Bachelor thesis
Computer Science

Radboud University

Comparing the Performance of the Laplace and Staircase Mechanisms in Differential Privacy

Author:
Jelle Loman
s4573382

First supervisor/assessor:
prof. dr. Bart Jacobs
bart@cs.ru.nl

Second assessor:
dr. Jurriaan Rot
jrot@cs.ru.nl

July 29, 2018

Abstract

This thesis researches the claims of the makers of the staircase distribution for differential privacy that their staircase distribution performs better than the Laplace distribution for sampling noise values to achieve differential privacy.

After implementing the staircase sampling method in C++, their claims were confirmed, meaning that the staircase mechanism is faster than the Laplace mechanism for sampling differentially private noise values.

Contents

Chapter 1

Introduction

When Irit Dinur and Kobbi Nissim published their paper 'Revealing Information while Preserving Privacy' in 2003 [4], it became clear that a new approach to providing privacy needed to be taken. In their paper, Dinur and Nissim proved that it was impossible to maintain privacy when publishing information gathered from a private statistical database. These findings were directly contradictory to the semantic definitions of privacy that were given by Delanius in 1977 [2]. This definition—translated to the field of statistical databases—prescribes that access to a database shouldn't reveal information about an individual.

One new approach to privacy in statistical databases was introduced by Nissim, together with Cynthia Dwork, Frank McSherry, and Adam Smith in 2006 [10]. Their paper—'Calibrating Noise to Sensitivity in Private Data Analysis'—laid the foundation for the mathematical definition now known as *differential privacy*. This term was coined by Dwork the same year [5], and she defined its algorithmic foundations with Aaron Roth in 2014 [12]. She has sort of become the figurehead for the research field, writing many papers on the topic over the past decade [6, 7, 11, 8].

The idea behind differential privacy is quite simple; the presence or absence of one individual entry won't affect the result of a statistical query significantly. Because of this effect, individual privacy can be guaranteed when using cumulative queries on statistical databases through adding appropriately chosen mathematical noise.

A small part of the framework of differential privacy is choosing this mathematical noise that needs to be added. Typically, noise is generated by sampling from a probability distribution, which is what this thesis will focus on. The goal is to find the optimal mechanism for calculating noise by comparing two different probability distributions; the Laplace distribution [5], and the staircase distribution [14]. Performance is based on execution time—

that is, how long it takes for a sample to be drawn from the distribution.

This paper is divided into five further chapters. Chapter 2 discusses the background information on differential privacy, and introduces the chosen distributions in more detail. Chapter 3 will show the first endeavour into implementing these mechanisms into a test environment, and Chapter 4 will focus on the optimization of this implementation to be able to draw conclusions. Chapter 5 will discuss related work, and Chapter 6 will summarize the results from Chapters 3 and 4.

Chapter 2

Preliminaries

As with most theories in computer science, differential privacy has its roots in mathematics. Its algorithmic foundations were described by Dwork and Roth in 2014 [12], but its mathematical foundations had already been described before [5, 10]. This chapter will give the necessary background information by summarizing those findings, defining some key terms, and giving a toy example. At the end, the two noise generation mechanisms used in this research will be described in more detail.

2.1 What is privacy?

One of the most important parts of a privacy preserving technique is defining when an individual's privacy is violated. Finding a balance between utility—actually giving the user information they can work with—and privacy—making sure that people's sensitive information isn't leaked—has always been hard to get right. Swinging too much to either side negatively impacts one party, while maybe helping the other; openly working with sensitive data might be ethically questionable, but it could lead to better research results. Working with data that has been perturbed too much might lead to better privacy, but it probably won't yield the best test results.

In 1977, Tore Dalenius wrote the paper 'Towards a methodology for statistical disclosure control'. In it, he defines and theorizes what statistical disclosure is, and he considers the possibility of developing a methodology for statistical disclosure control [2]. According to Dalenius, statistical disclosure would be controlled if access to a statistical database discloses no additional information about an individual. It turns out however, that this type of privacy is unattainable. This is proved by Dwork and Noar [11], and the proof relies on the existence of *auxiliary information*.

Suppose an attribute like height is considered a highly sensitive piece of information. Revealing one's exact height would then constitute a privacy

breach. Now assume a database that contains the average heights of people from different nations. An adversary that has access to that database, as well as the auxiliary information that 'Eve is two inches shorter than the average Dutch woman' can determine Eve's height and as a result, breaches privacy. Anyone without access to the database doesn't learn Eve's exact height, meaning that no privacy is breached.

The most important part of this realization is that it doesn't matter whether Eve's height was used to determine the average height present in the database. The core principle of differential privacy is that someone's (lack of) participation in the database should not substantially impact risks they may experience. These risks can range from privacy risks to risks of reputation damage, but they should not be influenced negatively by participating.

Another thing to take note of, is that Dalenius's definition shows close resemblance to the concept of semantic security, introduced by Goldwasser and Micali a few years later [16]. In that environment however, the notion that only a negligible amount of information can be learned about the plaintext within a feasible amount of time, does hold up. This comes down to the requirement that useful results be generated by querying the data set. Working with a data set that is perturbed too much can negatively impact produced results, thus lowering usability. A cryptographic ciphertext doesn't have this usability constraint; it has, in fact, the opposite requirement. Learning as little as possible from the ciphertext as an adversary is one of its most important goals. Because of this usability requirement, a privacy theory like Dalenius defined won't be of help, but differential privacy will.

2.2 Definitions

The main idea behind differential privacy—an individual's participation in the database doesn't heavily impact their privacy—is represented using probabilities. The probability that a disclosure will arise is just as likely whether or not the individual participates in the database, up-to a multiplicative factor. This does not free the database from bad disclosures however, as those can still happen. It only assures that the presence or absence of a single data entry did not cause the bad disclosure. It also ensures an individual that no action or inaction could prevent or have prevented the disclosure.

2.2.1 Datasets and queries

Let a dataset or database $D = (d_1, \ldots, d_n)$ be a collection of records, or vector, from a universe $\mathcal{D}^n$, where each entry d_i represents information contributed by one individual [22]. To then represent the distance between datasets, the *Hamming distance* is used.

Definition 1. *The Hamming distance $d(D_1, D_2)$ between two databases is the number of entries on which they differ:*

$$d(D_1, D_2) = |\{i : D_{1_i} \neq D_{2_i}\}|$$

Two datasets are neighboring if they differ in a single participant's data, that is, when $d(D_1, D_2) = 1$.

To obtain information from a dataset, queries are performed on it. One of the most simple types of queries is the *counting query*. A counting query answers the question 'How many rows of the database satisfy some predicate p?'

Definition 2. *A counting query is a function $q : \mathcal{D} \to \mathbb{N}$ using the boolean predicate $p : d_i \to \{0, 1\}$ that maps each row to 0 if they don't satisfy the property, and to 1 if they do. q is then evaluated by*

$$q(D) = \sum_{i=1}^{|D|} p(d_i) \tag{2.1}$$

2.2.2 Differential privacy

An essential part of differential privacy is *randomization*. The intuition behind this is that an adversary can determine the value of a single entry in a dataset if they use a deterministic algorithm. This is done by executing the query on two neighboring datasets, and observing the outcome. Since the datasets differ at only 1 entry, the adversary can easily determine that entry's value, breaching privacy in the process.

To combat this, instead of deterministic algorithms, *randomized algorithms* are used [12].

Definition 3. *A randomized algorithm $\mathcal{K}$ with domain A and discrete range B is associated with a mapping $M : A \to \Delta(B)$. On input $a \in A$, the algorithm $\mathcal{K}$ outputs $\mathcal{K}(a) = b$ with probability $(M(a))_b$ for each $b \in B$.*

Here, $\Delta(B)$ is the so called *probability simplex* over B, which ensures that all of B's individual values sum up to 1, that is

$$\Delta(B) = \left\{ x \in \mathbb{R}^{|B|} : \forall_i, x_i \geq 0 \wedge \sum_{i=1}^{|B|} x_i = 1 \right\}$$

To now define differential privacy, [5] gives

Definition 4. *A randomized algorithm $\mathcal{K}$ with domain $\mathcal{D}$ gives ϵ-differential privacy if for all data sets D_1 and D_2 satisfying $d(D_1, D_2) = 1$, and all $S \subseteq$ Range($\mathcal{K}$),*

$$\Pr[\mathcal{K}(D_1) \in S] \leq \exp(\epsilon) \times \Pr[\mathcal{K}(D_2) \in S] \tag{2.2}$$

Note that this definition only refers to a general mechanism $\mathcal{K}$, not an actual implementation. Dwork does come with an idea however: adding noise to the output. This means that the privacy of the analysis hinges on *output perturbation*, where privacy is achieved by adding random noise that 'masks' the private information [22]. Releasing the result of a query function $f : \mathcal{D} \to \mathbb{R}^d$ on the dataset D will then first require the addition of some random variable N before publishing the answer $A(D) = f(D) + N$. To properly calibrate the noise to lead to a usable result, the *global sensitivity* Δ of the query function is used.

To define this, the ℓ_1 norm on $\mathbb{R}^d$ (denoted $\| \cdot \|_1$) is used as a distance metric. This norm measures the distance between two points by the sum of the absolute differences of their coordinates. To illustrate this, take two vectors $p = (p_1, \ldots, p_n)$ and $q = (q_1, \ldots, q_n)$. Their ℓ_1 norm is then defined as $\|p - q\|_1 = \sum_{i=1}^{n} |p_i - q_i|$. So if p and q were to only differ at the entry with index x, their ℓ_1 norm would be equal to $|p_x - q_x|$.

As the query function f has domain $\mathbb{R}^d$, and thus produces vectors of dimension d, the global sensitivity can be defined as follows [22].

Definition 5. *For $f : \mathcal{D} \to \mathbb{R}^d$, the global sensitivity of f is*

$$\Delta(f) = \max_{D_1, D_2} \|f(D_1) - f(D_2)\|_1 \tag{2.3}$$

for all D_1, $D_2 \in \mathcal{D}^n, d(D_1, D_2) = 1$.

That is, the global sensitivity of a query function is the difference between the output when executed on two databases, measured by the ℓ_1 norm. Applying this definition to a counting query q, the global sensitivity $\Delta(q) = 1$. This results from the fact that a counting query, when performed on two neighboring datasets, will show a difference of at most 1, namely when the differing row also differs in the property specified by q.

Using the global sensitivity of a counting query, if noise is calibrated to mask a difference of 1 between neighboring datasets' query outputs, it will be hard to determine an individual's participation, increasing privacy.

2.3 A Toy Example

To get a better impression of differential privacy in practice, it's perhaps best to examine a toy example. The structure of this toy example was taken

from an article written by Christine Task in XRDS magazine [26]. When she wrote the article, she was a Ph.D. candidate at Purdue University, doing research on differentially private social network analysis.

2.3.1 Introduction

Suppose Alice was conducting a survey about people's history with using drugs and their current salary. It would be nice to ensure the presence of some privacy on this data set, since (a past of) drug use can be a very privacy sensitive issue. Alice has already collected some data, and asks Bob to fill out the survey as well. The data Alice has already collected is in the form of individual survey results that are stored in a dataset. The dataset is comprised of the following columns:

- *Income.* One of the set $I = \{i_1, i_2, i_3\}$, where i_1 represents incomes lower than $\$25K$, i_2 those between $\$25K$ and $\$50K$, and i_3 those over $\$50K$.
- *Drug use.* One or more of the set $D = \{M, X, C, H, N\}$, where each represents the (past) use of a drug. M stands for marijuana, X for XTC, C for cocaine, H for heroine, and N for none of the above. Of course, if a respondent has answered 'N', they can't have any other options filled out.

If Alice now calls the counting query 2.3.1 on the dataset, a response like Result 2.3.1 is returned.

Query 2.3.1 Alice's counting query

SELECT *drug*,
COUNT(*drug*) for each $i \in I$
GROUP BY *drug*

From hereon there are two possible scenarios:

- Bob does not fill out the survey, leaving the current counts untouched.
- Bob fills out the survey, and increases the count for 'M' and 'X' in the column for the salary category i_3.

The goal of differential privacy in this research is to make it impossible for Alice to determine whether Bob has participated in her survey or not. This means that she cannot have access to the raw data, as that would allow her to compare the counts before and after she asked Bob.

Result 2.3.1 The data Alice has collected before asking Bob. The counts in ***bold italics*** are those that Bob can increase by participating in the survey

	i_1	i_2	i_3
M	56	67	***31***
X	23	11	***15***
C	31	9	22
H	4	2	7
N	63	110	141

2.3.2 The Laplace distribution

The second part is what this research focuses on: adding the noise. To do this, a noise generation mechanism—or NGM for short—is used. Because differential privacy relies on probabilities, this added noise has to be randomized, and the simplest way to achieve this is by using a probability distribution. A Laplace distribution is a traditional example that is often used to sample noise values [5, 9, 15]. It is a continuous, symmetrical probability distribution described by the probability density function

$$L(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x-\mu|}{b}\right) \tag{2.4}$$

The formula for the Laplace distribution contains two parameters that have an influence on the dispersion:

- $\boldsymbol{\mu}$, also known as the location parameter. This value corresponds to the x-coordinate of the center of the graph's peak.
- $\boldsymbol{b}$, also known as the scale parameter. Its value must be greater than zero, and it affects the steepness of the curve. The lower its value, the more the area under the curve is centered around the peak. See Figure 2.1 for the impact the scale parameter has on the graph. The red line shows the Laplacian distribution when $b = 1$, the blue line for $b = 2$, and the green line for $b = 4$.

The Laplace distribution is a so called *continuous probability distribution*, meaning that the probability of the random variable X falling into a given interval $[a, b]$ is given by the integral

$$\mathrm{P}[a \leq X \leq b] = \int_a^b f(x)\ dx$$

where $f(x)$ is the probability density function that describes the distribution (in the case of the Laplace distribution that would be Equation 2.4). Note

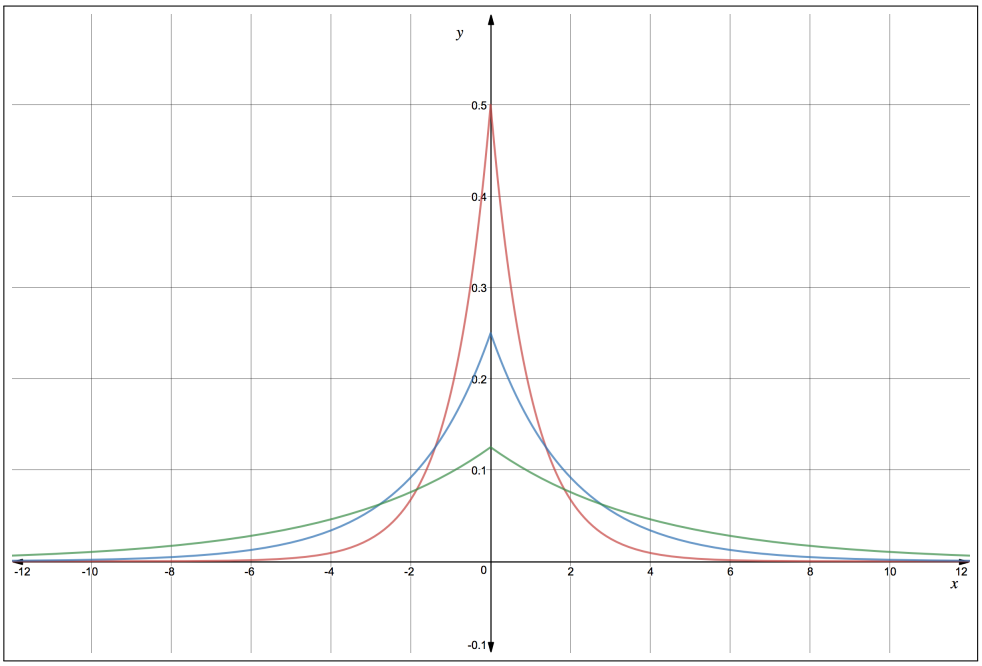

Figure 2.1: The Laplace distribution for $b = 1$, $b = 2$, $b = 4$

that this means that the probability of X assuming any definite point is 0, as the integral $\int_a^a f(x)\ dx$ evaluates to 0 by default.

In the case of differential privacy, this random variable X is the noise that will be added to the true answer. To get an idea of the distribution of noise values, suppose a Laplace distribution with $\mu = 0$ and $b = 2$. The chance that a noise value is then in the interval $[-\frac{1}{2}, \frac{1}{2}]$ is equal to

$$\int_{-\frac{1}{2}}^{\frac{1}{2}} L(x|\mu = 0, b = 2)\ dx =$$

$$\int_{-\frac{1}{2}}^{\frac{1}{2}} \frac{1}{4} \exp\left(-\frac{x}{2}\right)\ dx =$$

$$\frac{1}{4}\left[-2 \cdot \exp\left(-\frac{x}{2}\right)\right]_{-\frac{1}{2}}^{\frac{1}{2}} =$$

$$\frac{1}{4}\left(-2 \cdot e^{-\frac{1}{4}} + 2 \cdot e^{\frac{1}{4}}\right) = \frac{\sqrt[4]{e}}{2} - \frac{1}{2\sqrt[4]{e}} \approx 0.253$$

This means that there is a chance of roughly 25% that the sampled noise value is in the interval $[-\frac{1}{2}, \frac{1}{2}]$. That leaves a 75% chance the perturbation of the true answer will be greater.

2.3.3 Sampling

To use the Laplace distribution for actually sampling random values, *inverse transform sampling* is used. Inverse transform sampling is a method for generating numbers from a probability distribution using its *cumulative distribution function* [3].

The cumulative distribution function, or CDF, of a random variable X is given by

$$F(x) = \mathrm{P}(X \leq x)$$

where the evaluation of F at x will give the probability that X takes on a value less than or equal to x. To determine the CDF of X, the integral of its probability density function from the lowest end of its domain up to x must be calculated. To do this for the Laplace distribution $L(x)$, whose domain is $\mathbb{R}$, let's distinguish two cases:

$$L(x) = \begin{cases} \frac{1}{2b}\exp(-\frac{\mu - x}{b}), & \text{if } x < \mu \\ \frac{1}{2b}\exp(-\frac{x - \mu}{b}), & \text{if } x \geq \mu \end{cases} \tag{2.5}$$

Integrating the first case gives

$$\frac{1}{2b}\int_{-\infty}^{x} \exp\left(-\frac{\mu - t}{b}\right)\, dt =$$
$$\frac{1}{2b}\left[b \cdot \exp\left(\frac{t - \mu}{b}\right)\right]_{-\infty}^{x} =$$
$$\frac{1}{2b}\left(b \cdot \exp\left(\frac{x - \mu}{b}\right) - b \cdot \exp\left(\frac{-\infty - \mu}{b}\right)\right) =$$
$$\frac{1}{2b}\left(b \cdot \exp\left(\frac{x - \mu}{b}\right)\right) = \frac{1}{2} \cdot \exp\left(\frac{x - \mu}{b}\right)$$

To integrate the second case, we know that $\mu \leq x$, meaning that the integral can be split into two parts: from $-\infty$ to μ, and from μ to x. This reduces work, because we know the integral from $-\infty$ to μ is equal to $\frac{1}{2}$, as μ splits the distribution in half. This gives

$$\frac{1}{2b}\int_{-\infty}^{x} \exp\left(-\frac{t - \mu}{b}\right)\, dt =$$
$$\frac{1}{2} + \frac{1}{2b}\int_{\mu}^{x} \exp\left(-\frac{t - \mu}{b}\right)\, dt =$$
$$\frac{1}{2} - \frac{b}{2b}\left[\exp\left(-\frac{t - \mu}{b}\right)\right]_{\mu}^{x} =$$
$$\frac{1}{2} - \frac{1}{2}\left(\exp\left(-\frac{x - \mu}{b}\right) - \exp\left(-\frac{\mu - \mu}{b}\right)\right) =$$
$$\frac{1}{2} - \frac{1}{2}\left(\exp\left(-\frac{x - \mu}{b}\right) - 1\right) = 1 - \frac{1}{2}\exp\left(-\frac{x - \mu}{b}\right)$$

To sum up, the cumulative distribution $F_L(x)$ is

$$F_L(x) = \begin{cases} \frac{1}{2} \cdot \exp\left(\frac{x-\mu}{b}\right), & \text{if } x < \mu \\ 1 - \frac{1}{2}\exp\left(-\frac{x-\mu}{b}\right), & \text{if } x \geq \mu \end{cases} \tag{2.6}$$

Turning back to inverse transform sampling, the following steps have to be taken to sample a random value from the probability distribution [3]:

1. Determine $F_L^{-1}(x)$, so that $F_L(F_L^{-1}(x)) = x$.
2. Generate a random variable u, uniformly distributed on the interval $[0, 1]$.
3. The noise N is then equal to $F_L^{-1}(u)$.

The first step of this method requires F_L^{-1}, the inverse of the cumulative distribution function F_L. To determine this, it must hold that $F(F_L^{-1}(a)) = a$. This gives the first case

$$\begin{aligned} \frac{1}{2}\exp\left(\frac{F_L^{-1}(a)-\mu}{b}\right) &= a \\ \frac{F_L^{-1}(a)-\mu}{b} &= \ln(2a) \\ F_L^{-1}(a) &= b \cdot \ln(2a) + \mu \end{aligned}$$

and the second case

$$\begin{aligned} 1 - \frac{1}{2}\exp\left(-\frac{F_L^{-1}(a)-\mu}{b}\right) &= a \\ -\frac{F_L^{-1}(a)-\mu}{b} &= \ln(-2(a-1)) \\ F_L^{-1}(a) &= -b \cdot \ln(2-2a) + \mu \end{aligned}$$

That means $F_L^{-1}(x)$ can be defined as

$$F_L^{-1}(x) = \begin{cases} b \cdot \ln(2x) + \mu, & \text{if } x < 0.5 \\ -b \cdot \ln(2-2x) + \mu, & \text{if } x \geq 0.5 \end{cases} \tag{2.7}$$

Note that the domain of $F_L^{-1}(x)$ is $(0, 1)$, coinciding with the random variable u that determines the noise.

2.3.4 Adding the Noise

Bob has told Alice that he has made his decision, leaving Alice with research to do. Of course she doesn't know if Bob has filled out her survey, she can only query the dataset using the counting query q_{M3} that counts the number of respondents whose income level is i_3 and who have used marijuana before. In stead of directly receiving $A = q_{M3}(D)$, she receives $A = q_{M3}(D) + N$, where N is sampled from the Laplace distribution, parameterized by ϵ and $\Delta(q_{M3})$.

Previous research [10] has shown that a Laplace distribution with $\mu = 0$ and $b = \frac{\Delta(q_{M3})}{\epsilon}$ will result in ϵ-differential privacy. $\Delta(q_{M3})$ is equal to 1, as it's a simple counting query. The optimal value of ϵ obviously depends on the line Alice wants to walk between privacy and output usability. There are some heuristic choices [20, 17], but for this example $\epsilon = 0.5$ will provide a good balance. This gives $b = \frac{1}{0.5} = 2$.

Combining all of this together, sampling a random noise value from a Laplace distribution with $\mu = 0$ and $b = 2$ will be done by

1. Determining $F_L^{-1}(x)$, so that $F_L(F_L^{-1}(x)) = x$. See equation 2.7.
2. Generating a random number $u \in (0, 1)$. For this example, say $u = 0.22$.
3. Calculate $F_L^{-1}(u) = 2 \cdot \ln(0.44) + 0 \approx -1.642$. This means that the noise value $N = -1.642$.

If Alice now receives the response $A = q_{M3}(D) + N = 31 + -1.642 = 29.358$, she won't be able to reliably tell whether Bob has participated, because for all she knows, the response was the result of adding -2.642 to 32.

2.4 The Two NGMs

This research will focus on two different noise generation mechanisms that can achieve ϵ-differential privacy: the Laplace mechanism, as discussed before, that samples noise values from a Laplace distribution parameterized by the sensitivity of the query function and ϵ. The second NGM is the so called *staircase mechanism*, that also takes ϵ and Δ as parameters.

2.4.1 The Laplace Mechanism

The Laplace mechanism is one of the most used techniques for adding noise to a true answer [7, 10]. An example of its use, together with its definitions was given in section 2.3. It is defined by the probability density function

$$L(x|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|x - \mu|}{b}\right)$$

where μ is the location parameter, and b is the scale parameter.

2.4.2 The Staircase Mechanism

In 2013, Quan Geng and Pramod Viswanath published their paper 'The optimal mechanism in differential privacy' [15]. In it, they introduce a new type of noise generation mechanism that they dub *'the staircase mechanism'*. According to their research, the new staircase mechanism can replace the Laplace mechanism in any instance, while outperforming it as well. They have since done more research on the topic, solidifying their findings [14].

Geng and Viswanath define the probability density function of the staircase distribution with the case distinction

$$S(x|\gamma) = \begin{cases} a(\gamma), & x \in [0, \gamma\Delta] \\ e^{-\epsilon}a(\gamma), & x \in [\gamma\Delta, \Delta) \\ e^{-k\epsilon} \cdot S(x - k\Delta|\gamma), & x \in [k\Delta, (k+1)\Delta) \text{ for } k \in \mathbb{N} \\ S(-x|\gamma), & x < 0 \end{cases} \tag{2.8}$$

where

$$a(\gamma) \triangleq \frac{1 - e^{-\epsilon}}{2\Delta(\gamma + e^{-\epsilon}(1 - \gamma))} \tag{2.9}$$

The function $a(\gamma)$ exists to make sure that integrating $S(x|\gamma)$ over all real numbers results in a value of 1; that is, the area under the curve must be equal to 1. Furthermore, γ controls the shape of the distribution, and its optimal setting is $\frac{\sqrt{\alpha}}{1+\sqrt{\alpha}}$, with $\alpha = e^{-\epsilon}$ [1]. Δ is the sensitivity of the query function (see Definition 5).

To get a better idea of this distribution, it is plotted against both the Laplace and the Gaussian distribution in Figure 2.2. The privacy factor ϵ was set to 0.5, and the sensitivity of the query function was equal to 1.

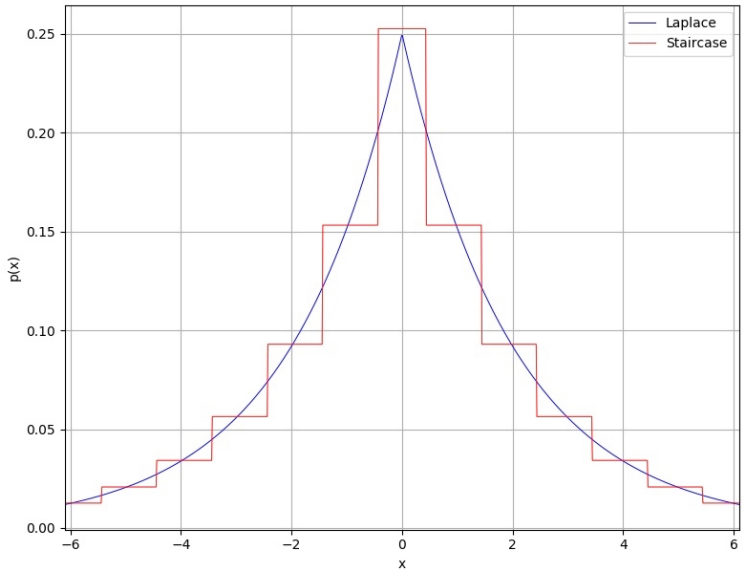

Figure 2.2: The staircase distribution in red versus the Laplace distribution in blue. ($\epsilon = 0.5, \Delta = 1 \rightarrow \alpha \approx 0.61, \gamma \approx 0.44$)

Chapter 3

Initial Research

The research portion of this thesis will consist of an implementation of the aforementioned differentially private mechanisms, after which an analysis of their performance will be made.

3.1 Programming Environment & Hardware

To research the optimality of the three noise generation mechanisms, a research environment must first be set up.

Python was used as the main programming language, because of its high level of extendability and ease of use. To keep everything up-to-date, the most recent stable release of Python was used, in this case Python 3.6.5.[1] For the IDE, I chose PyCharm Community Edition[2], an Apache 2 licensed developing environment made by JetBrains.

Because the calculations that probability density functions require aren't that intensive, no external computing power was needed beyond my laptop. That means all calculations were done on a 2015 MacBook Pro with a 2.7GHz dual-core Intel Core i5 processor and 8GB of RAM.

3.2 The Data Set

Differential privacy wouldn't exist if there wasn't data that needed to be privatized. A classic data set that is used for classification in machine learning is the 'Adult Data Set' from the UCI Machine Learning Repository.[3] According to the website, its purpose is to predict whether an individual's income exceeds $50K per year based on a collection of their attributes. There are 14 attributes, and they're listed in Table 3.1, together with their possible values. It has 48,842 individual entries, and the data was extracted from

[1] `https://blog.python.org/2018/03/python-365-is-now-available.html`
[2] `https://www.jetbrains.com/pycharm/`
[3] `https://archive.ics.uci.edu/ml/datasets/adult`

the 1994 Census database by Barry Becker. A database like this is ideal for counting queries, as you can simply query the data set for all rows whose attributes meet a certain mask.

Attribute	Type	Description
`age`	integer	{`17, ..., 90`}
`workclass`	categorical	{?, State-gov, Self-emp-not-inc, Private, Federal-gov, Local-gov, Self-emp-inc, Without-pay, Never-worked}
`fnlwgt`	integer	Final sampling weight
`education`	categorical	{1st-4th, 5th-6th, 7th-8th, 9th, 10th, 11th, 12th, Assoc-acdm, Assoc-voc, Bachelors, Doctorate, HS-grad, Masters, Preschool, Prof-school, Some-college}
`education-num`	integer	{`1, ..., 16`} (similar to `education`)
`marital-status`	categorical	{Divorced, Married-AF-spouse, Married-civ-spouse, Married-spouse-absent, Never-married, Separated, Widowed}
`occupation`	categorical	{?, Adm-clerical, Armed-Forces, Craft-repair, Exec-managerial, Farming-fishing, Handlers-cleaners, Machine-op-inspct, Other-service, Priv-house-serv, Prof-specialty, Protective-serv, Sales, Tech-support, Transport-moving}
`relationship`	categorical	{Husband, Not-in-family, Other-relative, Own-child, Unmarried, Wife}
`race`	categorical	{Amer-Indian-Eskimo, Asian-Pac-Islander, Black, Other, White}
`sex`	categorical	{Female, Male}
`capital-gain`	integer	{`0, ..., 99999`}
`capital-loss`	integer	{`0, ..., 99999`}
`hours-per-week`	integer	{`0, ..., 99`}
`native-country`	categorical	{?, Cambodia, Canada, China, Columbia, Cuba, Dominican-Republic, Ecuador, El-Salvador, England, France, Germany, Greece, Guatemala, Haiti, Holand-Netherlands, Honduras, Hong, Hungary, India, Iran, Ireland, Italy, Jamaica, Japan, Laos, Mexico, Nicaragua, Outlying-US(Guam-USVI-etc), Peru, Philippines, Poland, Portugal, Puerto-Rico, Scotland, South, Taiwan, Thailand, Trinadad&Tobago, United-States, Vietnam, Yugoslavia}
`income`	categorical	{`'<=50k', '>50k'`}

Table 3.1: The attributes in the data set with their type and possible value

3.3 Querying the Data Set

Before any sort of noise can be added to a true answer, the true answer must be determined. The easiest way to query a database is by using SQL as a

query language. The only problem is that SQL can only act on a database, not a comma-separated values (`.csv`) file, which is how the Adult data set is distributed. To circumvent this problem, the data from the Adult data set must be turned into a database that can be queried in Python. This was done in 3 steps:

1. Editing the `.csv` file to make for easier database production.
2. Making a database from the altered `.csv` file.
3. Making it possible to execute SQL queries on the newly constructed database through the Python interface.

3.3.1 Editing the `.csv` file

A database needs to have a header row with column names, otherwise querying it would be impossible. Because the Adult data set's `.csv` file didn't come with the header names included, they were inserted before the first data row by separating the attribute names with commas.

Because SQL doesn't play nice with column names that contain a hyphen, all hyphens were removed when making the header. This means that columns like `education-num` will have to be queried with `educationnum`.

3.3.2 Making a Database

To turn the `.csv` file into a database, its contents must be extracted in a way that delivers an object Python's syntax can deal with. This is done using the Python module `csv`. The header with the column names is then split from the data rows. For making the database in Python, SQLite[4] (and its corresponding Python module `sqlite3`[5]) was used. SQLite is the most widely deployed database in the world, and as very intricate queries aren't necessary for this research, its functionality scope should suffice.

To now make a database, there are two SQL commands of interest:

- `CREATE TABLE name (column_names);`
  This command creates a table named `name`, with a header full of `column_names`.
- `INSERT INTO name VALUES (values);`
  To actually add data into an existing database, this command is used. `name` specifies the table where `values` needs to be inserted into.

Instantiating the database with the first command, and performing the second command for every row in the `.csv` file creates a database instance that can be queried.

[4] `https://www.sqlite.org/about.html`
[5] `https://docs.python.org/3/library/sqlite3.html`

3.3.3 Executing Queries

Because the `sqlite3` module complies with Python's DB-API 2.0, all possible database functions are built in. This means that once the database is created, querying it is as simple as calling the `execute` function with the desired query as a parameter.

SQL 3.3.1 An SQL query that links marital status to average capital gain

```
SELECT maritalstatus,
        AVG(capitalgain) AS CapitalGain FROM adult
GROUP BY maritalstatus
ORDER BY CapitalGain ASC;
```

If for example SQL query 3.3.1 is executed on the Adult data set, Result 3.3.1 is returned.

Result 3.3.1 Result for SQL query 3.3.1

maritalstatus	CapitalGain
Never-married	376.58831788823363
Married-AF-spouse	432.6521739130435
Separated	535.5687804878049
Widowed	571.0715005035247
Married-spouse-absent	653.9832535885167
Divorced	728.4148098131893
Married-civ-spouse	1764.8595085470085

3.4 NGM Implementation

To test the performance of the noise generation mechanisms, they need to be implemented. This is again done in Python, using a combination of literature and NumPy[6], an importable Python package that contains tools for scientific computing.

3.4.1 The Laplace Mechanism

Each noise generation mechanism is defined as a class instance that has attributes relating to differential privacy parameters. For the Laplace mecha-

[6] `http://www.numpy.org`

nism, there are four of those attributes:

- `epsilon`, the differential privacy parameter.
- `delta`, the global sensitivity of the query function.
- `loc`, the location parameter (μ) of the Laplace distribution.
- `scale`, the scale parameter (b) of the Laplace distribution, calculated by dividing `delta` by `epsilon`.

Adding noise is done through a separate method, called `add_noise` (see Code 3.4.1). It works by simply adding a sampled noise value to the true answer. To keep track of the performance of the noise addition, the `perf_counter` method of Python's `time` library[7] is used. After all noise calculations have taken place, the time difference is multiplied by 1,000,000 to return the execution time in microseconds. In the case of the Laplace mech-

Code 3.4.1 The noise addition method for the Laplace mechanism

```
def add_noise(self, value):
    start = perf_counter()
    noisy = value + self.calculate_noise()
    end = perf_counter()
    return noisy, (end - start) * 1000000
```

anism, the method that calculates the noise is simply a call to NumPy's built-in `laplace` method. This returns a floating point number that is sampled from a Laplace distribution, parameterized by the `loc` and `scale` parameters.

3.4.2 The Staircase Mechanism

Adding noise with the staircase mechanism isn't as trivial as calling a built-in function from a library. Thankfully, Geng et al. give a simple algorithm to generate a random variable according to the staircase distribution specified by ϵ, Δ, and γ [15] (see Algorithm 3.4.1). A noise value is calculated through Equation 3.1, which consists of three major parts: a sign indication, a left hand side, and a right hand side.

$$X = S((1-B)((G+\gamma U)\Delta) + B((G+\gamma+(1-\gamma)U)\Delta)) \tag{3.1}$$

- S determines whether the noise is negative or positive (the sign), with equal probability.

[7] `https://docs.python.org/3/library/time.html`

– The left hand side of the addition that is multiplied by S is a multiplication of $(1 - B)$ by $(G + \gamma U)\Delta$. That means the left hand side will equate to zero when $B = 1$. $(G + \gamma U)\Delta$ shows that G determines the interval of the noise; its lowest is $\gamma = U = 0$, its highest is $\gamma = 1, U \to 1$, or the interval $[G\Delta, (G + 1)\Delta)$.

– If $B = 0$ however, the right hand side will equate to zero. That means B controls whether the noise is in the interval $[G\Delta, (G + \gamma)\Delta)$ for $B = 0$, or $[(G + \gamma)\Delta, (G + 1)\Delta)$ for $B = 1$.

Say noise must be generated for a query function with $\Delta = 1$, and with $\epsilon = 0.1$. Setting γ to its optimal setting of $\frac{\sqrt{\alpha}}{1+\sqrt{\alpha}}$ ([15]) gives $\gamma \approx 0.49$. This means that the noise will be on $[G, G + 1)$ in general, and on $[G, G + 0.49)$ if $B = 0$, or $[G + 0.49, G + 1)$ if $B = 1$ for a geometric random variable G.

Algorithm 3.4.1 Sampling a noise value from the staircase distribution

Input: $\epsilon, \Delta, \gamma \in [0, 1]$.
Output: X, a random variable from the staircase distribution.

Generate S, with $\Pr[S = 1] = \Pr[S = -1] = \frac{1}{2}$.
Generate G, with $\Pr[G = i] = (1 - b)b^i$ for $i \in \mathbb{N}$, where $b = e^{-\epsilon}$.
Generate U, uniformly distributed in $[0, 1]$.
Generate B, with $\Pr[B = 0] = \frac{\gamma}{\gamma+(1-\gamma)b}$ and $\Pr[B = 1] = \frac{(1-\gamma)b}{\gamma+(1-\gamma)b}$.
$X \leftarrow S((1 - B)((G + \gamma U)\Delta) + B((G + \gamma + (1 - \gamma)U)\Delta))$.

The only thing to take note of is the geometric variable G. NumPy's function for sampling a geometric variable uses the formula $f(k) = (1 - p)^{k-1}p$, which is supported on the positive integers, $k \in \mathbb{Z}^+$[8]. This type of geometric distribution shows the number of trials X needed to get one success, where each trial has a probability p of success. This differs from the geometric distribution proposed in Algorithm 3.4.1, $f(k) = (1-p)^k p$, which shows the number of failures before the first success, supported on the set of natural numbers, $k \in \mathbb{N}$ [24]. To rhyme NumPy's implementation of the geometric distribution with the type of geometric distribution required by the algorithm, two things must happen:

– If the first distribution determines the number of trials X, the number of failures Y can be expressed as $Y = X - 1$. This means a value of 1 must be subtracted from a random variable generated by NumPy's geometric sampling function.

[8] `https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.geometric.html`

– Secondly, where NumPy uses the formula $f(k) = (1-p)^k p$, the algorithm requires a distribution of $(1-b)b^k$. Passing the probability $p = 1 - b$ will give

$$\begin{aligned} f(1-b) &= (1-(1-b))^k(1-b) \\ &= b^k(1-b) \end{aligned}$$

which is the desired distribution.

Similar to the Laplace mechanism, the class for the staircase mechanism has some attributes relevant to differential privacy:

– `epsilon`, the differential privacy parameter.

– `delta`, the global sensitivity of the query function.

– `gamma`, the staircase parameter.

– `b`, equal to $e^{-\epsilon}$. Used for computing `biszero` and `bisone`.

– `biszero`, `bisone`, used for determining the value of B.

To implement this algorithm in Python, NumPy's `random` library does come in handy. Its method `choice` can be used for generating S and B, `geometric` is needed for G, and `uniform` for U. All independent values are determined beforehand to save computing time. See Code 3.4.2 for the corresponding implementation. Just like the Laplace implementation, the

Code 3.4.2 The noise calculation method for the staircase mechanism

```
def calculate_noise(self):
    S = np.random.choice([-1, 1])
    G = np.random.geometric(p=1-self.b) - 1
    U = np.random.uniform(0.0, 1.0)
    B = np.random.choice([0, 1], p=[self.biszero, self.bisone])
    X = S * (
        (1 - B) * ((G + self.gamma * U) * self.delta)
        +
        B * ((G + self.gamma + (1 - self.gamma) * U) * self.delta)
        )
    return X
```

staircase mechanism has a method to add generated noise to a true value. Its body is exactly the same as that of Code 3.4.1.

3.5 Testing & Analysis

Comparing the performance of the two NGMs can only be done by executing queries on the data set, adding noise to them, and examining their execution times. There is still much discussion on choosing the appropriate value for ϵ [20, 17], but most papers on this topic seem to conclude that the value for ϵ needs to be determined on a case-by-case basis.

Because the examples here aren't excessively large, experimenting was the easiest way to determine a good value for ϵ. Using $\epsilon = 0.1$ seemed to deliver a good balance of privatized and usable results, as the noisy values didn't deviate so much from the true answers to make them unusable (i.e. positive/negative values kept their sign, noise additions didn't exceed a value of roughly 20 points), but it was not reliably possible to determine the true value from the noisy value.

The query used here is a counting query, meaning that the global sensitivity of the analysis is $\Delta = 1$.

3.5.1 Query & Results

The query that is going to get tested is depicted in SQL 3.5.1. It is a simple counting query that aggregates the total number of respondents per marital status, grouped by race. Due to their relatively low information density, 'Asian Pacific Islander', 'American Indian Eskimo', and 'Other' entries were grouped together.

SQL 3.5.1 Query 1: Counting marital status per race

```
SELECT maritalstatus,
        COUNT(case when race = 'White' then 'White' end) AS 'White',
        COUNT(case when race = 'Black' then 'Black' end) AS 'Black',
        COUNT(case when race = 'Asian-Pac-Islander'
                    OR race  = 'Amer-Indian-Eskimo'
                    OR race  = 'Other' then 'Other' end) AS 'Other'
FROM adult
GROUP BY maritalstatus;
```

Executing SQL 3.5.1 yields the table in Result 3.5.1. Because of its lack of zero values, and relatively high population, the values in the `White` column will be subject to noise addition.

Now adding noise to the `White` column via the previously mentioned `add_noise` method gives Result 3.5.2 (all floating point numbers have been truncated to three decimals). Each individual table contains the noisy answer next to the true answer, and the execution time in microseconds in the

Result 3.5.1 Result after executing SQL 3.5.1

`maritalstatus`	White	Black	Other
Divorced	3797	485	161
Married-AF-spouse	22	1	0
Married-civ-spouse	13410	837	729
Married-spouse-absent	291	62	65
Never-married	8757	1346	580
Separated	717	265	43
Widowed	822	128	43

third column. The mechanism used is printed above.

It has already been proved that using these mechanisms with the correct parameters will result in ϵ-differential privacy [10, 14], and the results in this very small sample seem to reflect that. $\epsilon = 0.1$ is a relatively low value for ϵ—and thus represents a relatively high level of privacy—but general trends in the data are still clear. Noisy answers differ from the true values by only a few points, but remain in the vicinity.

Result 3.5.2 Result after adding noise to Result 3.5.1

Laplace Mechanism

Noisy value	True value	Execution (μs)
3802.529	3797	9.579
14.641	22	3.499
13408.543	13410	2.729
294.871	291	2.457
8771.380	8757	2.385
716.137	717	2.435
817.570	822	2.499

Staircase Mechanism

Noisy value	True value	Execution (μs)
3839.793	3797	287.164
26.058	22	64.918
13431.967	13410	49.474
290.698	291	46.033
8766.160	8757	43.981
710.290	717	44.160
822.900	822	44.360

As evident from Result 3.5.2, there can be quite large swings in execution time when adding noise a single time: the slowest staircase noise addition took 287 microseconds, and the fastest 43.9, which is almost seven times slower. To more accurately compare performance, it's best to add noise multiple times and compare the average execution time.

Taking the average of the noisy values isn't of interest in this scenario, and can even corrode privacy [25], which is why those values will be left

out and replaced by ranking numbers. Performing 100 noise additions and taking the average execution time yields Result 3.5.3.

Result 3.5.3 Average execution times of 100 noise additions with the Staircase and Laplace mechanism

Value	Laplace (μs)	Staircase (μs)
1	2.213	45.654
2	2.092	42.777
3	2.092	42.535
4	2.132	44.384
5	2.087	43.031
6	2.070	42.935
7	2.093	42.831

3.5.2 Conclusions

Comparing their performance, it's quite clear to see that the staircase mechanism performs markedly worse than the Laplace mechanism in terms of execution time. Running the tests multiple times had no noticeable influence on the execution times of both mechanisms. In this scenario, the staircase mechanism is slower than the Laplace mechanism by a factor of approximately 25.

This is directly in contrast with the literature, which states that the staircase mechanism should be optimal [14, 15], especially when $\epsilon \to +\infty$. Adjusting the value for ϵ to math this didn't improve the execution times listed above.

The most likely reason behind this is the fact that NumPy's `laplace` function is written in C, which is a much more efficient language than Python in terms of memory use [13, 23]. To more accurately compare the two mechanisms, the `calculate_noise` method from the staircase mechanism should therefore also be written in C.

Chapter 4

Optimizing Staircase Sampling in C

To better compare the sampling performance of the two noise generation mechanisms, it's best to write them in the same language. Since C and C++ are more efficient than Python [23, 13], they form a more apt choice. The NumPy library's `laplace` method is already written in C, so only the method that samples a noise value from the staircase mechanism (see Code 3.4.2) needs to be written in C.

4.1 C(++) in Python

Luckily, Python supports extending its functionality with C or C++ code[1], eliminating the need to rewrite the entire program. This is done through the creation of *extension modules* that can be imported into a regular Python program. These extension modules can, as opposed to regular Python code, call C library functions. This will aid in optimizing memory use, and therefore also aid in speeding up the program.

The `<Python.h>` header enables the parsing of Python parameters in a C environment, making them usable for function calls. When a C function has returned a value, Python's header also includes methods to parse that result to a Python object, which can be returned to the Python environment.

4.2 C vs. C++

Sampling from a probability distribution relies on randomization. As discussed before, numbers that determine the noise value must be drawn from an evenly distributed interval. C has a built-in random number generator

[1] `https://docs.python.org/3/extending/extending.html`

that could be modified to draw samples from a uniform interval, but its randomness and reliability are questionable at best, as illustrated by Stephan Lavavej in 2013 [19].

Luckily, Lavavej has a solution for C's lack of randomness: using C++'s `<random>` header. This part of C++'s standard library contains many different objects and functions associated with pseudo-random number generation that *can* achieve the desired randomness.

To see what library elements are useful, let's restate the algorithm used for sampling from the staircase mechanism. The values for ϵ, Δ and γ are

---

Input: $\epsilon, \Delta, \gamma \in [0, 1]$.
Output: X, a random variable from the staircase distribution.

Generate S, with $\Pr[S = 1] = \Pr[S = -1] = \frac{1}{2}$.
Generate G, with $\Pr[G = i] = (1 - b)b^i$ for $i \in \mathbb{N}$, where $b = e^{-\epsilon}$.
Generate U, uniformly distributed in $[0, 1]$.
Generate B, with $\Pr[B = 0] = \frac{\gamma}{\gamma+(1-\gamma)b}$ and $\Pr[B = 1] = \frac{(1-\gamma)b}{\gamma+(1-\gamma)b}$.
$X \leftarrow S((1 - B)((G + \gamma U)\Delta) + B((G + \gamma + (1 - \gamma)U)\Delta))$.

---

fixed, so they aren't involved in the randomized steps. For the generation of S, G, U and B on the other hand, randomization is important. They can be grouped into three different categories:

1. Sampling from a uniform distribution, which is the case for U.
2. Sampling from a geometric distribution, which is the case for G.
3. Choosing between two options with their relative probabilities, which is the case for S and B.

Before any randomization can occur, C++ requires the presence of a source of randomness, a so called *engine*. For most applications, including this one, [19] recommends a *Mersenne twister*. This is a pseudorandom number generator proposed in 1997 [21] whose algorithm is based on the Mersenne prime $2^{19937} - 1$, giving it a very long period, and thus a very small chance of repeating.

For sampling from a uniform distribution, C++11 and onwards offer the `uniform_real_distribution` that returns random numbers from a specified interval. It can be initialized with a Mersenne twister engine, after which it will yield pseudorandom numbers.

For the geometric distribution, C++ has implemented the formula [18]

$$P(i|p) = p \cdot (1 - p)^i$$

This formula does correspond with Algorithm 3.4.1 in that it distributes the number of failures before a success. Passing it the probability $p = 1 - b$ will result in the desired distribution $P(i|1-b) = (1-b) \cdot b^i$.

For choosing between two possibilities, the `uniform_int_distribution` can be used for choosing between two options with the same probability. Determining S can be done by generating either a 0 or a 1 with equal probability, and using that as an index for an array storing the possible values of S. This can't be done for B, as the two possibilities don't have an equal probability. To get around this, a random sample on the interval $[0, 1]$ can be drawn, and if that sample's value is smaller than $\frac{\gamma}{\gamma+(1-\gamma)b}$, B will be set to 0, and to 1 otherwise.

4.3 The Code

Code 4.3.1 The noise calculation method for the staircase mechanism in C++

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<double> real_distr(0.0,1.0);
std::uniform_int_distribution<int> int_distr(0,1);
double s_[2] = {-1.0, 1.0};

double calculate_noise(double delta, double gamma,
                       double b, double biszero) {
    std::geometric_distribution<long> geo(1 - b);
    double U = real_distr(gen);
    double G = geo(gen);
    double S = s_[int_distr(gen)];
    double B;
    double b_help = real_distr(gen);
    if (b_help < biszero)
        B = 0.0;
    else
        B = 1.0;

    return S * (
        (1 - B) * ((G + gamma * U) * delta)
        +
        B * ((G + gamma + (1 - gamma) * U) * delta)
    );
}
```

Combining all this information together, the resulting C++ code is shown in Code 4.3.1. To make this function callable from the Python environment, some wrapping C++and Python code had to be written, after which a compiling step builds a Python module. This new module, titled `_staircase`, provides the new function `staircase` that calculates a noise value based on the staircase mechanism in C++, and can be used in a Python environment, similar to NumPy's `laplace` function.

4.4 Performance & Conclusions

To accurately compare performance between Python and C++ sampling, the same query (and results) were used. Performance is still measured using Python's `perf_counter` method to determine execution time in microseconds. The results of adding noise to the true values 100 times is shown in Result 4.4.1.

Result 4.4.1 Execution times of adding noise to Result 3.5.1 with Laplace, Staircase Python, and Staircase C++ sampling

Value	Laplace (μs)	Staircase (Python) (μs)	Staircase (C++) (μs)
1	2.213	45.654	0.892
2	2.092	42.777	0.888
3	2.092	42.535	0.884
4	2.132	44.384	0.839
5	2.087	43.031	0.861
6	2.070	42.935	0.878
7	2.093	42.831	0.859

The performance increase between the Python and C++ sampling is quite dramatic. Where the former took about 42 microseconds in its fastest average execution time, the latter takes just under 0.9 microseconds; that's roughly 46 times faster.

Comparing the new type of sampling with the original Laplace distribution, the performance increase is also evident here, because the staircase mechanism has surpassed the Laplace mechanism in performance. This result reflects the conclusions from [14, 15] that the staircase mechanism is the optimal mechanism between the two.

Chapter 5

Conclusions

Differential privacy is a relatively new privacy framework that aims to protect individuals' privacy while maintaining output usability for querying a dataset. It aims to achieve this by adding proportional noise to the true answer returned by the query function. The proportionality of this noise depends on the desired privacy level, the sensitivity of the query function, and the source where the noise is sampled from.

The most used source is the Laplace distribution, which is a probability distribution whose scale parameter is expressed in terms of the differential privacy parameters. From this parameterized distribution, noise values are sampled, with which the query output is perturbed.

The goal of this thesis was to compare the sampling performance of the Laplace distribution with that of the more recently introduced staircase distribution. Its creators claim that the staircase distribution can be used whenever the Laplace distribution can be used, and that it performs better.

To test this, a dataset was queried in a Python environment, and noise was added with both the Laplace and the staircase distribution. After finding the staircase distribution to be dramatically slower, the noise calculation function was written in C++ to level the playing field, as the Laplace sampling function was written in C.

This led to an improvement in performance. So much so, that the staircase mechanism was faster than the Laplace mechanism by a factor of roughly 2. This corresponded to the claims of the makers of the staircase distribution, solidifying their findings.

Bibliography

[1] Chien-Lun Chen, Ranjan Pal, and Leana Golubchik. Oblivious mechanisms in differential privacy: experiments, conjectures, and open questions. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 41–48. IEEE, 2016.

[2] Tore Dalenius. Towards a methodology for statistical disclosure control. *statistik Tidskrift*, 15(429-444):2–1, 1977.

[3] Luc Devroye. *Non-Uniform Random Variate Generation*, chapter 2, pages 27–29. Springer-Verlag, 1986.

[4] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '03, pages 202–210, New York, NY, USA, 2003. ACM.

[5] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Cynthia Dwork. Ask a better question, get a better answer a new approach to private data analysis. In *11th International Conference on Database Theory (ICDT 2007)*, volume 4353, pages 18–27, Barcelona, Spain, January 2007. Springer.

[7] Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[8] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 486–503. Springer, 2006.

[9] Cynthia Dwork, F McSherry, K Nissim, and A Smith. Calibrating noise to sensitivity in private data analysis. *Theory of cryptography*, pages 265–284, 2006.

[10] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[11] Cynthia Dwork and Moni Naor. On the difficulties of disclosure prevention in statistical databases or the case for differential privacy. *Journal of Privacy and Confidentiality*, 2(1), 2010.

[12] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[13] Mathieu Fourment and Michael R Gillings. A comparison of common programming languages used in bioinformatics. *BMC bioinformatics*, 9(1):82, 2008.

[14] Quan Geng, Peter Kairouz, Sewoong Oh, and Pramod Viswanath. The staircase mechanism in differential privacy. *IEEE Journal of Selected Topics in Signal Processing*, 9(7):1176–1184, 2015.

[15] Quan Geng and Pramod Viswanath. The optimal mechanism in differential privacy. In *Information Theory (ISIT), 2014 IEEE International Symposium on*, pages 2371–2375. IEEE, 2014.

[16] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377. ACM, 1982.

[17] Justin Hsu, Marco Gaboardi, Andreas Haeberlen, Sanjeev Khanna, Arjun Narayan, Benjamin C Pierce, and Aaron Roth. Differential privacy: An economic method for choosing epsilon. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*, pages 398–410. IEEE, 2014.

[18] Nicolai M Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley, 2012.

[19] Stephan T. Lavavej. rand() considered harmful. In *GoingNative C++ Conference*, 2013.

[20] Jaewoo Lee and Chris Clifton. How much is enough? choosing ε for differential privacy. In *International Conference on Information Security*, pages 325–340. Springer, 2011.

[21] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[22] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 75–84. ACM, 2007.

[23] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx and tcl. *IEEE Computer*, 33(10):23–29, 2000.

[24] Prasanna Sahoo. Probability and mathematical statistics. *Louisville, KY40292, USA*, pages 585–60, 2013.

[25] Bibil Seleshi and Samrawit Asseffa. A case study on differential privacy. Master's thesis, Umeå University, 2015.

[26] Christine Task. An illustrated primer in differential privacy. *XRDS: Crossroads, The ACM Magazine for Students*, 20(1):53–57, 2013.