{52})$ to zeros. Note that there is an edge case here where $m_i = 1$ for all i in which the mantissa becomes $(\bar{0})_2$ and we instead increment the exponent. Rounding down requires maintaining the integral part of the mantissa and changing the fractional part to zeros.

Let $(m'_1 \ldots m'_y)_2 = (m_1 \ldots m_y)_2 + 1$ and $(f'_1 \ldots f'_{11})_2 = (f_1 \ldots f_{11})_2 + 1$. Then we get:

$$x'' = \begin{cases} (-1)^S(1.m'_1 \ldots m'_y\bar{0})_2 * 2^{(f_1 \ldots f_{11})_2 - 1023}, & \text{if } m_{y+1} = 1 \text{ and } \exists i : m_i = 0 \\ (-1)^S(1.\bar{0})_2 * 2^{(f'_1 \ldots f'_{11})_2 - 1023}, & \text{if } m_{y+1} = 1 \text{ and } \forall i : m_i = 1 \\ (-1)^S(1.m_1 \ldots m_y\bar{0})_2 * 2^{(f_1 \ldots f_{11})_2 - 1023}, & \text{if } m_{y+1} = 0 \end{cases} \tag{4.2}$$

Case 3: $y = -1$ We think of this case similarly to the beginning of Case 2 and shift the radix point to the left:

$$x'' = (-1)^S(0.1m_1m_2 \ldots)_2.$$

W know the bit directly to the right of the radix point is the implicit 1 in the IEEE representation, so we know we will round up. Rounding up always rounds to 1, so we know

$$x'' = (-1)^S(1.\bar{0})_2 * 2^{(0\bar{1})_2 - 1023}.$$

Case 4: $y < -1$ This is exactly the same as Case 3, except we know there are some number of leading zeros between the radix point and the implicit 1. Therefore, we always round down to 0 and get:

$$x'' = (-1)^0(1.\bar{0})_2 * 2^{(\bar{0})_2 - 1023}.$$

The notation should not be taken literally, as 0 has a special IEEE representation (all 0s) that does not quite follow the standard formula. We present the standard formally for the sake of consistency and comparison with the earlier cases.

4.2.3 Multiply x'' by Λ

Finally, we multiply x'' by Λ to get our desired result. This is effectively the opposite of Step 1, in that we add m to the exponent of x''. The only difference is that we now have to deal with the case in which $x'' = 0$, because $0 * 2^m \neq 0$ if you implement multiplication by 2^m through exponent addition.[5]

[5]This is due to the special representation of 0.