Leopold-Franzens Universität Innsbruck
Fakultät für Mathematik, Informatik und Physik

Institut für Mathematik

Bachelorarbeit

zur Erreichung des akademischen Grades

Bachelor of Science

Post-Quantum Cryptography: Learning with Errors

von

Maximilian Harren
(Matr.-Nr.: 12022305)

Submission Date: February 17, 2025
Supervisor: Tobias Fritz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelorarbeit einverstanden.

Ort und Datum: ______________________________

Unterschrift: ______________________________

Contents

1 Introduction

The development of quantum computers offers a new type of machine that will far exceed the capabilities of our current computers in some areas. The development is still struggling with difficulties in hardware technology, algorithm development, and interdisciplinary research, but this technology offers the possibility of solving problems that currently cannot be solved with conventional computers. However, this incredible performance also brings with it certain new challenges, such as in cryptography.
Our current cryptography is based on the fact that certain mathematical problems are hard to solve on our existing computers. One of these problems is prime factorization, which is at the heart of the RSA-method, for example.

1.1 RSA-method

The RSA-method is an encryption method that is used in many of today's technical devices. It works as follows:

Receiver		Public		Sender
Choose p, q prime numbers				
Calculate: $n = p \cdot q$, $k = (p-1) \cdot (q-1)$				
Find $e \in \mathbb{N}$ such that $ggT(e,k) = 1$				
Calculate $d \in \mathbb{N}$ with $e \cdot d \equiv 1 \mod k$				
	$\rightarrow$	(e,n)	$\rightarrow$	
				Message $a \in \{0, 1, \ldots, n-1\}$
				Calculate: $[b]_n = [a^e]_n$
	$\leftarrow$	$[b]_n$	$\leftarrow$	
Decryption: $a = [b^d]_n$				

Here $gcd(e,k)$ stands for the greatest common divisor of e and k and $[b]_n$ for b modulo n.

As you can see in the illustration, a sent message would no longer be secure if an external party who got hold of n during transmission had the possibility to calculate p and q. By default, n is currently a 2048-bit number. The number you can find in the appendix is an example of a number this size. It is 617 digits long. This number would then have to be broken down into prime factors, a task that even our supercomputers are not up to, as no efficient algorithm for prime factorization is known yet. The situation is different with quantum computers, however, because in 1994 Peter Shor, an American mathematician and computer scientist, published an algorithm that can perform factorization using quantum computers. The only problem here is the existence of quantum computers that are powerful enough for this task. Because of this threat, the US authority National Institute of Standards and Technology (NIST) started a program where possible post-quantum cryptography schemes can be submitted. These are evaluated in rounds and the best ones advance

to the next round. At the time of writing this thesis one method is considered a favorite. It's name is Learning With Errors and it was first introduced in 2005 by Oded Regev. This bachelor thesis will take a closer look at this method.
Chapter 2 starts with a few definitions. Chapter 3 then deals with learning with errors. The standard scheme is presented formally in 3.1, pictorially in 3.2 and with an example in 3.3. The next section 3.4 is dedicated to the correctness of the method and finally, in 3.5, the implementation of the method in Python is presented.

2 Definitions

Definition 2.1 ($\mathbb{Z}_q$)**.** *Let $q \in \mathbb{N}$. Then $\mathbb{Z}/q\mathbb{Z}$ will be denoted as $\mathbb{Z}_q$.*

Definition 2.2 (Little-o)**.** *Let $f(x)$ and $g(x)$ be two nonnegative real-valued functions. Also let $g(x)$ be non zero for all $x \geq x_o$ for an $x_0 > 0$. Then we say $f \in o(g)$ or $f = o(g)$ if $\underset{x\to\infty}{lim} \frac{f(x)}{g(x)} = 0$*

Definition 2.3 (Distance from 0)**.** *Let $e \in \mathbb{Z}_q$ with $q \in \mathbb{N}$. Then the distance of e from zero $|e| = e$, if $e \in \left\{0, \ldots, \left\lfloor \frac{q}{2} \right\rfloor\right\}$ and else $|e| = q - e$.*

Definition 2.4 (Power set)**.** *Let A be an arbitrary set. The power set $\mathcal{P}(A)$ of A contains all possible subsets of A. So $S \in \mathcal{P}(A) \Rightarrow S \subseteq A$.*

Definition 2.5 (Convolution)**.** *Let $k \in \mathbb{N}$ and χ be a probability distribution. The convolution χ^{*k} is defined as the distribution you get by summing up k independent samples from χ.*

Definition 2.6 ($\mathbb{S}$)**.** *The unit circle will be denoted as $\mathbb{S}$.*

Definition 2.7 (Discretization)**.** *[1] For a density function $\phi : \mathbb{S} \to \mathbb{R}^+$ it's discretization $\bar{\phi} : \mathbb{Z}_p \to \mathbb{R}^+$ is obtained by sampling from ϕ, multiplying by q and rounding to the closest integer modulo q. More formally:*

$$\bar{\phi}(i) = \int_{\frac{i-\frac{1}{2}}{q}}^{\frac{i+\frac{1}{2}}{q}} \phi(x)dx$$

Definition 2.8 (wrapped normal distribution). *[1] For $\alpha \in \mathbb{R}^+$ the distribution $\Psi_\alpha$is the distribution on $\mathbb{S}$ obtained by sampling from a normal variable with mean 0 and standard deviation $\frac{\alpha}{\sqrt{2\pi}}$ and reducing the result modulo 1 (i.e., a periodization of the normal distribution). Thus the density is*

$$\forall r \in [0,1), \Psi_\alpha(r) := \sum_{k=-\infty}^{\infty} \frac{1}{\alpha} \cdot \exp\left(-\pi \left(\frac{r-k}{\alpha}\right)^2\right).$$

3 Learning with Errors

3.1 Standard scheme

Let $n, m \in \mathbb{N}$, $q \in \mathbb{N}$ prime number and χ a probability distribution on $\mathbb{Z}_q$.

Receiver:

1. **Private Key:** Choose an arbitrary vector $s \in \mathbb{Z}_q^n$. s will be called the 'private key'.
2. **Public Key:** An arbitrary matrix $A \in \mathbb{Z}_q^{m \times n}$ is selected. Also, choose a random 'error'-vector $e \in \mathbb{Z}_q^m$ with $e_i \sim \chi$ for $i \in \{1, \ldots, m\}$.
   Then the 'public key' is the pair (A, b) whereas $b = A \cdot s + e$.

Sender:

3. **Encryption:** For every bit which is to be sent choose an arbitrary set $R \in \mathcal{P}(\{1, \ldots, m\})$. For $bit = 0$ calculate $\left(\sum\limits_{i \in R} (A_{1i}, \ldots, A_{ni}), \sum\limits_{i \in R} b_i\right)$. In the case of $bit = 1$ calculate $\left(\sum\limits_{i \in R} (A_{1i}, \ldots, A_{ni}), \left\lfloor \frac{q}{2} \right\rfloor + \sum\limits_{i \in R} b_i\right)$. The calculated pair, denoted as $\left(\widetilde{A}, \widetilde{b}\right)$, is then transmitted to the receiver.

Receiver:

4. **Decryption:** The decrypted bit of a pair $\left(\widetilde{A}, \widetilde{b}\right)$ is 0, if $\left|\widetilde{b} - \widetilde{A} \cdot s\right| < \left\lfloor \frac{q}{4} \right\rfloor$ and otherwise $bit = 1$.

3.2 Visualization

Generate:

$$s = \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix} \in \mathbb{Z}_q^n, \quad A = \begin{pmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mn} \end{pmatrix} \in \mathbb{Z}_q^{m \times n}, \quad e = \begin{pmatrix} e_1 \\ \vdots \\ e_m \end{pmatrix} \in \mathbb{Z}_q^m$$

$$b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} = \begin{pmatrix} A_{11} & \dots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{m1} & \dots & A_{mn} \end{pmatrix} \cdot \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix} + \begin{pmatrix} e_1 \\ \vdots \\ e_m \end{pmatrix}$$

Private Key:	**Public Key:**
s	(A, b)

Encryption:

$$R = \{r_1, \dots, r_k\} \subseteq \{1, \dots, m\} \text{ with } k \in \{1, \dots, m\}$$

Calculate: $\widetilde{A} = \sum_{i \in R} (A_{i1}, \dots, A_{in})$ and send $\left(\widetilde{A}, \widetilde{b} = \sum_{i \in R} b_i\right)$ for $bit = 0$ or $\left(\widetilde{A}, \widetilde{b} = \sum_{i \in R} b_i + \left\lfloor \frac{q}{2} \right\rfloor\right)$ for $bit = 1$.

Decryption:

$$\text{Calculate } d = \widetilde{b} - \left(\widetilde{A}_{11}, \dots, \widetilde{A}_{1n}\right) \cdot \begin{pmatrix} s_1 \\ \vdots \\ s_n \end{pmatrix}$$

if $|d| < \left\lfloor \frac{q}{4} \right\rfloor$, the message is interpreted as $bit = 0$, else $bit = 1$.

3.3 Example

An example calculation with $m = n = 4$ and $q = 31$.

$$A = \begin{pmatrix} 12 & 16 & 15 & 24 \\ 8 & 23 & 19 & 4 \\ 6 & 22 & 10 & 24 \\ 3 & 11 & 6 & 20 \end{pmatrix}_{31}, \quad s = \begin{pmatrix} 2 \\ 6 \\ 26 \\ 1 \end{pmatrix}_{31}, \quad e = \begin{pmatrix} 0 \\ 2 \\ 1 \\ 1 \end{pmatrix}_{31},$$

$$c = A \cdot s = \begin{pmatrix} 534 \\ 652 \\ 428 \\ 248 \end{pmatrix}_{31} = \begin{pmatrix} 7 \\ 1 \\ 25 \\ 0 \end{pmatrix}_{31}, \quad b = c + e = \begin{pmatrix} 7 \\ 3 \\ 26 \\ 1 \end{pmatrix}_{31}$$

$$\left(\widetilde{A}, \widetilde{b}\right) = \left(\begin{pmatrix} 12 & 16 & 15 & 24 \\ 8 & 23 & 19 & 4 \\ 6 & 22 & 10 & 24 \\ 3 & 11 & 6 & 20 \end{pmatrix}_{31}, \begin{pmatrix} 7 \\ 3 \\ 26 \\ 1 \end{pmatrix}_{31}\right)$$

The sender wants to send the bits 01. To do this, two subsets of $\{1, 2, 3, 4\}$ have been created, as $m = 4$. He selects the arbitrary sets $R_1 = \{1, 3\}$ and $R_2 = \{1, 2, 4\}$. Now, the bit 0 is first encrypted with R_1. To do this, the rows of A with indices in R_1 are added together. Then do the same with B. So:

$$\begin{array}{cccccc} 12 & 16 & 15 & 24 & & 7 \\ + & + & + & + & & + \\ 6 & 22 & 10 & 24 & & 26 \\ \multicolumn{4}{c}{\begin{pmatrix} 18 & 7 & 25 & 17 \end{pmatrix}_{31}} & & \begin{pmatrix} 2 \end{pmatrix}_{31} \end{array}$$

Since our first *bit* is a 0, $(2)_{31}$ is not further processed. So, the first pair the recipient will obtain is: $((\; 18 \; 7 \; 25 \; 17 \;)_{31}, \; (\; 2 \;)_{31})$.

Next, the bit 1 needs to be sent. For this, we have $R_2 = \{1, 2, 4\}$:

$$\begin{array}{cccccc} 12 & 16 & 15 & 24 & & 7 \\ + & + & + & + & & + \\ 8 & 23 & 19 & 4 & & 3 \\ + & + & + & + & & + \\ 3 & 11 & 6 & 20 & & 1 \end{array}$$
$$\begin{pmatrix} 23 & 19 & 9 & 17 \end{pmatrix}_{31} \quad \begin{pmatrix} 11 \end{pmatrix}_{31}$$

Since $bit = 1$ must be transmitted, we calculate $(11)_{31} + \left\lfloor \frac{31}{2} \right\rfloor = (11)_{31} + 15 = (26)_{31}$. Thus, the receiver now has the pairs $\Big((\ 18 \ \ 7 \ \ 25 \ \ 17 \ )_{31}, \quad (\ 2 \ )_{31}\Big)$ and $\Big((\ 23 \ \ 19 \ \ 9 \ \ 17 \ )_{31}, \quad (\ 26 \ )_{31}\Big)$.

Now, these pairs need to be decrypted. To do this, the recipient calculates:

$$\begin{pmatrix} 18 & 7 & 25 & 17 \end{pmatrix}_{31} \cdot \begin{pmatrix} 2 \\ 6 \\ 26 \\ 1 \end{pmatrix}_{31} = \begin{pmatrix} 745 \end{pmatrix}_{31} = \begin{pmatrix} 1 \end{pmatrix}_{31} < 7.5 \Rightarrow Bit_1 = 0$$

$$\begin{pmatrix} 23 & 19 & 9 & 17 \end{pmatrix}_{31} \cdot \begin{pmatrix} 2 \\ 6 \\ 26 \\ 1 \end{pmatrix}_{31} = \begin{pmatrix} 411 \end{pmatrix}_{31} = \begin{pmatrix} 8 \end{pmatrix}_{31} \geq 7.5 \Rightarrow Bit_2 = 1$$

where $7.5 = \left\lfloor \frac{q}{2} \right\rfloor / 2$. So both bits have been correctly decrypted by the receiver.

3.4 Correctness Dependent on Parameter Selection

The correctness of the decrypted transmission will now be analyzed depending on χ, m and q. This analysis is based on [1].

Theorem 3.1 (Correctness). *Let $\delta > 0$. Assume that for every $k \in \{0, 1, \dots, m\}$ the distribution χ^{*k} satisfies the following:*

$$\mathop{\mathbb{P}}_{e \sim \chi^{*k}} \left[|e| < \left\lfloor \frac{q}{2} \right\rfloor /2 \right] > 1 - \delta \tag{3.1}$$

Then the probability of a decryption error is at most δ. In other words, if we apply the procedure for a bit $c \in \{0, 1\}$, the probability that the bit is correctly decrypted as c is at least $1 - \delta$.

Proof. First, consider the encryption of the bit $c = 0$. The receiver receives the pair $(\widetilde{A}, \widetilde{b})$ with $\widetilde{A} = \sum\limits_{i \in R} (A_{1i}, \dots, A_{ni})$ and $\widetilde{b} = \sum\limits_{i \in R} b_i = \sum\limits_{i \in R} (A_{1i}, \dots, A_{ni}) \cdot s + e_i$. Following the scheme, the receiver now decrypts the message by calculating $\widetilde{b} - \widetilde{A} \cdot s$, which corresponds to $\sum\limits_{i \in R} e_i$. The distribution of this sum is the aforementioned one, namely $\chi^{*|R|}$. Therefore, we can use our assumption in the lemma that $\sum\limits_{i \in R} e_i < \left\lfloor \frac{q}{2} \right\rfloor /2$ with probability $1 - \delta$. In this case, our decrypted message is closer to $[0]_q$ than to $\left\lfloor \frac{q}{2} \right\rfloor$. Thus, it has been correctly decrypted with a probability of $1 - \delta$.
For the bit $c = 1$, $\widetilde{b} - \widetilde{A} \cdot s = \left\lfloor \frac{q}{2} \right\rfloor + \sum\limits_{i \in R} e_i$. Given our assumption that $\sum\limits_{i \in R} e_i < \left\lfloor \frac{q}{2} \right\rfloor /2$ with a probability of $1 - \delta$, and since $\sum\limits_{i \in R} e_i$ is therefore closer to 0, $\left\lfloor \frac{q}{2} \right\rfloor + \sum\limits_{i \in R} e_i$ is closer to $\left\lfloor \frac{q}{2} \right\rfloor$. Hence, the bit $c = 1$ is correctly decrypted with a probability of $1 - \delta$. □

A choice of parameters, which according to O. Regev [1] provides corectness of the procedure with high probability is the following: Choose $q \geq 2$ prime number with $n^2 < q < 2 \cdot n^2$. Also let $m = (1 + \epsilon)(n + 1) \log(q)$ for an arbitrary constant $\epsilon > 0$. The distribution χ shall be $\bar{\Psi}_{\alpha(n)}$ where $\alpha(n) = o(\frac{1}{\sqrt{n} \log(n)})$, so $\lim_{n \to \infty} \alpha(n) \cdot \sqrt{n} \log(n) = 0$. For that a possible option is $\alpha(n) = \frac{1}{\sqrt{n} \log^2(n)}$. This claim has also been proven by Regev [1]. To prove the claim, the following lemma is first shown.

Lemma 3.2. *Let $q \geq 2$ prime number with $n^2 < q < 2 \cdot n^2$ and $m = (1 + \epsilon)(n + 1) \log(q)$ for an arbitrary constant $\epsilon > 0$. Then for large enough q holds*

$$(1 + \epsilon)(\sqrt{q} + 1) \log(q) < \frac{q}{32}$$

Proof. The derivative of $f(q) = (1+\epsilon)(\sqrt{q}+1)\log(q)$ is $f'(q) = \frac{(\epsilon+1)\log(q)}{2\sqrt{q}} + \frac{(\epsilon+1)(\sqrt{q}+1)}{q}$
It applies that $\lim\limits_{q\to\infty} f'(q) = 0$ while $\lim\limits_{q\to\infty} g'(q) = \frac{1}{32}$ with $g(q) = \frac{q}{32}$.
Thus, for q large enough, the given inequality applies. $\square$

Claim 3.3. *With the choice of parameters mentioned above the claim, $k \in \{0, 1, \dots, m\}$ and a negligible function $\delta(n)$ holds:*

$$\underset{e\sim\bar{\Psi}_\alpha^{*k}}{\mathbb{P}}\left[|e| < \left\lfloor\frac{q}{2}\right\rfloor/2\right] > 1 - \delta(n) \tag{3.2}$$

Proof. Let $x_1, \dots, x_k$ be a sample from Ψ_α. A sample of $\bar{\Psi}_\alpha^{*k}$ can be obtained by calculating $\sum_{i=1}^k \lfloor qx_i \rceil \bmod q$. Since rounding can result in a maximum error of 1/2 per sample of Ψ_α, the difference between $\sum_{i=1}^k \lfloor qx_i \rceil \bmod q$ and $\sum_{i=1}^k qx_i \bmod q$ is a maximum of k. With $k \in \{0, \dots, m\}$, $k \leq m$. Here, we will use the Lemma and say $k \leq m < \frac{q}{32}$ for q large enough. It is therefore enough to show that $|\sum_{i=1}^k qx_i \bmod q| < \frac{q}{16}$ with high probability. This is the same as $|\sum_{i=1}^k x_i \mod 1| < \frac{1}{16}$. The distribution of $\sum_{i=1}^k x_i \mod 1$ is $\Psi_{\sqrt{k}\alpha}$ with $\sqrt{k}\alpha = o\left(\frac{1}{\sqrt{\log(n)}}\right)$. Therefore the probability that $|\sum_{i=1}^k x_i \mod 1| < \frac{1}{16}$ is $1 - \delta(n)$ for a negligible function $\delta(n)$. $\square$

Despite this proof, errors are of course not impossible. However, the larger n is chosen, the less likely it is that the decryption will be incorrect.

3.5 Python Implementation

As part of this work, the basic scheme of LWE was implemented in Python. This implementation will now be analyzed here. Initially, all matrix calculations were done with Numpy. However, since there were issues with Numpy in combination with large prime numbers and dimensions concerning the size of the numbers, the code was converted into a version without Numpy matrices. This slightly reduces performance but delivers correct values. This version will be considered here. For those interested, both complete versions of the codes can be found in the appendix.

The packages used are:

```
import random
import math
import numpy as np
```

Next, the values of the parameters q, n, and m are defined in the code.

```
q = 1500019; n =1000; m=1000
```

q is our prime number, n is the dimension of the private key and therefore also the number of columns of our matrix A, and m is the number of our equations, thus the number of rows of matrix A and therefore also the dimension of our error vector E and B from the public key. First, the private key is generated:

```
def generate_private_key(n, q):
    return [[random.randint(0, q-1)] for i in range(n)]
S = generate_private_key(n, q)
```

As input values for our private key, n and q are required. n defines the dimension of S, and q is the maximum number that can appear in the vector. Since we are in the modulus space, 0 and q are equivalent, and the maximum value is therefore $q-1$. To generate a random vector, we use the *random* package, and since we only want whole numbers, we use it in conjunction with randint.
$random.randint(0, q-1)$ generates only one number. To generate multiple at once a $for-loop$ is used. This must be outside the first right square bracket to form a column vector: [[...],...,[...]]. If the $for-loop$ were placed before the first right bracket, we would get a row vector of the form $[[x_1, \dots, x_n]]$. Since the sender and receiver were implemented in the same code for the sake of simplicity, S is stored globally to prevent a new key from being generated for decryption, which would of course result in something incorrect.

Next, we define our Gaussian normal distribution that will generate our error:

```
1 def distribution(n, m, q):
2     std_dev = q/(math.sqrt(n)*math.log(n)**2)
3     error = np.random.normal(loc=0,scale=std_dev,size=m)
4     error = np.round(error)
5     error = error.astype(int)
6     return error.tolist()
```

For this, n, m, and q are required. n and m are used to calculate our standard deviation. The formula from 3.4 is used for the calculation and is multiplied by q. To calculate the square root and the logarithm, the *math* package is used. In line 3, the error vector is created. To do this, the *random* module is not used this time, but *np.random*. This offers the possibility of generating a normally distributed random number. Here, *loc* is the mean of the normal distribution and *std_dev* is the previously calculated standard deviation. In *size*, we specify how many numbers are generated. These are then stored in a numpy row vector. An output for

```
    np.random.normal(loc=0, scale=std_dev, size=1)
```

is for example $array([1658.62307372])$ with $q = 1500019$ and $n = 1000$.
In line 4, the error values are rounded to the nearest whole number using *np.round*. In the above example, this results in $array([1659.])$ Since the value is still a $float$, it is converted to an int in line 5 using $.astype(int)$, resulting in whole numbers without decimal places. To continue without *numpy* arrays, the array is converted to a regular Python list in line 6 using $.tolist()$. This list is then returned by the function.
To visualize the functionality, a histogram of the results is shown below, with the values on the x-axis and their frequency on the y-axis.

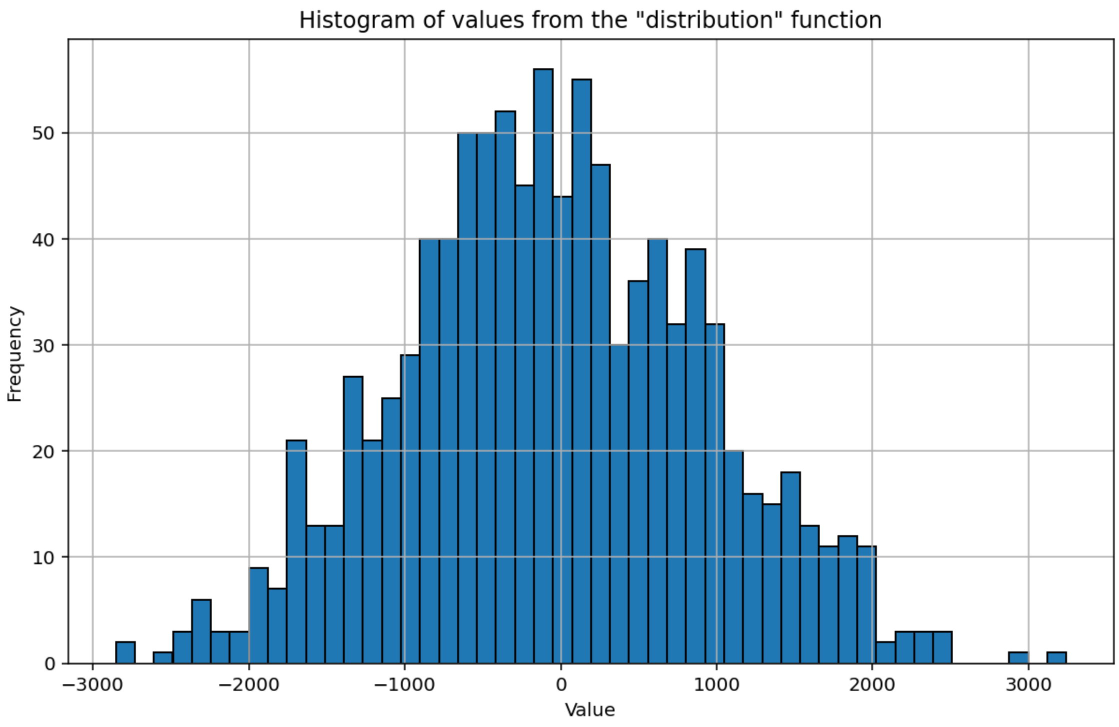

Figure 3.1: Histogram with parameters $q = 1500019$, $n = 1000$, $m = 1000$

Now, the public key can be generated. The function *generate_public_key* is responsible for this.

```python
def generate_public_key(m, n, q):
    A = [[random.randint(0, q-1) for j in range(n)] for i in
  range(m)]
    E = distribution(n, m, q)
    B = [[(sum(A[i][j] * S[j][0] for j in range(n)) + E[i]) %
    q] for i in range(m)]
    return A, B
```

First, in line 2, the arbitrary matrix A is generated using the same method as for the private key, with the only difference that this time 2 for-loops are used. The first loop with $range(n)$ ensures n columns by being inside the innermost brackets. The second loop outside the inner brackets ensures m rows with $range(m)$. For example, $[[1,2],[3,4]]$ represents the matrix $\left(\begin{smallmatrix}1 & 2\\ 3 & 4\end{smallmatrix}\right)$.

In line 3, the error vector E is created using the previously introduced function distribution. This vector is then used in line 4 for calculating B. The matrix multiplication is implemented here using lists and the $sum()$ function. With each i of the for-loop, the i-th row of the column vector B is calculated. A row consists of the multiplication of the i-th row of A with the private key, and the i-th error value from E is added. The final result is then calculated modulo q. This is then done for each line of A or B, i.e. m times. The resulting pair (A, B) is now the public key and is output at the end of the function.

The next function is responsible for requesting a message to be sent and translating it into bits.

```python
def text_to_binary():
    binary_result = ''.join(format(ord(char), '08b') for char
    in input("What is your message?:"))
    return binary_result
```

This function does not require any input values in its execution, as it queries a value from the sender using the $input()$ function. The for-loop then picks up each individual character of the received $string$. Using $ord(char)$, we get the ASCII code of the individual $char$. For example, $ord('a')$ returns 97. The $format()$ function then converts the ASCII code of the $char$ into binary code. Here $'08b'$ does the following: The 0 ensures leading zeros, the 8 sets the total length, and the b stands for binary representation. Therefore, $ord(char)$ is converted into an 8-digit binary number with leading zeros. In the example of $'a'$ with 97 as the ASCII code, the output is $'01100001'$. If you used $'8b'$ instead of $'08b'$, the output would be $'\ 1100001'$, with a space before the first 1 With $.join()$ the $string$ is finally extended with the binary code of each individual $char$.

Next comes the function that is to generate arbitrary subgroups of the set $\{1, \ldots, m\}$. Therefore, m is also required as a parameter.

```
def choose_random_subset(m):
    return random.sample(range(m), random.randint(1, m-1))
```

The function *random.sample*$(sequence,\ k)$ is used to generate a subset. *sequence* is the set for which subsets are generated. *range*(m) is the list with values from 0 to $m-1$. k indicates how many elements the subset should have. This number is generated using the previously shown method *random.randint*(). For example, $[3, 1, 0, 7, 6]$ is a possible output for

```
    random.sample(range(10), random.randint(1, 9))
```

Next, the encryption will be implemented:

```
def encryption(n, q, m):
    binary_output = text_to_binary()
    l = len(binary_output)
    A1 = [[0 for j in range(n)] for i in range(l)]
    B1 = [[0] for i in range(l)]
    A, B = generate_public_key(m, n, q)
    for y, bit in enumerate(binary_output):
        random_subset = choose_random_subset(m)
        a = [(sum(A[j][i] for j in random_subset)) % q for i in range(n)]
        b = (sum(B[j][0] for j in random_subset) + (q // 2 if bit == '1' else 0)) % q
        A1[y] = a
        B1[y][0] = b
    return A1, B1, l
```

The function starts by querying the binary representation of the text to be sent and saves it as *binary_output*. The length of this is measured in the next step, as each bit is encoded individually. The matrices $A1$ and $B1$ are initialized in lines 4 and 5. A $l \times n$ matrix is generated with the two for-loops in line 4 and $B1$ is l dimensional. In line 6, the public key (A, B) is generated with the function *generate_public_key*. The for-loop starting in line 7 has two variables, y and *bit*. y is the index of the selected bit from *binary_output*. In line 8, since it is in the loop, a subset of $\{1, \dots, m\}$ is generated for each bit using the function *choose_random_subset*(m). Now the rows of A, whose index is in *random_subset*, are added up per column. The result, implemented with *sum*() modulo q, is a row vector of dimension $1 \times n$. The same procedure is used with B, except that this time $\lfloor \frac{q}{2} \rfloor$ is added in the case that $bit = 1$. These values are then stored in $A1$ or $B1$ at the index position y. After completing the loop, $A1$ and $B1$ are passed on to the receiver.

The recipient's decryption is implemented with the *decryption* function.

```
1 def decryption(n, q, m):
2     A1, B1, l = encryption(n, q, m)
3     text = []
4     for i in range(l):
5         x = (B1[i][0] - sum(A1[i][j] * S[j][0] for j in range
  (n))) % q
6         text.append(0 if x < q / 4 or x > 3 * q / 4 else 1)
7     text = ''.join([str(bit) for bit in text])
8     return text
```

First, the execution of *encryption* is triggered in line 2. In line 3, *text* is initialized. Then the value x is calculated in the for-loop using the formula $B1_i - A1_{i\dots} \cdot S \bmod q$. x is then checked to see whether $|x| < \frac{q}{4}$ applies. If true, the list text will be appended by a 0, else by a 1. After that, every element of the list will be converted to *str* and appended to a *string*. The output for example for

```
    ''.join([str(i) for i in [0,0,4,2,5]])
```

would be $'00425'$. The string *text* will then be returned by the function.

The final step in the program is the retranslation.

```python
def retranslate():
    bittext_received = decryption(n, q, m)
    text = ''
    for i in range(0, len(bittext_received), 8):
        byte = bittext_received[i:i+8]
        text += chr(int(byte, 2))
    print('Translating bits back, the person now received the message:\n', text)
    return text
```

After getting the string *bittext_received* from the *decryption*-function and initializing the string *text*, the retranslation starts with a for-loop with $range(0,\ len(bittext\_received),\ 8)$ Here, 0 is the start value, $len(bittext\_received)$ the end value and 8 is the step size.

```python
    for i in range(0, 24, 8):
```

would result in $i \in \{0, 8, 16\}$. Now we take the first 8 bit for the first loop and save them as *byte*. $int(byte,\ 2)$ is converting the 8 bit into the decimal system with the base 2.

4 Appendix

4.1 2048-bit number

27524972888058235658985616499933355156094233033455978588568
56008658131672795865396723276016502051556526703195136527247
60572754101203013541034054934073802111415192557231916129336
24090899782375438212753272939537068847502556190320865364850
05116337679807234130010371540620694598796179645330866686688
28141019948322513361696897946095185622410735989094497353416
33256221636411687417928918050784177940343360447477044587607
97233976157887153143324116683319804558634837457745305948834
09157859371862166122088609688673966438425880336140865040592
70196375229266654409238643866971637059325538729962074847222
830944700071974387010973391

4.2 Code without Numpy

```python
import random
import math
import numpy as np

#q = 19207; n = 100; m = 100
q = 1500019; n =1000; m=1000
#q prime number, n dimension private key, m number equations

def generate_private_key(n, q):
    return [[random.randint(0, q-1)] for i in range(n)]
S = generate_private_key(n, q) #Vector 1xn

def distribution(n, m, q):
    std_dev = q/(math.sqrt(n)*math.log(n)**2)
    error = np.random.normal(loc=0, scale=std_dev, size=m)
    error = np.round(error)
    error = error.astype(int)
    return error.tolist()

def generate_public_key(m, n, q):
    A = [[random.randint(0, q-1) for j in range(n)] for i in
  range(m)]
    E = distribution(n, m, q)
    B = [[(sum(A[i][j] * S[j][0] for j in range(n)) + E[i]) %
    q] for i in range(m)]
    return A, B

def text_to_binary():
    binary_result = ''.join(format(ord(char), '08b') for char
    in input("What is your message?:"))
    return binary_result

def choose_random_subset(m):
    return random.sample(range(m), random.randint(1, m-1))

def encryption(n, q, m):
    binary_output = text_to_binary()
    l = len(binary_output)
    A1 = [[0 for j in range(n)] for i in range(l)]
    B1 = [[0] for i in range(l)]
    A, B = generate_public_key(m, n, q)
    for y, bit in enumerate(binary_output):
        random_subset = choose_random_subset(m)
        a = [(sum(A[j][i] for j in random_subset)) % q for i in range(n)]
        b = (sum(B[j][0] for j in random_subset) + (q // 2 if bit == '1' else 0)) % q
        A1[y] = a
        B1[y][0] = b
    return A1, B1, l

def decryption(n, q, m):
    A1, B1, l = encryption(n, q, m)
    for i in range(l):
        x = (B1[i][0] - sum(A1[i][j] * S[j][0] for j in range(n))) % q
        text = ''.join('0' if x < q / 4 or x > 3 * q / 4 else '1')
    return text

def retranslate():
```

```python
    bittext_received = decryption(n, q, m)
    text = ''
    for i in range(0, len(bittext_received), 8):
        byte = bittext_received[i:i+8]
        text += chr(int(byte, 2))
    print('Using a bit translater, the person now received
   the message:\n', text)
    return text

retranslate()
```

4.3 Code with Numpy

```python

#Bachelorarbeit Code

import numpy as np
import math
import random


q = 8053; n = 100; m = 100


def text_to_binary(text):
    binary_result = ''.join(format(ord(char), '08b') for char
    in text)
    return binary_result

text = input("What is your message?:")
binary_output = text_to_binary(text)
print("The Message translated into bits is:", binary_output)
l = len(binary_output)


def distribution(n, m, q):
    std_dev = q/(math.sqrt(n)*math.log(n)**2)

    # Generate a vector with Gaussian distributed values
    values = np.random.normal(loc=0, scale=std_dev, size=m) #
  0 ist Mittelwert

    # Clip the values to be within the range [-q, q]
    values = np.clip(values, -q, q) # Runde die Werte und wende das Modulo an
    values = np.round(values)
    values = values.astype(int) # Konvertiere in ganze Zahlen
    return values


def generate_private_key(n,q):
    return np.random.randint(0,q,(n,1))

S = generate_private_key(n, q)


def generate_public_key(m, n, q):
    A = np.random.randint(0, q, (m, n))
    E = distribution(n, m, q)
    B = np.mod((np.dot(A, S) + E.reshape(-1, 1)),q)

    return A, B


def choose_random_subset_efficient(m):
    return random.sample(range(m), random.randint(1, m-1))

def encryption(n, q, m):
    A1 = np.zeros((1, n))
    B1 = np.zeros((1, 1))
    A, B = generate_public_key(m, n, q)
    binary_output = text_to_binary(text)
    for y, bit in enumerate(binary_output):
        random_subset = choose_random_subset_efficient(m)
        a = np.mod(np.sum(A[random_subset, :], axis=0),q)
        b = np.mod((np.sum(B[random_subset, 0]) + (math.floor
   (q / 2)if bit == '1' else 0)),q)
        A1[y, :] = a
        B1[y, 0] = b
    return A1, B1


def decryption(n, q, l, m):
    A1, B1 = encryption(n, q, m)
    text = []

    for i in range(l):
        x = np.mod((B1[i, 0] - np.dot(A1[i, :], S).item()),q)
        text.append(0 if x < q / 4 or x > 3 * q / 4 else 1)
    return text

bittext_received = decryption(n, q, l, m)

def retranslate(bittext_received):
    bittext_received = ''.join(map(str, bittext_received))
    text = ''
    for i in range(0, len(bittext_received), 8):
        byte = bittext_received[i:i+8]
        text += chr(int(byte, 2))
    print('Using a bit translater, the person now received
   the message:\n', text)
    return text

retranslate(bittext_received)
```

Bibliography

[1] Oded Regev. *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography*. May 2, 2009. URL: `https://cims.nyu.edu/~regev/papers/qcrypto.pdf`.