small attack on lwe

110 • 4 Apr 2025

Edit: new version

Here is a humble (and shy) attack on lwe with a binary secret and a gaussian error. It is inspired by the BKW attack, but it needs much less samples. Instead of selecting very rare vectors in a huge collections of vectors we manage to get the most information possible from each vector.

We present two codes, we will start with a simple attack on few bits and then we discuss an improvement to the first code.

To run the first program:

- Install rust
- in the Cargo.toml file, add

```
[dependencies]
rand = "0.9.0"
rand_distr = "0.5.1"
rayon = "1.10.0"
```

- cargo run –release

The instructions for the second program are described later

Let’s start with the basic idea.

Basic principle

LWE

LWE is a a mathematical problem I will not detail too much here, you just need to know that it is at the core of several post quantum crypto system recently proposed for the reason that, at the time of writing those lines, there are no known quantum algorithm that could break them.

To sum it up (way too) quickly the LWE problem is a set of m linear equations of size n where all coefficients $a_i, i \in \mathbb{Z}_q$ are being sampled uniformly, (in theory, the problem is not defined over a ring, but in real life, it is). The collection of coefficients can be seen as a vector of $\mathbb{Z}_q^n$. The secret $\mathbf{s}$ is also a vector of $\mathbb{Z}_q^n$, thus each linear equation can be seen as a scalar product between two vector of $\mathbb{Z}_q^n$ having their result in $\mathbb{Z}_q$. $\mathbf{s}$ will be called the secret key, because it usually constitutes the core of the private key for all crypto system relying on LWE.

$$\begin{aligned} \langle \mathbf{a}, \mathbf{s} \rangle_0 &\equiv r_0 \mod q \\ &\dots \\ \langle \mathbf{a}, \mathbf{s} \rangle_{m-1} &\equiv r_{m-1} \mod q \end{aligned}$$

Solving that system is easy, but the LWE problem modifies it slightly such that it becomes almost impossible to find the solution, unless m becomes very large. Adding a small random error e_j to each r_j, leads to one of the toughest cryptographic problem to solve.

Problems in cryptography are often not very complicated, but they are hard. It is easy to find a way to solve LWE, yet it is very expensive in terms of computation to do so.

In an lwe sample, or its ring version, the last term is calculated as follow:

$$a_n \equiv \sum_{i=0}^{n-1} a_i s_i + e \mod q \equiv r + e \mod q$$

Usually e follows a (discrete) gaussian distribution, but it is not mandatory. In cryptographic systems relying on LWE, the key is often taken in $\mathbb{B}^n = \{0,1\}^n$ and not in $\mathbb{Z}_q^n$. The attack presented heavily relies on that asumption.

BKW

The BKW attack on LWE relies on finding several vectors of LWE where the a_i are all zero but one, and then average them. For example, if we wanted to find the first component, we would need to find p vectors:

$$(a_0, ..., a_{n-1}, a_n)_j = (1, 0, 0, ..., 0, \langle \mathbf{a}, \mathbf{s} \rangle + e_j \mod q = s_1 + e_j)$$

then sum those p last components and divide the sum by p in order to average them.

$$\frac{\sum_{j=1}^{p} a_{nj}}{p} = s_1 + \frac{\sum_{j=1}^{p} e_j}{p} \approx s_1 + \frac{\sqrt{p}\sigma_e}{p} = s_1 + \frac{\sigma_e}{\sqrt{p}}$$

Those samples are very unlikely, the probability of picking up one is $\frac{1}{q^n}$ and the attack requires the average over the error to not wrap around the modulus, so p needs to be small. The process has to be repeated for each component and thus the attack is very slow despite having a very fast treatment once the samples are obtained.

Alternatively finding p samples with the same first n non zero terms $(a_0, a_1, ..., a_{n-1})$ and averaging the p different a_n should lead to a value close to $\langle \mathbf{a}, \mathbf{s} \rangle$. (As long as $\langle \mathbf{a}, \mathbf{s} \rangle$ is far enough from the representative bounds). To solve a system, We're looking for n different linearly independent vectors, and for each of these vectors, we need to find p occurrences of that exact same vector. Finding several collisions among m samples is less unlikely because of the "birthday effect", but we need to find n independent ones. Our method allows for finding "approximate" collisions, i.e. samples whose components are close to each others without being exactly the same. I suppose it might also be doable with the original BKW attack. One should consider the number of different values to be $\left(\frac{q}{\Delta}\right)^n$ instead of q^n, where Δ is a spread around each component. When a collision is detected in our method, it is used to create a linear combination. It is insensitive to the representatives, and additions can provide new valid samples as well as subtraction, multiplying the number of collisions by 4 in a given set. One key difference between our method and the riginal BKW attack is that because all approximate collisions give rise to a new sample with small values, new collisions between aproximate collisions are increasingly more likely.

In the BKW attack, notable improvements can be obtained through linear combinations, yet the attack is still fairly slow and require a huge amount of samples.

Our proposition

We suggest several improvements over that method that allow to drastically reduce the number of initial samples and create a much faster attack overall. The attack relies on the asumption that the secret is distributed in $U(\mathbb{B}^n)$ and not in $U(\mathbb{Z}_q^n)$. The improvements are namely

- Avoiding the modulus wrapping of the error problem.
- Considering the scalar product as an error.
- Using the Hamming weight.
- Using the circular mean.
- Step by step reducing the scalar product error through linear combinations.

Avoiding the modulus wrapping of the error problem

Avoiding the problem of the sum of the errors wrapping around the modulus is actually the easiest part. We just need to sum it in $\mathbb{Z}$ instead of $\mathbb{Z}_q$. In practice we average over 128 bits integers while q is a 64 bits integer.

Considering the scalar product as an error

Now that the wrapping of the *sum* of the error is no longer a problem, we can consider bigger errors. The error itself just needs to not wrap around the modulus. Thus we can consider the rest of the scalar product as an error.

$$a_n = \langle \mathbf{a}, \mathbf{s} \rangle + e = a_0 s_0 + \langle \mathbf{a}', \mathbf{s} \rangle + e$$

where

$$\mathbf{a}' = (0, a_1, ..., a_{n-1})$$

As long as a_n does not wrap around the modulus, and we select only a_n such that a_0 is **positive**

$$\frac{\sum_{j=1}^{p} \langle \mathbf{a}_j, \mathbf{s} \rangle + e_j}{p} \approx \sum_j \frac{a_{0j} s_0}{p} \approx \frac{q s_0}{4} \approx \frac{q}{4} \text{ or } 0 \text{ if } s_i \in \mathbb{B}$$

If a_0 is negative, we can take $a'_n = -a_n$ instead, and add it to the calculation of the mean.

This is the principle behind the data treatment applied to selected samples:

$r \leftarrow (0_0, 0_1, ..., 0_{n-1})$

for $a \in$ selected samples **do**
  for $i \in 0..n$ **do**
    if $a_i \geqslant 0$ **then**
      $r_i \leftarrow r_i + a_n$
    else
      $r_i \leftarrow r_i - a_n$
    end if
  end for
end for
for $r_i \in r$ **do**
  if $r_i \geqslant 0$ **then**
    $s_i \leftarrow 1$
  else
    $s_i \leftarrow 0$
  end if
end for

You could wonder why it should work differently for different bits, since it seems we always add the same term. In reality we create a different mean for each s_i because adding a_n or $-a_n$ is conditionned by the sign of a_i

We create m samples following an LWE (like) distribution, each of which is of size $n+1$, the first n terms are the a_is and the last term is $a_n = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q$.

We can have a great certitude that the scalar product lies in $I = [\![-\frac{q}{2}, \frac{q}{2}]\!]$ by selecting samples where the values of each a_i is so small that it becomes very likely for the sum of the a_i to land in I.

For example, a radical solution to achieve such a feat is to set a limit ℓ on each a_i such that the $\sum_i |a_i| = n\ell \leqslant \frac{q}{2}$. Then the limit has the value:

$$\ell = \frac{q}{2n}$$

Such a drastic constraint leads to very few samples being accepted, the probability of finding a random sample verifying this is

$$p = \frac{1}{n^n}$$

which is worse than exponential. If $n > q$, it's worse than $\frac{1}{q^n}$. Happily, in most cases $n \ll q$. Yet it is worse than $\frac{1}{2^n}$ which would be a brute force attack on the key.

We can already increase the number of acceptable samples by observing that on average, only half of the a_i will participate to the scalar product. We increase the number of accepted samples by taking the hamming weight h instead of n. The hamming weight is the number of non zero bits of $\mathbf{s}$, thus $h = \sum_i s_i$. We usually have $h \approx \frac{n}{2}$. Any large deviation of h from $n/2$ can be used by the attack to recover the secret key more easily so we consider the hamming weight to be approximately $n/2$.

the probability of finding a sample becomes

$$p = \frac{1}{h^h} \approx \sqrt{\frac{2^n}{n^n}}$$

But because we do not know which s_i is non zero, we just increase the detection threshold accordingly.

There are now

$$\frac{\sqrt{\frac{2^n}{n^n}}}{\frac{1}{n^n}} = (2n)^{n/2} = \sqrt{2n}^n$$

more samples now than in the case where all n were considered. So we define a new limit

$$\ell' = \frac{q}{2n} \times \sqrt{2n} = \frac{q}{\sqrt{2n}}$$

which is still terrible but much higher and gives a good estimate of the threshold we should use.

For $n = 20$ and $q = 2^{64}$ the limit for each individual a_i becomes

$$\ell' = 2^{61.33}$$

Two key advantages appear here over the classical BKW attack.

- The probability of finding a correct sample is much bigger than in the BKW attack
- Each good sample can be used for any component

In addition, if we set a strict limit ℓ for each $|a_i|$ we miss all the sample which "almost made it". Some samples have coefficients a_i all below the limit except for few of them, and those coefficient could very well count for nothing because s_i could be equal to 0. Those samples can certainly be of statistical usefulness for computing an average.

The constraint we use instead is that the ***sum*** of the $|a_i|$ must be below a given threshold instead of constraining each component. Given that the negative terms often balance the positive terms and vice versa, having a sample that should be accepted is much more likely than with the previous constraint.

Calculating $n\ell' = T$ for $n = 20$ we find a threshold T:

$$\sum_i |a_i| \leqslant T = 2^{65.65}$$

The calculation of the mean over p samples veryfying the constraint should then return

$$\frac{\sum_{j=1}^{p} \langle \mathbf{a_j}, \mathbf{s} \rangle + e_j}{p} = \frac{q}{2\sqrt{2n}} \text{ if } s_i = 1$$

instead of $\frac{q}{4}$ as mentionned earlier

Distribution

Ideally, we might be able to increase further more the limit by studying more in detail the distribution of $\sum_i a_i$ instead of $\sum_i |a_i|$. As previously stated the positive terms often balance the negative terms. $\sum_i |a_i|$ grows like $n\frac{\ell}{2}$ with a standard deviation growing proportionnal to $\sqrt{n}$ while $\sum_i a_i$ should grow like its standard deviation $\sqrt{n}$ if its mean was to be 0. In our code that last constraint is not exactly verified because the mean is -0.5.

The distribution followed by a random variable $Y = \sum_i X_i$ where all X_i are iid random variables following a uniform distribution over I uses the polynomial coefficients.

If $Y = \sum_{i=0}^{n-1} X_i$ where all the X_i follows a uniform distribution $U(I'), I' = [\![0, q-1]\!]$, then the probability

$$P(Y = y) = \frac{1}{q^n} \binom{n}{y}_q, y \in [\![0, nq]\!]$$

where

$$\binom{n}{y}_q$$

is a polynomial coefficient, i.e. coefficient in front of x^j when in the expansion of

$$(1 + x + ... + x^{q-1})^n$$

Polynomial cofficients can be expressed as the sum of multinomial coefficients or by recurrence using the binomial coefficients.

(CAMILA C. S. CAIADO)

For the interval I instead of I' we can shift all values and define a new variable $Z = \sum_i (X_i - \frac{q}{2}) = Y - \frac{nq}{2}$, then

$$\begin{aligned} P(|Z| \leqslant \ell) &= P(-\ell \leqslant Z \leqslant \ell) \\ &= P(-\ell \leqslant Y - \frac{nq}{2} \leqslant \ell) \\ &= P(\frac{nq}{2} - \ell \leqslant Y \leqslant \frac{nq}{2} + \ell) \\ &= P(Y \leqslant \frac{nq}{2} + \ell) - P(Y \leqslant \frac{nq}{2} - \ell - 1) \\ &= \frac{1}{q^n} \sum_{y=\frac{n}{2}-\ell}^{\frac{n}{2}+\ell} \binom{n}{y}_q \end{aligned}$$

Because the polynomial coefficients are hard to compute, and because of the central limit theorem, when n is big enough, we can fairly approximate those coefficients by a gaussian distribution of expected value $\mu = \mathbb{E}(Z) = -\frac{n}{2}$, $\sigma = \sqrt{\frac{n(q^2-1)}{12}} \approx q\sqrt{\frac{n}{12}}$

$$P(-\frac{q}{2} \leqslant Z \leqslant \frac{q}{2} - 1) \approx \frac{1}{\sigma\sqrt{2\pi}} \int_{-\frac{q}{2}}^{\frac{q}{2}-1} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \sqrt{\frac{6}{nq^2\pi}} \int_{-\frac{q}{2}}^{\frac{q}{2}-1} e^{-6\frac{(x+\frac{n}{2})^2}{nq^2}} dx$$

So what is the approximate probability that a random sample lies in the interval?

For $n = 20$ this is 0.30. Approximately one third of all samples lie in I. The problem is that our criterion on the sum of the a_i can not pick up all samples landing in that interval.

On average $|a_i| = \frac{q}{4}$, a reasonable heuristic is thus to have:

$$\sum_i |a_i| \leqslant P(-\frac{q}{2} \leqslant Z \leqslant \frac{q}{2} - 1) n \frac{q}{4}$$

For $n = 20$ this gives

$$\sum_i |a_i| \leqslant 2^{64.592}$$

But we already said on average, only half of the a_i would be used so:

$$P(-\frac{q}{2} \leqslant Z \leqslant \frac{q}{2} - 1) \approx \sqrt{\frac{6}{hq^2\pi}} \int_{-\frac{q}{2}}^{\frac{q}{2}-1} e^{-6\frac{(x+\frac{h}{2})^2}{hq^2}} dx$$

and

$$\sum_i |a_i| \leqslant P(-\frac{q}{2} \leqslant Z \leqslant \frac{q}{2} - 1) h \frac{q}{4} = T$$

For $n = 20$, this gives an threshold actually lower than by using n, $T = 2^{64.06}$.

In practice though, we recover lot of signal by setting a threshold around $2^{65.5}$.

Back to our first idea to find the probability, we define new random variables $X_i \in I'' = [\![0, \frac{q}{2} - 1]\!]$ for the $|a_i|$ following a uniform distribution $U(I'')$. Then, if $Y = \sum_{i=0}^{n-1} X_i$, then the probability to find a valid sample among random samples is

$$P(Y \leqslant T) = \frac{1}{q^n} \binom{n}{y}_q, y \in [\![0, T]\!]$$

$$P(Y \leqslant T) \approx \frac{1}{\sigma' \sqrt{2\pi}} \int_{-\infty}^{T} e^{-\frac{(y-\mu')^2}{2\sigma'^2}} dy$$

With $\mu' = \frac{nq}{4}$ and $\sigma' = n\sqrt{\frac{\frac{q^2}{4}-1}{12}} \approx \frac{nq}{4\sqrt{3}}$

For $T = 2^{64.592}$ and $n = 20$, this gives a proability

$$P(Y \leqslant T) \approx 0.1132$$

Over 2000000 samples, we should recover approximately 22000. This is absolutely not the number of valid samples that we find, so there might be a huge mistake somewhere.

Hamming weight

We can further increase the number of samples by using some partial information about the key, namely the hamming weight h of the key. The two techniques described here can be used for the other techniques described later.

If we know the hamming weight of the key, we can recover many more samples. In a real life situation, we might not know h, but there are so few possibilities that if the attack is efficient, we might be able to simulate all the possible hamming weights brute forcing them, either on a single machine or on multiple machines. For a key of size 20, there are only 21 different possibilities for the hamming weight, and the key $(0, 0.., 0)$ not only is unlikely but is also not acceptable. The very same is true for $(1, 1.., 1)$. In that case, any sample would be valid if we would remove $\sum_i a_i$ from a_n, furthermore obviously, the keys with small or big hamming weights are easy to brute force.

Not small "small samples"

We previously explained that we were searching for samples where $\sum_i |a_i| \leqslant T = n\ell'$. Consider a sample where all values a_i are "bigger" than ℓ' but there exist a value $c > T$ such that

$$\sum_i |a_i - c| \leqslant T$$

then knowing the hamming weight, we can transform the sample according to

$$(a_0, a_1, ..., a_{n-1}, a_n) \to (a_0 - c, a_1 - c, ..., a_{n-1} - c, a_n - ch) = (a'_0, a'_1, ..., a'_{n-1}, a'_n)$$

and get a new valid sample verifying the constraint.

This is because

$$a_n - ch = \langle \mathbf{a}, \mathbf{s} \rangle + e - ch = \langle \mathbf{a}, \mathbf{s} \rangle + e - c \sum_i s_i = \langle \mathbf{a} - \mathbf{c}, \mathbf{s} \rangle + e$$

How many more samples can it represent? If we take a similar constraint than the one stated initially on each component of a vector, we need all values a_i to verify $|a_i - c| < \ell'$.

Calling a_s the smallest value of all the a_i and a_b the biggest value, if such c exist then there is certainly one between a_s and a_b. Thus we require that $|a_b - a_s| < 2\ell'$. That condition can therefore be valid for any difference between any two components of the vector. Specifically we can apply that constraint to consecutive components.

We can thus require that

$$\forall i \in [\![0, n), \ |a_{i+1} - a_i| < 2\ell' \Leftrightarrow \forall i \in [\![0, n-1) \ |a'_i| < 2\ell'$$

Since there are $n - 1$ components instead of n and the limit is twice as big as it was previously, the number of samples verifying that constraint is

$$\frac{\frac{2^{n-1}}{(n-1)^{n-1}}}{\frac{1}{n^n}} = 2^{n-1} n \frac{n^{n-1}}{(n-1)^{n-1}} = 2^{n-1} n (1 + \frac{1}{n-1})^{n-1} \approx 2^{n-1} ne$$

bigger than with the previous calculation, which means we are $2^{n-1}ne$ more likely to find a sample knowing the hamming weight than when we do not know it. It seems that knowing the hamming weight provides more information than just reducing the key size by a single bit because the limit is twice bigger.

We take that factor for the probability p' of finding a valid sample

$$p' \approx p 2^{n-1} ne = \frac{e 2^{\frac{3n-2}{2}}}{n^{\frac{n}{2}}} = \frac{e}{2} \frac{8^{n/2}}{n^{n/2}}$$

put some data here

There seems to be some sort of non sense here because it basically states that for $n \leqslant 8$ all samples can become valid. Yet, it's easy to verify that it is actually the case, which does not prove the model is valid per se.

Big hamming weight = small hamming weight

To recover a signal the algorithm piles up many a_n, it takes advantage of the fact that $s_i \in \{0, 1\}$. If $s_i \in \{-1, 0\}$ instead, the algorithm works just the same, the values will average to $-\frac{q}{2\sqrt{2n}}$ where $s_i = -1$ and ≈ 0 where $s_i = 0$.

On the other side, it's pretty obvious that it is easier to find a key with a small hamming weight than a key with a big hamming weight with the technique described. With a smaller hamming weight, a_n is less likely to wrap around the modulus, thus the algorithm can accept samples with bigger components, hence more samples.

There is a way to handle big hamming weights just as small hamming weights. Let $\mathbf{s}$ be a secret key of hamming weight $h > \frac{n}{2}$. Then we can define

$$\mathbf{s}' = \mathbf{s} - (1, 1, ..., 1)$$

$\mathbf{s}'$ has a hamming weight $h' < \frac{n}{2}$, with components $\in \{-1, 0\}$

Let

$$(a_0, ..., a_{n-1}, a_n)$$

be a sample under the secret key $\mathbf{s}$ then it's easy to show that

$$(a_0, ..., a_{n-1}, a'_n = a_n - \sum_i a_i) = (a_0, ..., a_{n-1}, \langle \mathbf{a}, \mathbf{s}' \rangle + e)$$

A big hamming weight can thus be treated as a small hamming weight creating large negative values were the $s_i = 0$

circular mean

We already stated that if an LWE sample can be written:

$$(a_0, a_1, ..., a_{n-1}, a_n = \sum_{i=0}^{n-1} a_i s_i + e \mod q)$$

then if c is a constant,

$$(a_0 - c, a_1 - c, ..., a_{n-1} - c, a_n - ch \mod q)$$

is also an LWE sample.

For some constant c and some representatives of the a_i we can lower the disance $\sum_i |a_i - c|$. It's then possible that a sample which previously did not verify the criterion

$$\sum_i |a_i| \leqslant T$$

verifies

$$\sum_i |a_i - c| \leqslant T$$

after finding the correct c and the correct representative of the a_i

Thus, knowing the hamming weight allows to transform the sample.After finding "the best" c and "the best" representatives, applying the transformation to all samples change their distribution.

Let's illustrate the idea by a small example with an error of 0. We choose the modulus to be 16 with representatives initially in $[\![-8, 7]\!]$, $n = 5$, the secret key $\mathbf{s} = (0, 1, 1, 0, 1)$

if

$$(a_0, a_1, a_2, a_3, a_4, a_5) = (4, 7, -3, -5, 7, 11)$$

the calculation of a_5 is detail is given by:

$$a_5 = 4 \times 0 + 7 \times 1 + (-3) \times 1 + (-5) \times 0 + 7 \times 1 = 7 - 3 + 7 = 11$$

then the "mean" of all the a_i is $(4 + 7 - 3 - 5 + 7)/5 = 2$ the sample is not "centered" around the mean. In addition, the distance to 0 is $4 + 7 + 3 + 5 + 7 = 26$, the distance to its mean is $2 + 5 + 5 + 7 + 5 = 24$.

Now if we are in a situation where we know the hamming weight $h = 3$ of the secret key $\mathbf{s}$, then we can search for a smaller distance to the mean because we can transform a_5.

To do so, we search for other representatives, compute the mean in that representative system and see if the distance to the mean is smaller than the distance we started with, here 24.

$(a_0, a_1, a_2, a_3, a_4) = (4, 7, -3, -5, 7) \equiv (4, 7, -3, 11, 7) \equiv (4, 7, 13, 11, 7) \equiv (20, 7, 13, 11, 7) \equiv (20, 23, 13, 11, 7$

With respective means in $\mathbb{R}$ of 2, 5.2, 8.4, 11.6, 14.8 and distances to the mean of

$$2 + 5 + 5 + 7 + 5 = 24$$

$$1.2 + 1.8 + 8.2 + 5.8 + 1.8 = 18.8$$

$$4.4 + 1.4 + 4.6 + 2.6 + 1.4 = 14.4$$

$$8.4 + 4.6 + 1.4 + 0.6 + 4.6 = 19.6$$

$$5.2 + 8.2 + 1.8 + 3.8 + 7.8 = 26.8$$

The smallest distance to the mean is obtained by the third choice of representative whose mean is 8.4. So let's ignore the fact that the mean is not an integer, and let's take 8 as a mean. We now remove 8 from all the a_i of $(4, 7, 13, 11, 7)$

$$(a_0, a_1, a_2, a_3, a_4) \to (-4, -1, 5, 3, -1)$$

Its distance to 0 is 14.

And to get a legitimate LWE sample, we also have to remove $8h = 24$ from a_5.

$$a'_5 \equiv a_5 - 8h \equiv \sum a_i s_i - 8h \equiv \sum (a_i - 8)s_i \equiv 11 - 24 \equiv 3 \mod 16$$

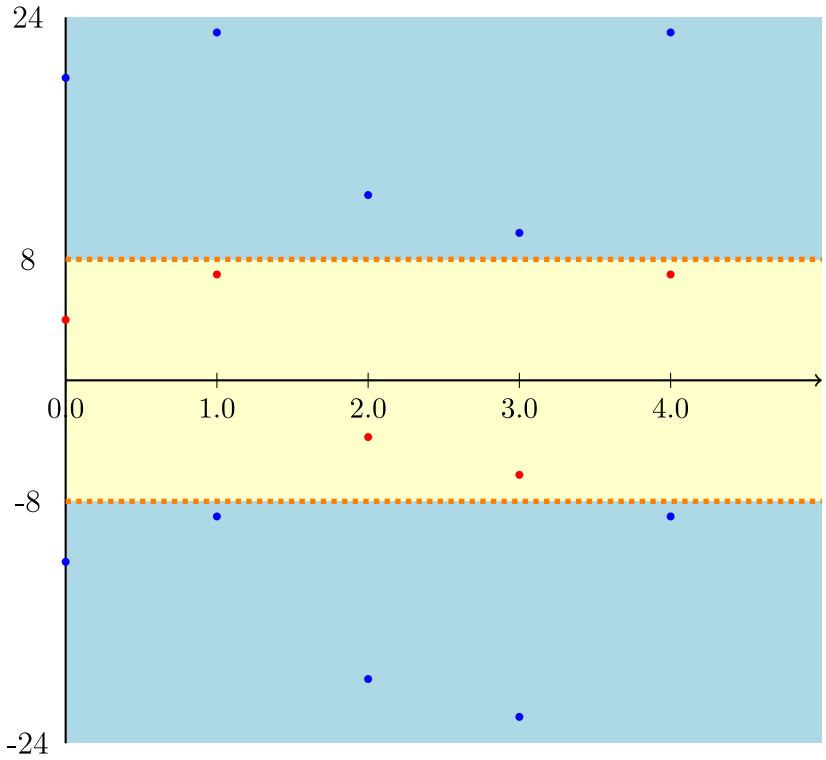
The new sample is thus

$$(-4, -1, 5, 3, -1, 3)$$

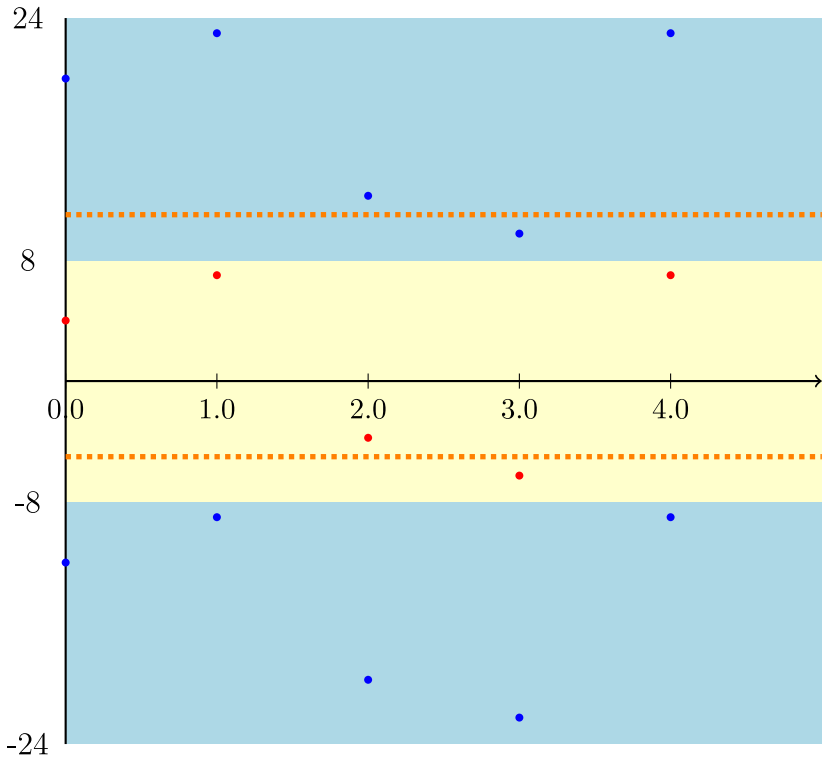
We computed a sort of circular mean on the ring, and we centered the sample around that mean. The transformation on a_5 is performed such that the sample is still a valid LWE sample and we modified a_5 using the hamming weight. One important point to note is that sutracting the mean c from all components expressed in other representative system move all the $a'_i = a_i - c$ into the interval I.

Here is an illustration of the procedure applied on the 5 first components of the vector, keeping the treatment of a_5 for later.

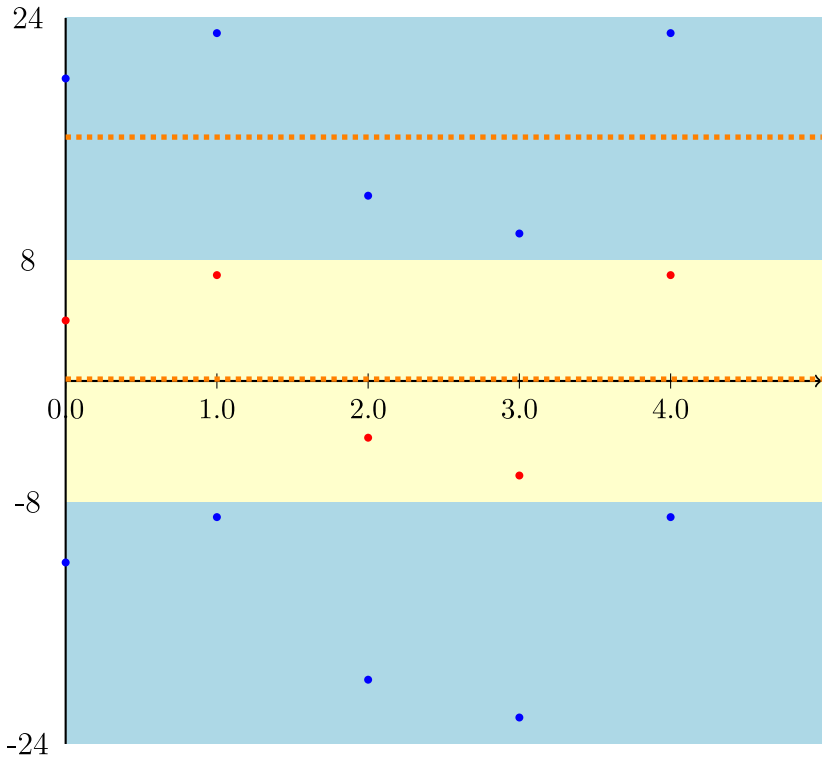
$(4, 7, -3, -5, 7)$:



$(4, 7, -3, 11, 7)$:

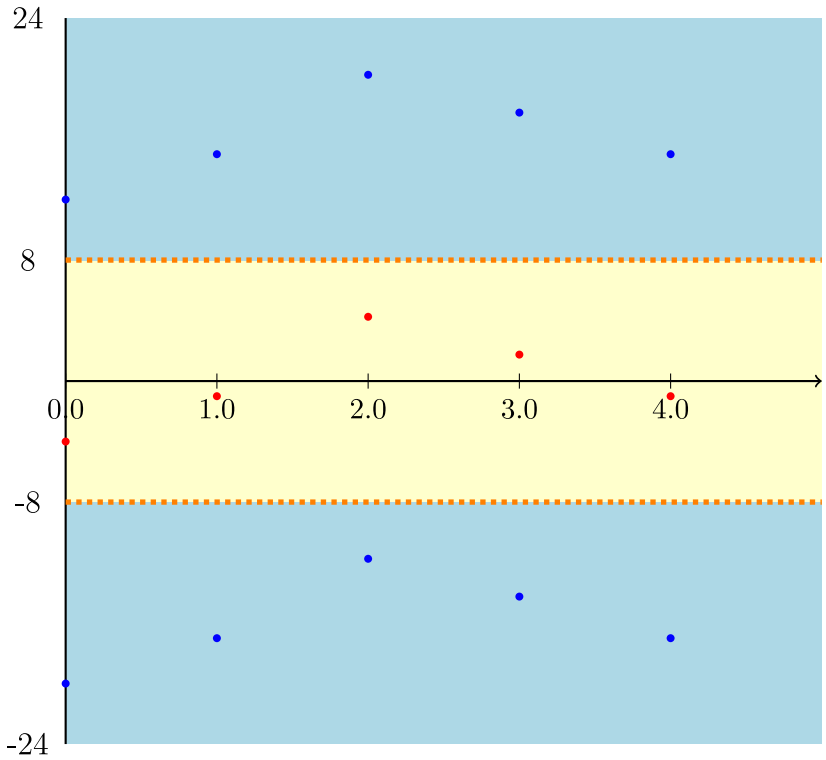


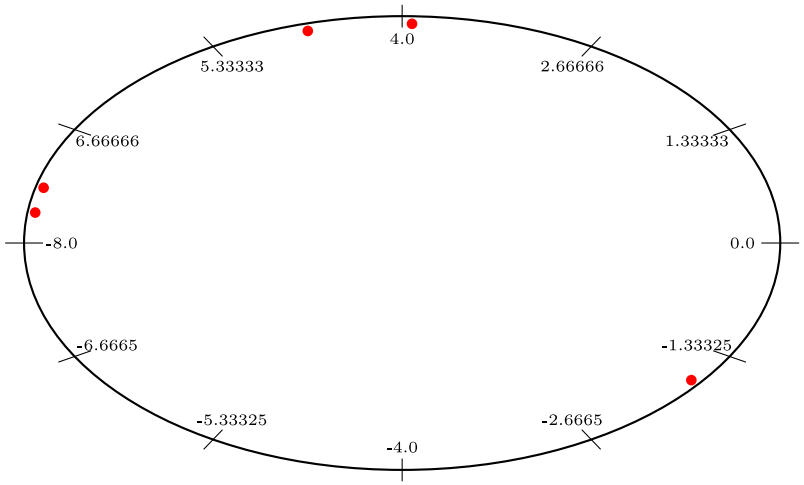
$(4, 7, 13, 11, 7)$:



After subtracting 8 to all the a_i

$(-4, -1, 5, 3, -1)$:





Let's try to evaluate how the distribution is transformed by the procedure described called roll_subarray in the code. We start with a vector $(a_0, ..., a_{n-1})$ where all a_i are distributed uniformly over I, the sample is the "best sample" by default, then the program

- computes the mean $\bar{m}$ in $\mathbb{Z}$,
- computes the distance to the mean $\sum_i |a_i - \bar{m}|$.
- if the new distance is smaller than the best distance, the sample becomes the best sample.
- looks to the "smallest" value a_s and add q to it.
- repeat n times
- keeps the best sample and subtract the best mean $\bar{m}_{best}$ to all a_i. All values should belong to I

On the ring, a_s is either closer to the value just above a_{s+} or closer to the value just under, $a_b - q \equiv a_b \mod q$ with a 50/50 probability. If a_s is closer to $a_b - q$ than to a_{s+} then adding q to a_s decreases the spread between the maximal value and the minimal value. It might not diminish the distance to the mean, but in general, we expect that adding q to the minimal value decreases the distance to the mean approximately half the time.

We model the situation by a 0.5 probability to diminish or increase the distance to the mean at eahc step of the circular mean algorithm. The standard deviation of the *number* of times the distance can diminish is thus

$$\sqrt{\frac{n}{4}}$$

The distance between two values is on average $\frac{q}{n+1}$. A value a_i is expected to be either closer to the a_{i+} above or the a_{i-} below with a chance of 0.5. We can expect a_i to be uniformly distributed on an interval of span $\frac{2q}{n+1}$, so we can expect that if the distance diminishes at one step, it diminishes on average by a distance of $\sqrt{\frac{4q^2}{12(n+1)^2}} = \frac{q}{\sqrt{3}(n+1)}$.

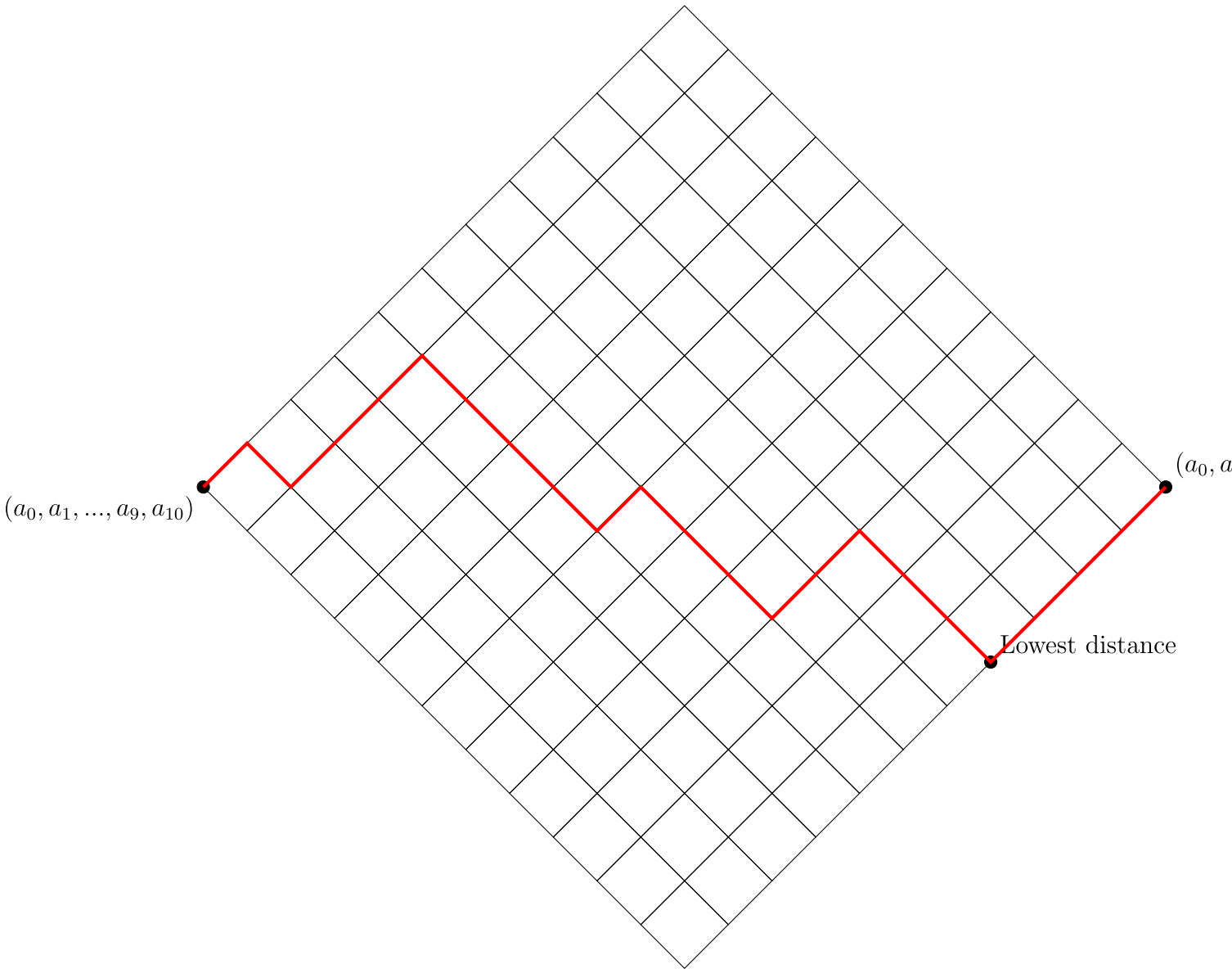
Finally averaging on averages, after the complete procedure, we can expect to have a diminution of

$$\Delta d = \frac{q}{\sqrt{3}(n+1)} \times \sqrt{\frac{n}{4}} = \frac{q}{2(\sqrt{3n} + \sqrt{\frac{3}{n}})} \approx \frac{q}{2\sqrt{3n}}$$

For $n = 20$, this evaluates to $\frac{q}{4\sqrt{15}} \approx \frac{q}{16}$

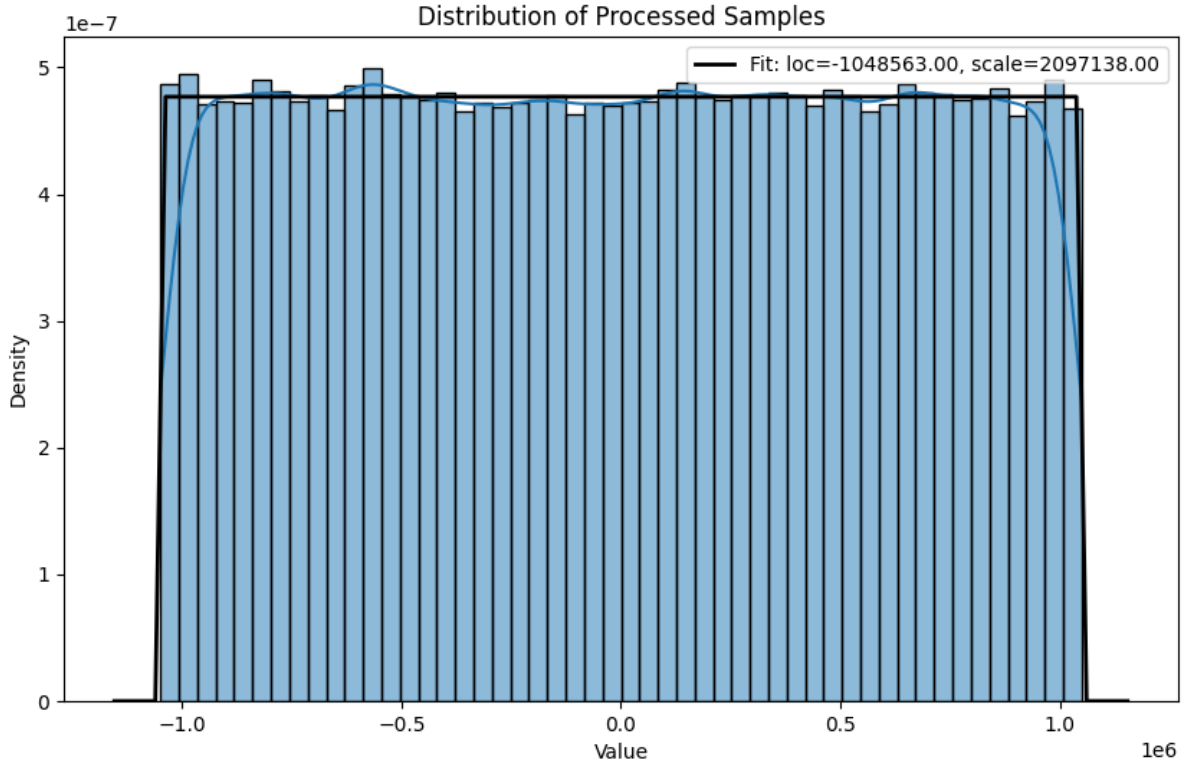
This is a very rough estimate because I do not have the exact distribution now. The procedure evaluates the total distance $\sum_i |a_i|$ so we can also expect a similar influence on all the other values.

A better model but still unperfect, might be describing the situation by a random walk on a square grid of size $d(n/2+1) \times d(n/2+1)$ where $d = \frac{q}{\sqrt{3}(n+1)}$, where each node is a state that can be reached by a walker starting on one corner. At each step $\sum_i |a_i - \sum_i \frac{a_i}{n}|$ either increases or decreases. The walker is able to only go down right (for a lower value of the sum) or up right (for a greater value of the sum). The sum decreases or increases by a fixed amount at each step, and must return to its original value.

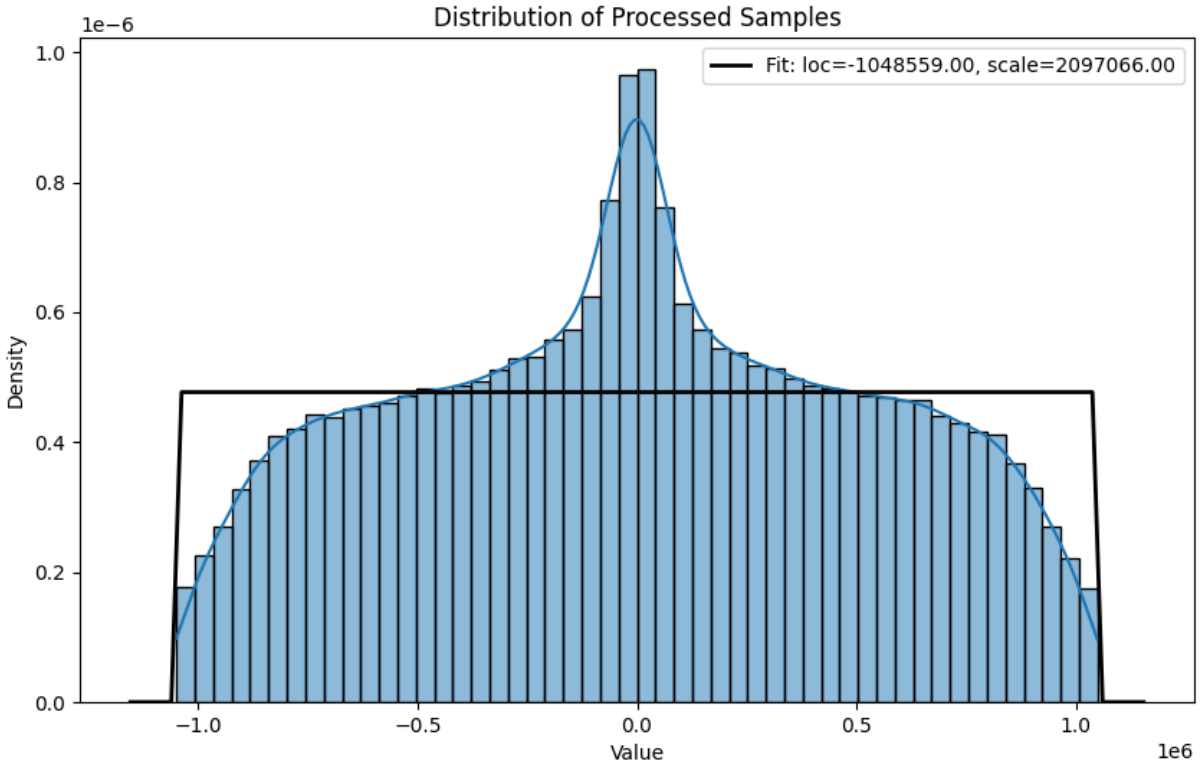


But that's still not a perfect model, because it uses a fixed average distance of improvement. In the real world, the very last step of the roll_sub_array function can provide the smaller distance to the mean.

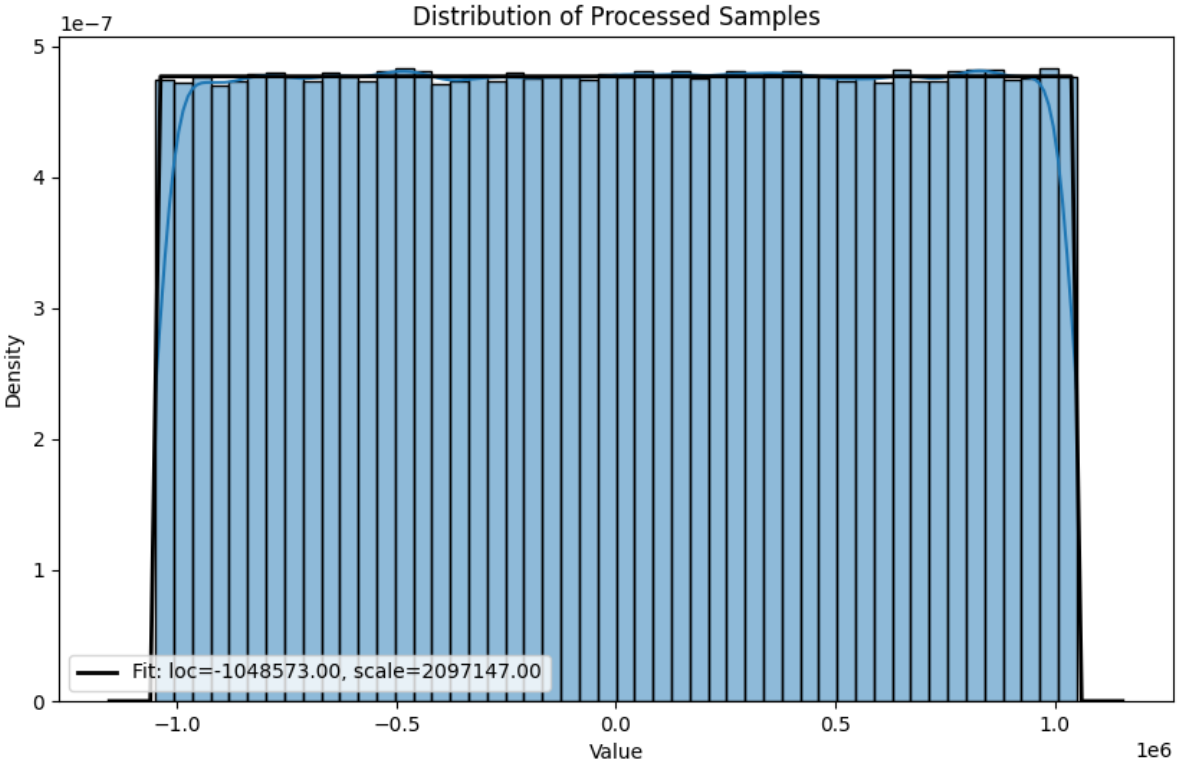
Hereafter are the distributions of the a_i before and after the circular mean centering:



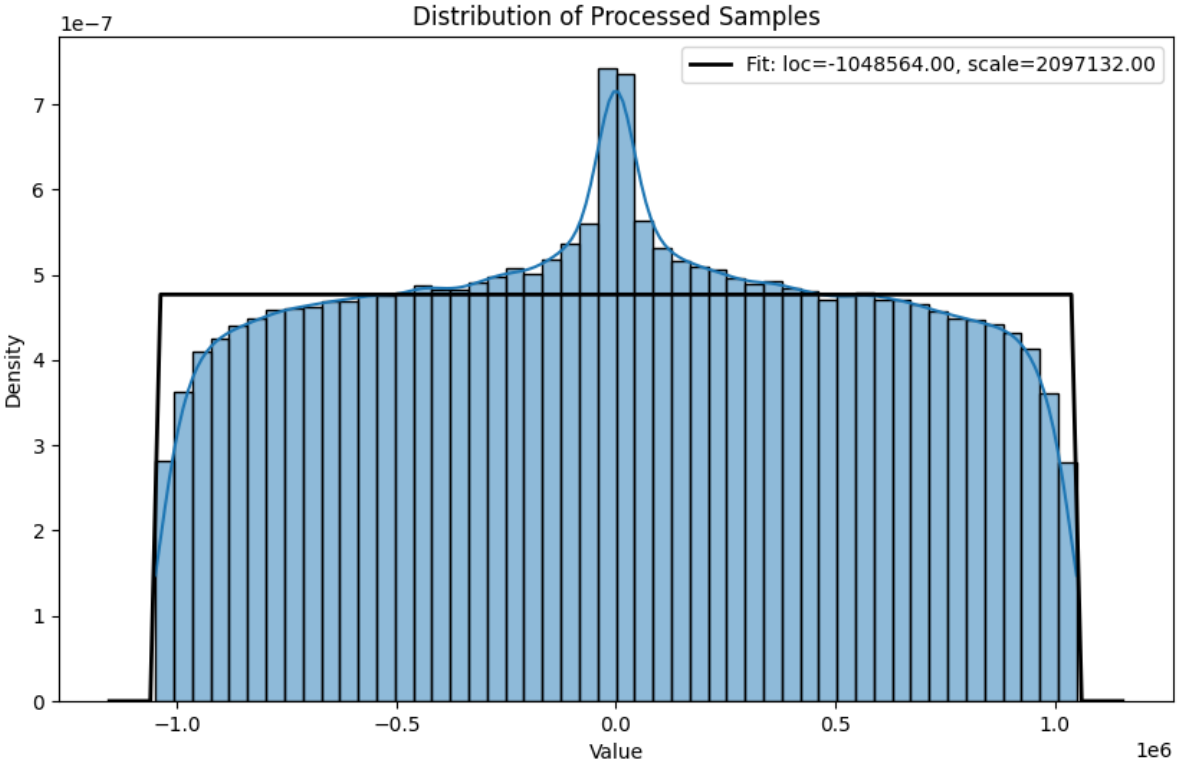
uniform distribution for $n = 20, q = 2^{21}$ and 10 000 samples.



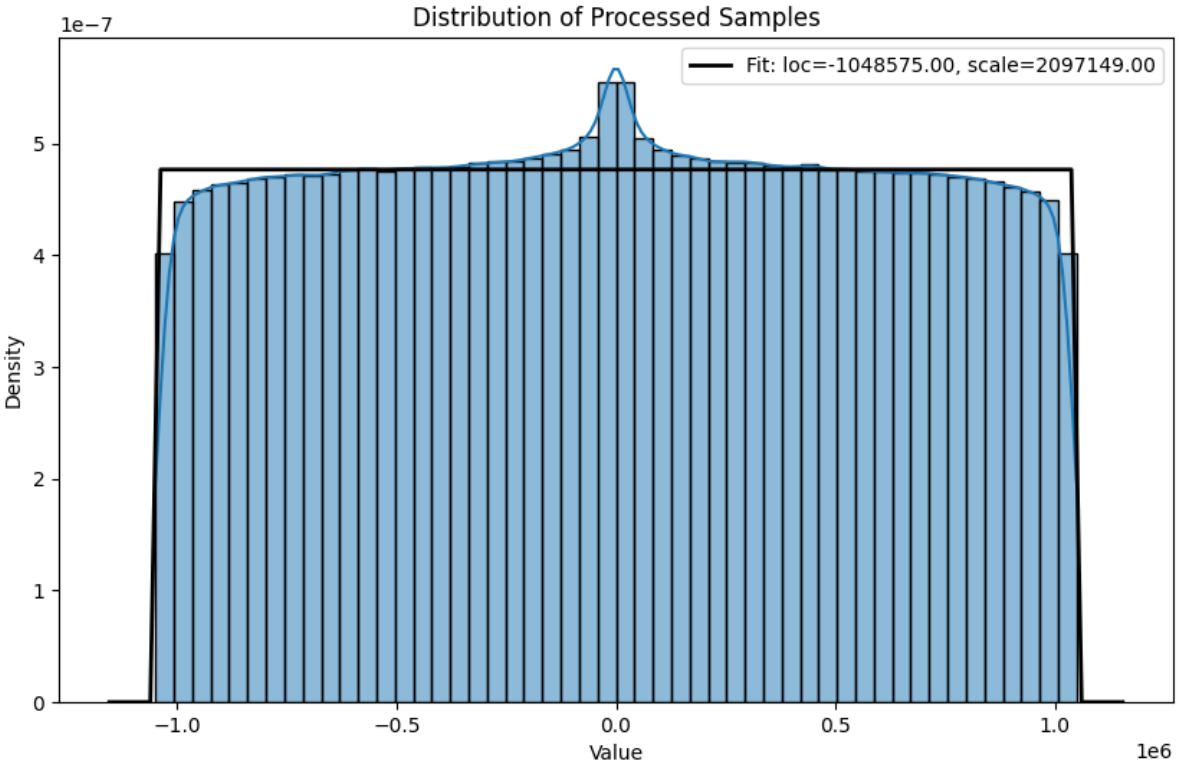
distribution after the transformation for $n = 20, q = 2^{21}$ and 10 000 samples.



uniform distribution for $n = 100, q = 2^{21}$ and 10 000 samples.

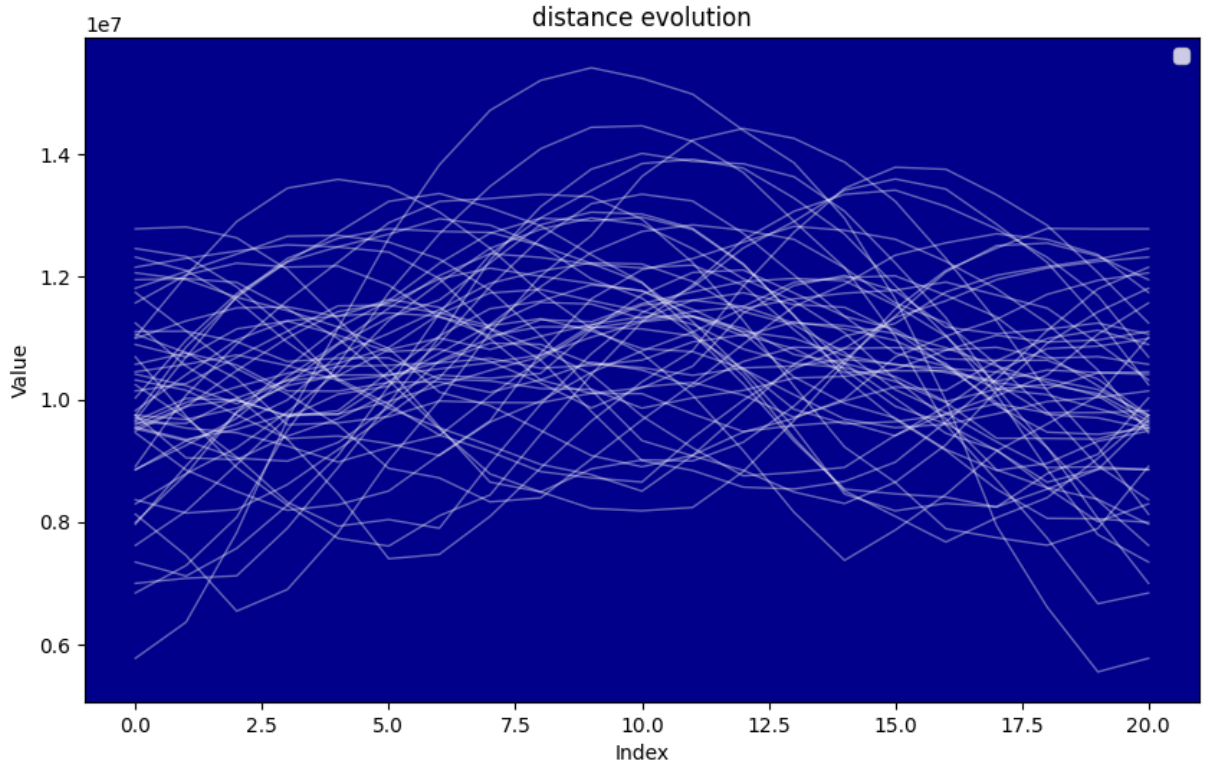


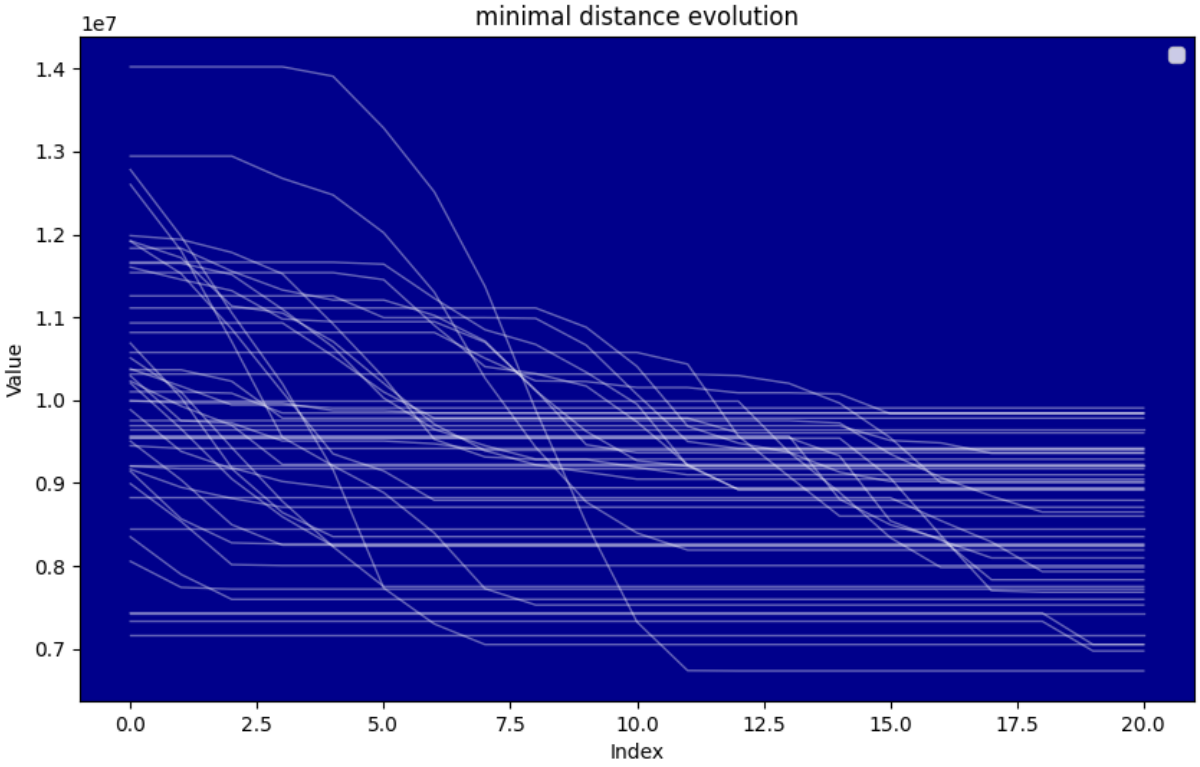
distribution after the transformation for $n = 100, q = 2^{21}$ and 10 000 samples.

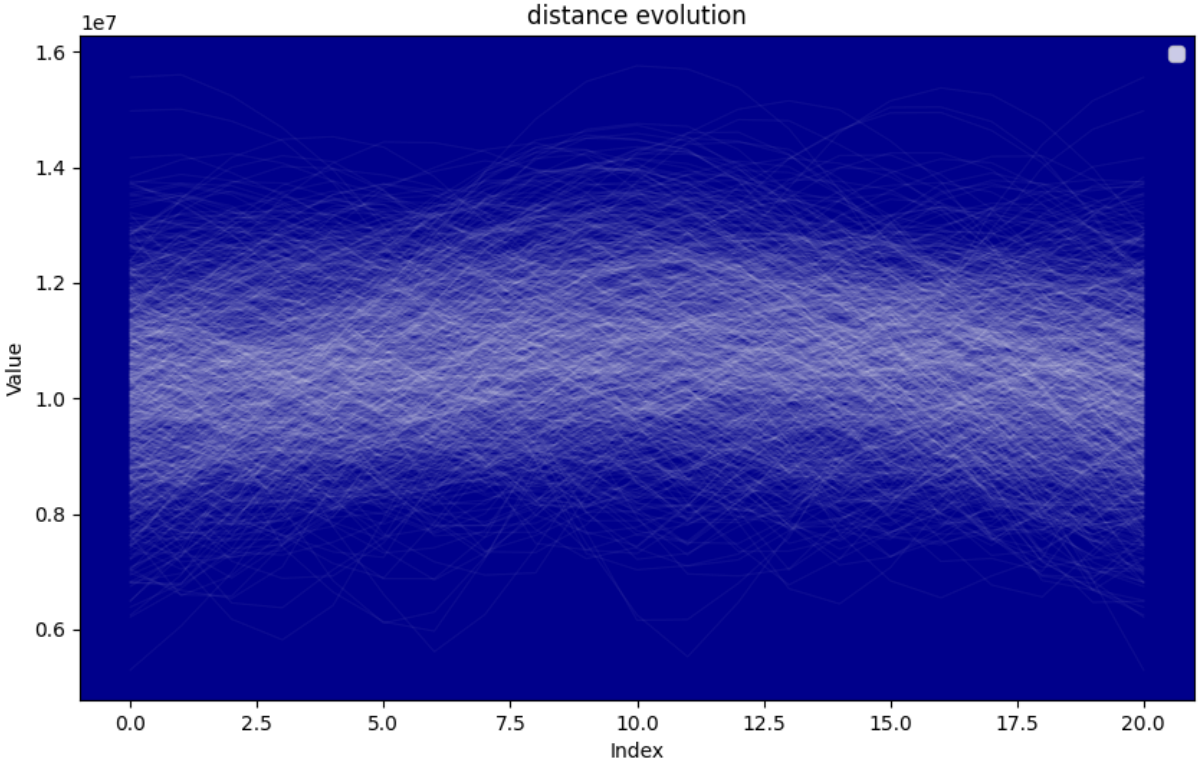


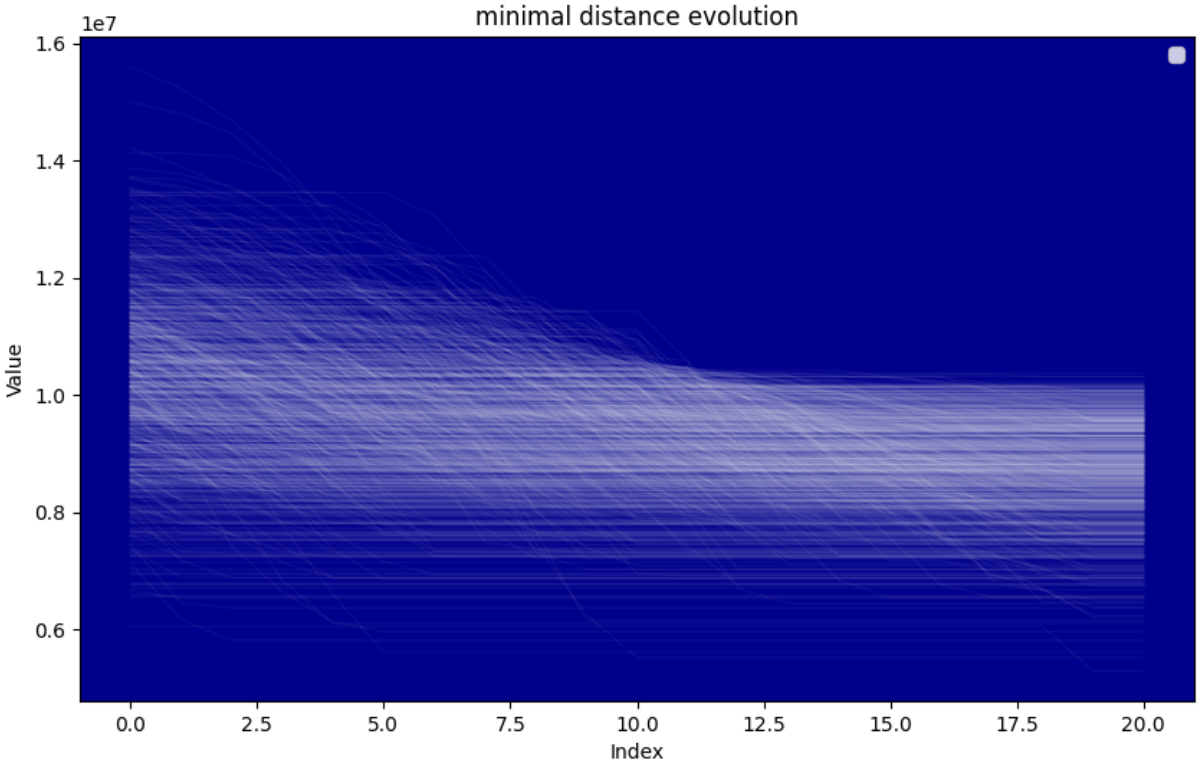
> distribution after the transformation for $n = 1000, q = 2^{21}$ and 10 000 samples.

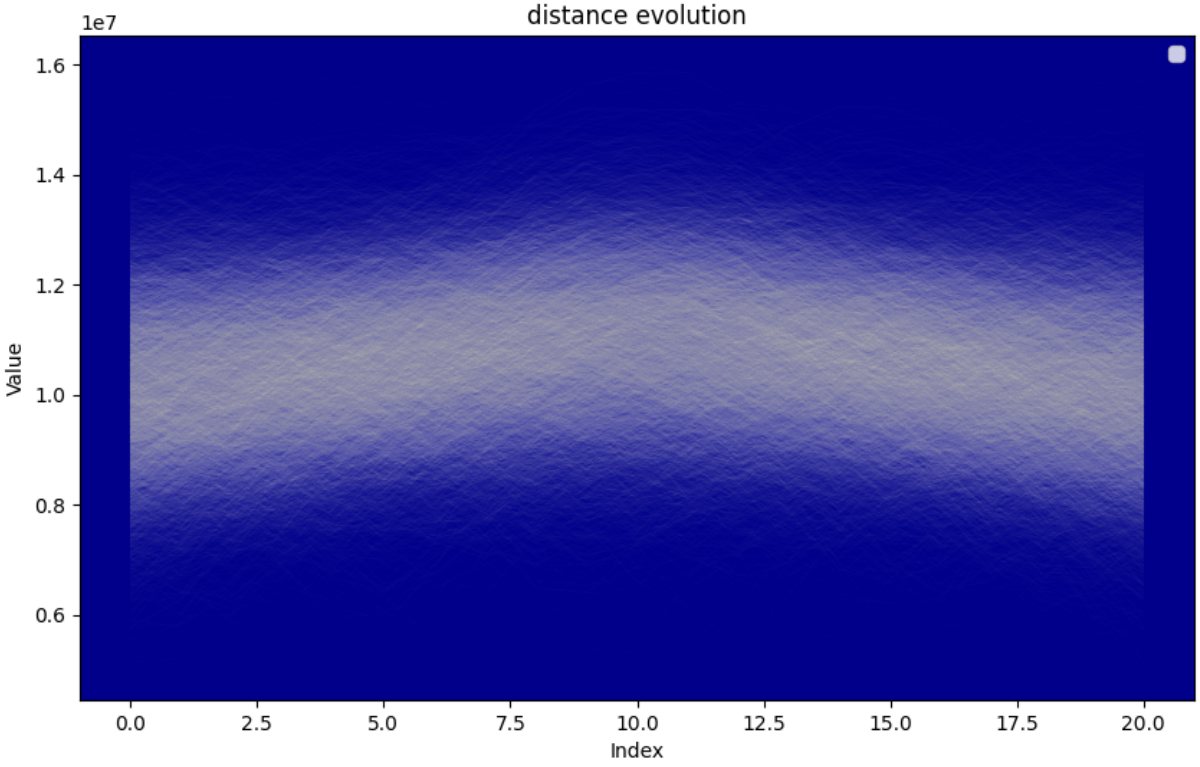
Hereafter the evolution of the distance $d = \sum_i |a_i - \frac{\sum_i a_i}{n}|$ and the evolution of the minimal distance during the algorithm searching for the circular mean.

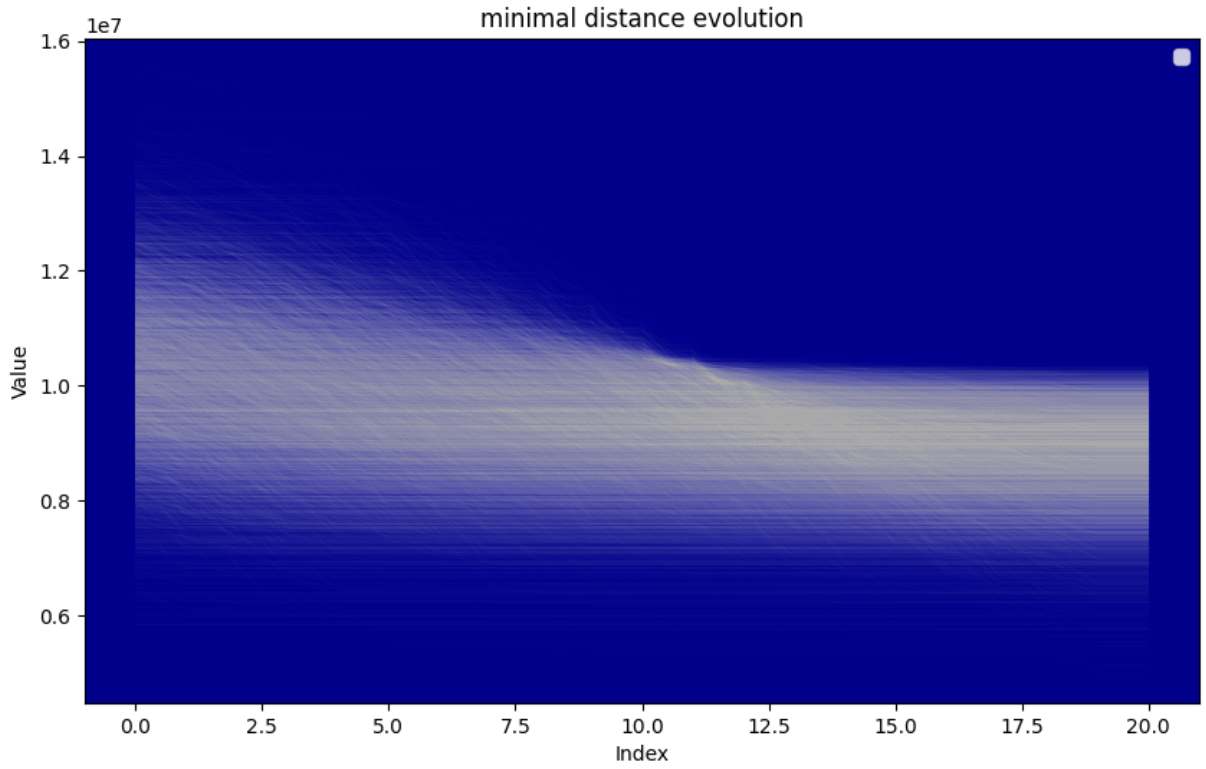



minimal distance evolution
1e7
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
Value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Index


distance evolution
1e7
1.6
1.4
1.2
1.0
0.8
0.6
Value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Index


minimal distance evolution
1e7
1.6
1.4
1.2
1.0
0.8
0.6
Value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Index


distance evolution
1e7
1.6
1.4
1.2
1.0
0.8
0.6
Value
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Index



The analysis shows that for $q = 2^{21}$ and $n = 20$ the average distance at the beginning of the circular mean centering is approximately 10238627 and the average distance at the end of the circular mean centering is 8740182, thus the distance reduction is 1498445 which is approximately $2^{20.515}$. The distance reduction is thus close to $\frac{q}{\sqrt{2}}$

Our model is thus not very good, the average distance reduction by step is $\frac{q}{n\sqrt{2}}$ and the simulation seems to show that on average the search tends to increase the distance at mid procedure rather than reducing it, yet a clear reduction of the distance is observed. A potential explanation is the skewness of the distribution of the distances at the beginning of the circular mean centering procedure. The smallest distances tend to be further from the mean than the greatest distances.

implementation

Hereafter is a first simple version of the algorithm that works for small keys, knowing only the total hamming weight of the key.

For 2000000 samples, with a noise having a standard deviation of 50 and a threshold of 65.3 which means samples are selected if $\log_2(\sum_i |a_i|) \leqslant 65.3$, you should get a result looking like that:

```
moyenne des 1:7267800886094045
moyenne des 0:-7057490744559746
moymoy       :-0.5
standard dev :0


   indice    : 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
  round_key : 0  1  1  0  1  1  1  1  0  1  0  0  0  1  1  1  0  0  1  1


round_key_2 : 0  1  1  0  1  1  1  1  0  1  0  0  0  1  1  1  0  0  1  1

     vrai s : 0  1  1  1  1  1  1  1  0  0  0  0  0  1  1  1  0  0  0  0

differences : V  V  V  X  V  V  V  V  V  X  V  V  V  V  V  V  V  V  X  X

differences2: V  V  V  X  V  V  V  V  V  X  V  V  V  V  V  V  V  V  X  X
compteur:540
nombre d'echantillons:2000000
```

V is for a match between the real key and the key found, and X is for a mismatch. Lowering the threshold gives better results, but you need more samples.

```
use rand::Rng;
use rand::rng;
use rand_distr::{Uniform, Normal, Distribution};
use rayon::prelude::*;
use std::convert::TryInto;

fn roll_subarray<const K: usize, const K_1: usize>(array: &mut [i64; K_1],
start: usize, end: usize) -> i64 {
    let m = end - start;
    assert!(m <= K, "m must be less than or equal to K");

    let mut temp_array: Vec<i128> = array[start..end].iter().map(|&x| x as
i128).collect();
    let mut best_distance = i128::MAX;
    let mut best_array = temp_array.clone();
    let mut final_mean = 0;

    for _ in 0..m {
        let mean: i128 = temp_array.iter().sum::<i128>() / (m as i128);
        let distance: i128 = temp_array.iter().fold(0, |acc, &x| acc + (x -
mean).abs());

        if distance < best_distance {
            //println!("distance:{}",(distance as f64).log2());
            best_distance = distance;
            best_array.copy_from_slice(&temp_array);
```

```
            final_mean = mean;
        }

        if let Some((index, _)) = temp_array.iter().enumerate().min_by_key(|
&(_, &x)| x) {
            temp_array[index] += 1i128 << 64;
        }
    }

    let mod_mask = (1i128 << 64) - 1;

    array[start..end].iter_mut().zip(best_array.iter())
        .for_each(|(a, &v)| *a = ((v - final_mean) & mod_mask) as i64);

    (final_mean & mod_mask) as i64
}


//main single hamming weight
//*
fn main() {
    // Taille du vecteur et des tableaux
    let n: usize = 4_000_000; // Taille de l'échantillon
    const K: usize = 20; // dimension du vecteur lwe
    const K_1: usize = 21;
    let hamming = 10; // Nombre de valeurs non nulles dans s
    let threshold = 65.3;

    // Distribution uniforme pour les valeurs a[i]
    let uniform_dist = Uniform::new(i64::MIN, i64::MAX).expect("Failed to
create uniform distribution");

    // Distribution gaussienne pour l'erreur e
    let sigma = 50.0;
    let normal_dist = Normal::new(0.0, sigma).expect("Failed to create normal
distribution");



    // Générer le tableau s de taille K avec un nombre spécifié de valeurs
non nulles
    let mut s = vec![0; K];
    let mut t = 0;
    while t < hamming {
```

```
        let index = (rng().random::<u64>() % (K as u64)) as usize;
        if s[index] == 0
        {
            s[index] = 1;
            t+= 1;
        }

    }


    let mut vect: Vec<[i64; K + 1]> = (0..n)
        .into_par_iter() // Utilisation de rayon pour paralléliser
        .map(|_| {
            // Créer un générateur de nombres aléatoires pour chaque thread
            let mut rng = rand::rng();
            let mut array: [i64; K + 1] = [0; K + 1];

            // Générer les K premières valeurs a[i]
            for i in 0..K {
                array[i] = uniform_dist.sample(&mut rng);
            }


            // Calculer la somme pondérée avec une erreur gaussienne
            let sum: i64 = s.iter().zip(&array[..K]).map(|(&si, &ai)| si *
ai).sum();
            let error: i64 = normal_dist.sample(&mut rng) as i64;
            let last_value = sum + error;

            // Ajouter la valeur calculée au tableau
            array[K] = last_value;

            array
        })
        .collect();

    // Moyenne circulaire
    //*
    vect.par_iter_mut().for_each(|x|{
        let mean = roll_subarray::<K,K_1>( x, 0, K);
        x[K] = x[K] - (((mean as i128)*(hamming as i128)) as i64);
    });
    // */
```

```
        // Fonction pour sélectionner les tableaux en parallèle


    let mut selected_vect: Vec<[i64; K + 1]> = vect.into_par_iter()
        .filter(|&array|{
            let acc = array[..K].iter().map(|&x| x.abs() as
i128).sum::<i128>() as i128;
            (acc as f64).log2() < threshold
        })
        .collect();



    let mut b:Vec<i128> =vec![0;K];
    let mut moyenne_a = vec![0i128;K];

    let len = selected_vect.len();
    for j in 0..len
    {
        moyenne_a.iter_mut().zip(selected_vect[j].iter()).for_each(|(x,y)|
{ *x= *x + (*y).abs() as i128 } );
        b.iter_mut().zip(selected_vect[j].iter()).for_each(|(x,y)|{*x= *x +
(((*y).signum())*selected_vect[j][K]) as i128 } );
    }
    println!();



    let mut max=moyenne_a[0];
    moyenne_a.iter_mut().for_each(|x|{if *x > max{max = *x};});
    let moy_b:i128 = (b.iter().sum::<i128>())/(K as i128);
    b.iter_mut().for_each(|x|{*x = *x -moy_b ;println!("b {}",*x)});

    b.iter_mut().zip(moyenne_a.iter_mut()).for_each(|(x,y)|{
        if *y != 0
        {
            let temp= (*x)*((max)/(*y));
            if (*x) > 0 && temp < 0 || (*x) < 0 && temp > 0
            {
                println!("parité{}",*x);
            }
            *x = ((*x as f64)*((max as f64)/(*y as f64))) as i128;
        }
    });
```

```
    let mut r:Vec<f64> = vec![0.0;K];

    println!();
    println!(" faux s: ");

    for i in 0..K
    {
        r[i]= (b[i] as f64)/((K*len) as f64) ;  //-0.5;
        println!(" {} :{} ",i+1,r[i]);
    }

    let mut moy0:f64 =0.0;
    let mut moy1:f64 =0.0;
    let mut standard_dev=0.0;
    let mut compt0:f64 =0.0;
    let mut compt1:f64 =0.0;
    for i in 0..K{
        if s[i] == 1 {
            moy1+=r[i];
            compt1+=1.0;
        }else{
            moy0+=r[i];
            compt0+=1.0;
        }
    }

    standard_dev = standard_dev/(K as f64-1.0);
    standard_dev = standard_dev.sqrt();
    let moymoy =-0.5;
    moy0=moy0/compt0;
    moy1=moy1/compt1;
    println!("moyenne des 1:{}",moy1);
    println!("moyenne des 0:{}",moy0);
    println!("moymoy       :{}",moymoy);
    println!("standard dev :{}",standard_dev);
    println!();
    let mut round_key:Vec<i64> = vec![0;K];
    let mut round_key_2:Vec<i64> = vec![0;K];
        for i in 0..K{
            if r[i] < 0.0{
                round_key[i]=0;
            }
```

```
            else{
                round_key[i]=r[i].round() as i64;
            }
            if round_key[i] > 1{
                round_key[i]=1;
            }
        }
        for i in 0..K{
            if r[i]-moymoy < 0.0{
                round_key_2[i]=0;
            }
            else{
                round_key_2[i]=(r[i]-moymoy).round() as i64;
            }
            if round_key_2[i] > 1{
                round_key_2[i]=1;
            }
        }

    println!();
    print!("   indice   :");
    for i in 0..K{
        if i+1<10{
            print!(" {} ",i+1);
        }else{
            print!("{} ",i+1);
        }

    }
    println!();
    print!("  round_key :");
    for i in 0..K{
        print!(" {} ",round_key[i]);
    }
    println!();
    println!();
    println!();
    print!("round_key_2 :");
    for i in 0..K{
        print!(" {} ",round_key_2[i]);
    }
    println!();
    println!();
```

```
    print!("     vrai s :");
    for i in 0..K{
        print!(" {} ",s[i]);
    }

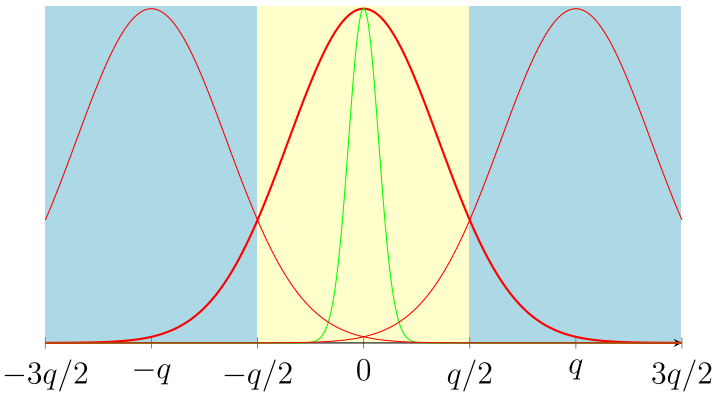
    println!();
    println!();

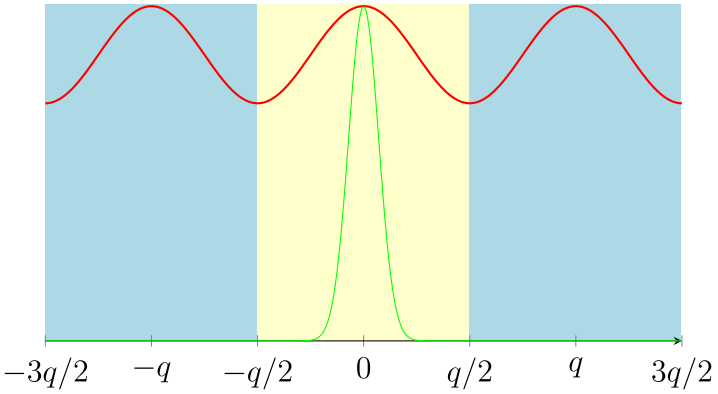
    print!("differences :");
    for i in 0..K{
        if round_key[i]-(s[i] as i64)==0{
           print!(" V ");
        }else{
           print!(" X ");
        }
    }
    println!();
    println!();

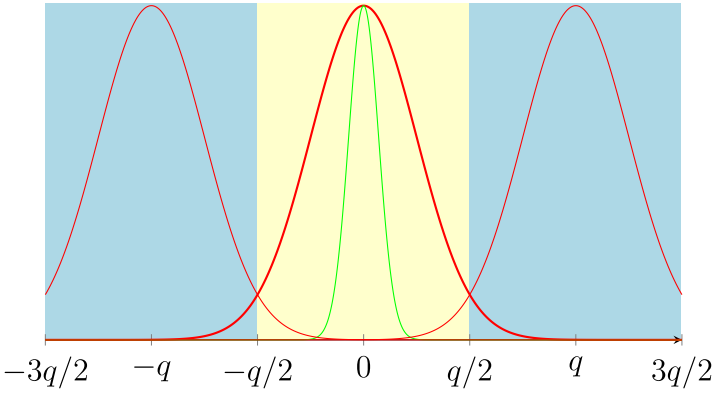
    print!("differences2:");
    for i in 0..K{
        if round_key_2[i]-(s[i] as i64)==0{
           print!(" V ");
        }else{
           print!(" X ");
        }
    }
    println!();
    println!("compteur:{}",len);
    println!("nombre d'echantillons:{}",n);

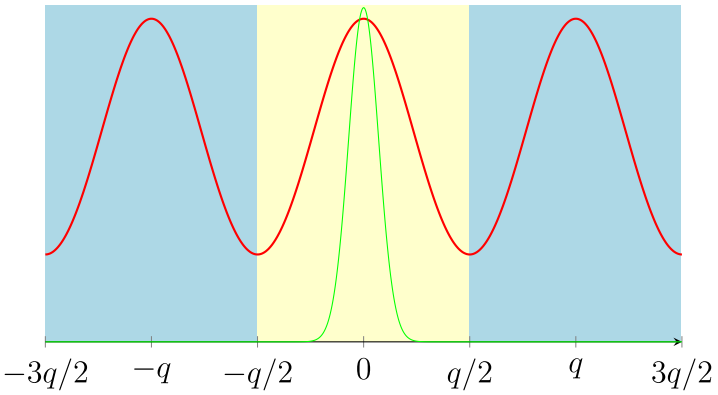
}

// */
```









Instead of assuming we know the hamming weight of the key, we now assume we know the hamming weights of chunks of arbitrary size the key. In a real life situation, instead of assuming we know them, we can just try all the possible hamming weigts of each chunk. For example, if the key is of sie 60, and we have 3 chunks of eual size, then the chunks have a size of 20, then the key can have either 0,1,2 or ...20 non zero bits. Thus in a real

life situation, we can just try all the possible hamming weights, and apply the same algorithm. That brute force method would multiply the duration of the calculation by $21^{\text{nb chunks}}$, although, it’s still unlikely that 20 bits in a row would all be 0 or 1.

In practice though, the bigger the chunk is, the smaller is the relative standard deviation of the number of non zero bits in that chunk. i.e. For a 100 bits key chunk, the average hamming weight is 50, with a $\sigma = \sqrt{\frac{100}{4}} = 5$. With a 95% confidence, the hamming weight of the chunk is thus between 40 and 60.

If the chunk has a size of 20 bits, you can expect that approximately 60% of the hamming weights to be between 7.77 and 12.23, and 95% of them to be between 5.54 and 14.46. So in most cases, the brute force method would likely end before trying $10^{\text{nb chunks}}$ different hamming weights.

Dividing the key in chunks of aritrary size allows the number of valid samples to increase by a lot.

In the code, the function that finds the circular mean is called "roll_subarray". We apply the "roll_subarray" function to chunks of size 20 for each sample. The function is linear in $O(n)$ with n the sie of the key, so for m samples, that’s $O(mn)$. Some parallelism might reduce that but each key chunk is too small to suffer the overhead provoked by parallelism.

The roll_subarray function finds the circular mean of each chunk and center all the sample values around that mean.

The function allows to slightly transform the distribution, then, we can make linear combinations between transformed samples to increase the likelihood of finding "small" samples little by little.

We start with 40 000 samples, $q = 2^{64}, \sigma = 10, n = 60, h = 30$ and a threshold of 65.6. We recovered 55 bits of the key, to recover 60 bits, the algo needs to start with more samples and or have a lower threshold. Let’s go higher.

```
  indice    : 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
 round_key : 0  1  1  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  1  1  0  0  1  0  1  1  0  0  0  0  1  1  1  1  0  0  1  0  1  1  0  1  1  0  0  1  1  0  1  0  0  0  0  1  0  1  0  1  1


round_key_2 : 0  1  1  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0  0  1  1  0  0  1  0  1  1  0  0  0  0  1  1  1  1  0  0  1  0  1  1  0  1  1  0  0  1  1  0  1  0  0  0  0  1  0  1  0  1  1

    vrai s : 1  1  1  1  0  1  0  1  0  0  1  0  1  0  0  0  0  0  1  1  1  0  0  1  0  1  1  0  0  0  0  1  1  1  1  0  0  1  0  1  1  0  1  1  0  0  1  1  0  1  0  0  0  0  1  0  1  0  1  1

differences : X  V  V  X  V  X  V  V  V  V  X  V  V  V  V  V  V  V  X  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V

differences2: X  V  V  X  V  X  V  V  V  V  X  V  V  V  V  V  V  V  X  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V  V
compteur:5345
nombre d'echantillons:40000
```

Twin samples

As explained previously, the data treatment needs to have samples whose components are "small" to recover the key.

By using the hamming weight, we can also recover a key using samples whose components are "close" to their mean, i.e. samples which have only small variations.

The probability of picking up two samples having close components up to a range $\Delta < q$ is

$$\frac{\Delta^n}{q^n}$$

The probability of not picking up two samples having close components up to a range $\Delta < q$ is

$$1 - \frac{\Delta^n}{q^n}$$

In a set of m samples, the probability of having no pair of samples having close components up to a range $\Delta < q$ is

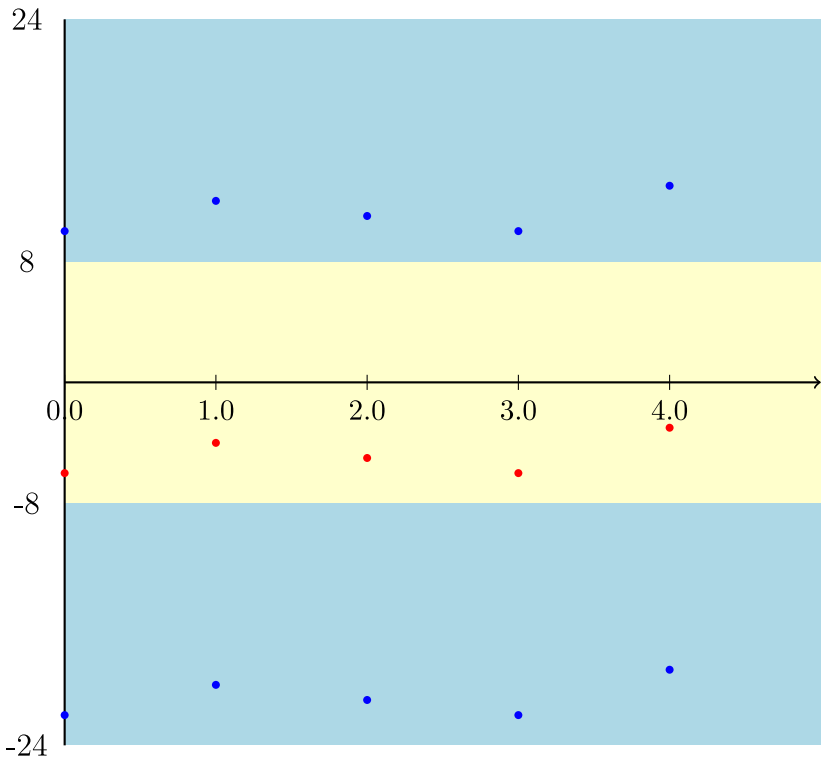
$$(1 - \frac{\Delta^n}{q^n})(1 - \frac{2\Delta^n}{q^n})(1 - \frac{3\Delta^n}{q^n})...(1 - \frac{(m-1)\Delta^n}{q^n}) = e^{-\sum_{k=1}^{m-1} \ln(1-\frac{k\Delta^n}{q^n})} \approx e^{-\frac{m(m-1)}{2q^n}} \approx e^{-\frac{m^2}{2q^n}}$$

So the probability of finding at least one pair is

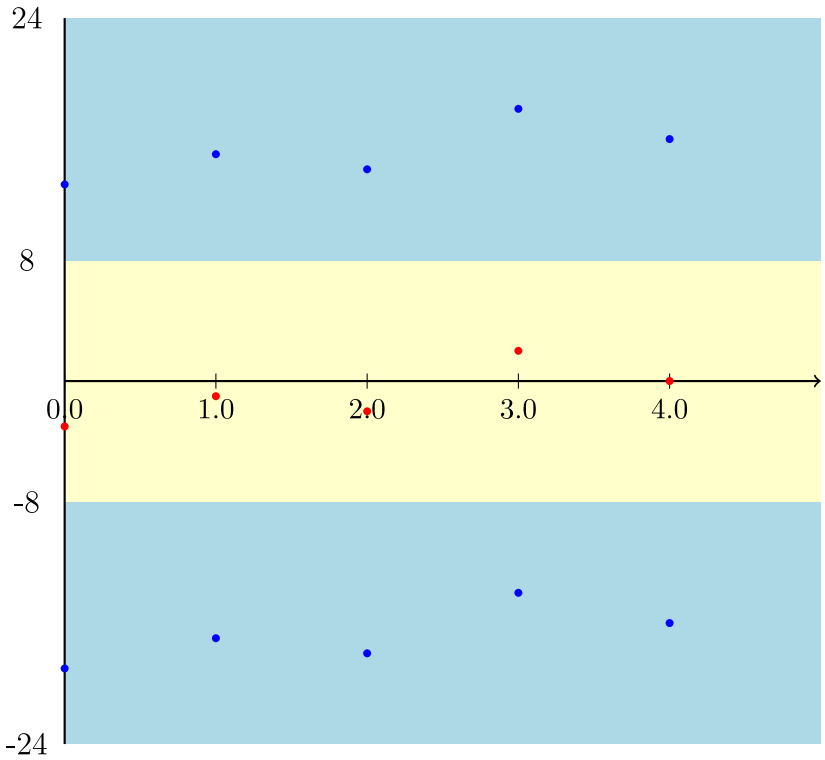
$$1 - e^{-\frac{m(m-1)}{2q^n}} \approx 1 - e^{-\frac{m^2}{2q^n}}$$

and the average number of pairs is equal to the average number of collisions

$$m(1 - (1 - \frac{\Delta^n}{q^n})^{m-1}) \approx m(1 - e^{-\frac{(m-1)\Delta^n}{q^n}})$$



> sample with small variations between components



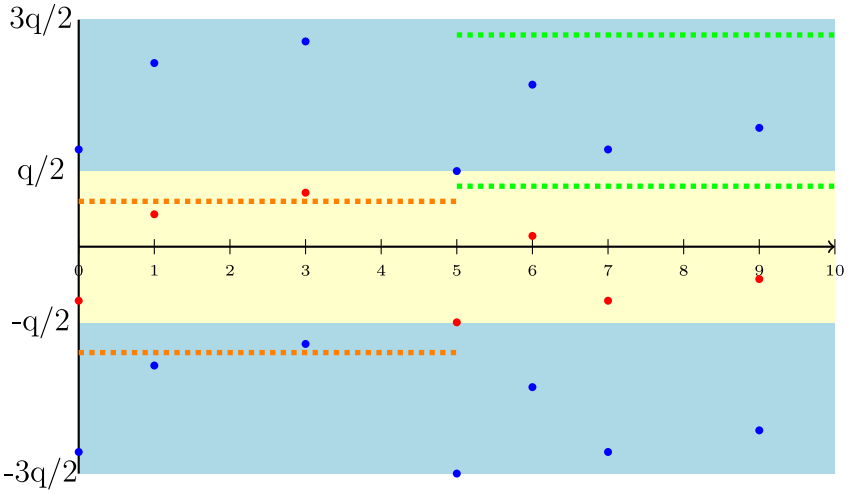
We can increase furthermore the number of acceptable samples by using linear combinations between samples. By making linear combinations of samples which only have close variations, (neither small nor having small components), and using the hamming weight, we can get more samples. We call them twin samples. Twin samples are samples whose components have approximately the same variations. For example, with the modulus 16 and representatives in $[\![-8, 7]\!]$ and two samples, $A = (3, -2, -7, -6, 5)$ and $B = (-1, -7, 3, 5, 2)$, $A - B = (4, 5, 6, 5, 3)$. The variations between the first 4 components are rather small, (1,1,-1), so we call them twin samples. By knowing the hamming weight, we can change it to $(-1, 0, 1, 0, -2 - 5h)$ by removing 5 to each component and modifying the last components $a_4'' = a_4 - a_4' - 5h$.

The new algorithm is described at high level here:

$n \leftarrow$ size of each sample
$N \leftarrow$ Number of chunks
$R \leftarrow$ Number of rounds
$Vec_1 \leftarrow$ vector containing m LWE samples
$Vec_2 \leftarrow$ copy of Vec_1
for $k \in 0..N$ **do**
  for $i \in 0..R$ **do**
    $\forall$ samples
    $size[i] \leftarrow \sum_{j=0}^{c_{kn/N}} |a_j|_{Vec_1}$
    make linear combinations (addition or subtraction) between samples of Vec_1 and Vec_2
    $size_{temp}[i] \leftarrow \sum_{j=0}^{c_{kn/N}} |a_j|_{\text{linear combination}}$
    if $size_{temp}[i] < size[i]$ **then**
      sample $\leftarrow$ linear combination
      $size[i] \leftarrow size_{temp}[i]$
    end if
  end for
end for

The idea is to first reduce the distance $\sum_i |a_i|$ of the first chunk by linear combination until it becomes very small. Since we do not look at the other components, they keep being distributed with a uniform distribution.

Then, the same method is applied to the first two chunks, then to the first three chunks etc...



The new version of the program should use a Cargo.toml file looking like that:

```
[package]
name = "attack_find"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.9.1"
rand_distr = "0.5.1"
rayon = "1.10.0"
tikv-jemallocator = "0.6"

[profile.release]
codegen-units = 1
lto = "fat"
```

and to run it, I suggest using the command:

```
RUSTFLAGS="-C target-cpu=native" cargo build --release
```

```
// */
#[global_allocator]
static GLOBAL: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;
use rand::Rng;
use rand::rng;
```

```
use rand_distr::{Uniform, Normal, Distribution};
use rayon::prelude::*;
use std::sync::atomic::{AtomicUsize, Ordering};

#[inline(always)]
fn find_a_twin_sub<const K: usize, const K_1: usize>(
    array_1: &mut [i64; K_1],
    array_2: &mut [i64; K_1],
    hi: usize,
    start: usize,
    end: usize
) -> i128 {
    let mut temp_array_1 = *array_1;
    let temp_array_2 = *array_2;

    let mut best_distance: i128 = temp_array_1[start..end].iter().fold(0, |
acc, &x| acc + x.abs() as i128);
    temp_array_1.iter_mut().zip(temp_array_2.iter()).for_each(|(x, y)| *x -=
*y);

    let mean = roll_subarray::<K, K_1>(&mut temp_array_1, start, end);
    temp_array_1[K] -= mean * (hi as i64);

    let distance: i128 = temp_array_1[start..end].iter().fold(0, |acc, &x|
acc + x.abs() as i128);

    if distance < best_distance {
        best_distance = distance;
        *array_1 = temp_array_1;
    }
    best_distance
}

#[inline(always)]
fn find_a_twin_add<const K: usize, const K_1: usize>(
    array_1: &mut [i64; K_1],
    array_2: &mut [i64; K_1],
    hi: usize,
    start: usize,
    end: usize
) -> i128 {
    let mut temp_array_1 = *array_1;
    let temp_array_2 = *array_2;
```

```
    let mut best_distance: i128 = temp_array_1[start..end].iter().fold(0, |
acc, &x| acc + x.abs() as i128);
    temp_array_1.iter_mut().zip(temp_array_2.iter()).for_each(|(x, y)| *x +=
*y);

    let mean = roll_subarray::<K, K_1>(&mut temp_array_1, start, end);
    temp_array_1[K] -= mean * (hi as i64);

    let distance: i128 = temp_array_1[start..end].iter().fold(0, |acc, &x|
acc + x.abs() as i128);

    if distance < best_distance {
        best_distance = distance;
        *array_1 = temp_array_1;
    }
    best_distance
}

//*
fn reduce_by_twin<const K: usize, const K_1: usize>(
    vec_1: &mut Vec<[i64; K_1]>,
    vec_2: &mut Vec<[i64; K_1]>,
    n: usize,
    h_v: &[usize],
    nb_chunck: usize
) {
    let h_vec = h_v.to_vec();
    let h_len = h_vec.len();
    let len = vec_1.len();
    let l = vec_2.len();
    let num_threads = rayon::current_num_threads();
    let progress = AtomicUsize::new(0);

    for k in 0..h_len {
        let f = (k + 1) * K / nb_chunck;
        let ini = k * K / nb_chunck;

        for r in 1..(n + 1) {
            let chunck_size = (len + num_threads - 1) / num_threads;
            let chunks: Vec<(usize, usize)> = (0..num_threads)
                .map(|i| (i * chunck_size, (i + 1).min(len / chunck_size + 1)
* chunck_size))
```

```
                .collect();

            let results: Vec<Vec<[i64; K_1]>> = chunks.into_par_iter().map(|
(start, end)| {
                let mut local_vec_1 = vec_1[start..end.min(len)].to_vec();
                let mut local_vec_2 = vec_2.clone();
                let mut vec_sub = local_vec_1.clone();
                let mut vec_add = local_vec_1.clone();
                let vec_ta = local_vec_1.clone();

                for i in 0..local_vec_1.len() {
                    let mut vec_temp_sub = vec_sub[i];
                    let mut vec_temp_add = vec_add[i];
                    let vec_tamp = vec_ta[i];
                    let mut bd = local_vec_1[i][ini..f].iter().fold(0, |acc,
&x| acc + x.abs() as i128);

                    for j in (start + i + 1)..l {
                        let ds = find_a_twin_sub::<K, K_1>(&mut vec_temp_sub,
&mut local_vec_2[j], h_vec[k], ini, f);
                        let da = find_a_twin_add::<K, K_1>(&mut vec_temp_add,
&mut local_vec_2[j], h_vec[k], ini, f);

                        if ds < bd && ds < da {
                            let bd_2 = local_vec_1[i][0..ini].iter().fold(0,
|acc, &x| acc + x.abs() as i128);
                            let ds_2 = vec_temp_sub[0..ini].iter().fold(0, |
acc, &x| acc + x.abs() as i128);
                            if ds_2 <= bd_2
                            {
                                bd = ds;

local_vec_1[i].copy_from_slice(&vec_temp_sub);
                            }
                        } else if da < bd {
                            let bd_2 = local_vec_1[i][0..ini].iter().fold(0,
|acc, &x| acc + x.abs() as i128);
                            let da_2 = vec_temp_add[0..ini].iter().fold(0, |
acc, &x| acc + x.abs() as i128);
                            if da_2 <= bd_2
                            {
                                bd = da;
```

```
local_vec_1[i].copy_from_slice(&vec_temp_add);
                            }
                        }
                        vec_temp_sub.copy_from_slice(&vec_tamp);
                        vec_temp_add.copy_from_slice(&vec_tamp);
                    }

                    //let progress_val = progress.fetch_add(1,
Ordering::SeqCst);
                    //if progress_val % 10 == 0 {
                        println!("bd:{} r:{} i:{} k:{}", (bd as f64).log2(),
r, i, k);
                    //}
                }

                local_vec_1
            }).collect();

            vec_1.par_chunks_mut(chunck_size)
                .zip(results)
                .for_each(|(chunk, result)| {
                    chunk.copy_from_slice(&result[..chunk.len()]);
                });

            vec_2.copy_from_slice(vec_1);
        }
    }
}

// */



#[inline(always)]
fn roll_subarray<const K: usize, const K_1: usize>(array: &mut [i64; K_1],
start: usize, end: usize) -> i64 {
    let m = end - start;
    assert!(m <= K, "m must be less than or equal to K");

    let mut temp_array: Vec<i128> = array[start..end].iter().map(|&x| x as
i128).collect();
    let mut best_distance = i128::MAX;
    let mut best_array = temp_array.clone();
```

```
    let mut final_mean = 0;

    for _ in 0..m {
        let mean: i128 = temp_array.iter().sum::<i128>() / (m as i128);
        let distance: i128 = temp_array.iter().fold(0, |acc, &x| acc + (x -
mean).abs());

        if distance < best_distance {
            best_distance = distance;
            best_array.copy_from_slice(&temp_array);
            final_mean = mean;
        }

        if let Some((index, _)) = temp_array.iter().enumerate().min_by_key(|
&(_, &x)| x) {
            temp_array[index] += 1i128 << 64;
        }
    }

    let mod_mask = (1i128 << 64) - 1;

    array[start..end].iter_mut().zip(best_array.iter())
        .for_each(|(a, &v)| *a = ((v - final_mean) & mod_mask) as i64);

    (final_mean & mod_mask) as i64
}

fn reduce_hamming_weight<const K: usize, const K_1: usize>(
    sample: &mut [i64; K_1],
    h_v: &[usize],
)
{
    let n = h_v.len();
    let size = K/n;
    for i in 0..n
    {
        if h_v[i] > (size>>1)
        {
            let sum = sample[i*size..(i+1)*size].iter().sum::<i64>();
            sample[K] -= sum;
        }

    }
```

```
}


fn main() {
    let n: usize = 40000;
    const K: usize = 60;
    const K_1: usize = 61;

    let threshold = 65.65;



    let n_chunck = 3;
    let cs= K/n_chunck;
    const NC: usize = 3;
    let mut h_vec = [0; NC];

    let h: usize = 30;
    let n_round = 10;

    let uniform_dist = Uniform::new(i64::MIN, i64::MAX).expect("Failed to
create uniform distribution");
    let sigma = 10.0;
    let normal_dist = Normal::new(0.0, sigma).expect("Failed to create normal
distribution");

    let mut s = vec![0; K];
    let mut t = 0;
    while t < h
    {
        let index = (rng().random::<u64>() % (K as u64)) as usize;
        if s[index ] == 0
        {
            s[index ] = 1;
            t += 1;
        }
    }




    for i in 0..n_chunck
    {
        h_vec[i] = (s[i*cs..(i+1)*cs].iter().sum::<i64>()) as usize;
```

```
    }


    let mut vect: Vec<[i64; K + 1]> = (0..n)
        .into_par_iter()
        .map(|_| {
            let mut rng = rand::rng();
            let mut array: [i64; K + 1] = [0; K + 1];

            for i in 0..K {
                array[i] = uniform_dist.sample(&mut rng);
            }

            let sum: i64 = s.iter().zip(&array[..K]).map(|(&si, &ai)| si *
ai).sum();
            let error: i64 = normal_dist.sample(&mut rng) as i64;
            let last_value = sum + error;

            array[K] = last_value;

            array
        })
        .collect();

    let mut vect_2 = vect.clone();
    reduce_by_twin::<K, K_1>(&mut vect, &mut vect_2, n_round, &h_vec,
n_chunck);

    for mut s in &mut vect
    {
        reduce_hamming_weight::<K,K_1>(&mut s, &h_vec);
    }

    let selected_vect: Vec<[i64; K + 1]> = vect.clone().into_par_iter()
        .filter(|&array| {
            let acc = array[..K].iter().map(|&x| x.abs() as
i128).sum::<i128>() as i128;
            (acc as f64).log2() <= threshold
        })
        .collect();

    let mut b: Vec<i128> = vec![0; K];
    let mut moyenne_a = vec![0i128; K];
```

```
    let len = selected_vect.len();
    for j in 0..len {
        moyenne_a.iter_mut().zip(selected_vect[j].iter()).for_each(|(x, y)|
{ *x = *x + (*y).abs() as i128 });
        b.iter_mut().zip(selected_vect[j].iter()).for_each(|(x, y)| { *x = *x
+ ((*y).signum() * selected_vect[j][K]) as i128 });
    }

    let mut max = moyenne_a[0];
    moyenne_a.iter_mut().for_each(|x| { if *x > max { max = *x } });
    let moy_b: i128 = (b.iter().sum::<i128>()) / (K as i128);
    b.iter_mut().for_each(|x| { *x = *x - moy_b });

    b.iter_mut().zip(moyenne_a.iter_mut()).for_each(|(x, y)| {
        if *y != 0 {
            *x = ((*x as f64) * ((max as f64) / (*y as f64))) as i128;
        }
    });

    let mut r: Vec<f64> = vec![0.0; K];

    for i in 0..K {
        r[i] = (b[i] as f64) / ((K * len) as f64);
    }

    let mut moy0: f64 = 0.0;
    let mut moy1: f64 = 0.0;
    let mut compt0: f64 = 0.0;
    let mut compt1: f64 = 0.0;
    for i in 0..K {
        if s[i] == 1 {
            moy1 += r[i];
            compt1 += 1.0;
        } else {
            moy0 += r[i];
            compt0 += 1.0;
        }
    }

    moy0 = moy0 / compt0;
    moy1 = moy1 / compt1;
```

```
    println!("moyenne des 1: {}", moy1);
    println!("moyenne des 0: {}", moy0);
    println!("compteur: {}", len);
    println!("nombre d'echantillons: {}", n);

    let mut round_key: Vec<i64> = vec![0; K];
    let mut round_key_2: Vec<i64> = vec![0; K];
    for i in 0..K {
        if r[i] < 0.0 {
            round_key[i] = 0;
        } else {
            round_key[i] = r[i].round() as i64;
        }
        if round_key[i] > 1 {
            round_key[i] = 1;
        }
    }
    for i in 0..K {
        if r[i] - (-0.5) < 0.0 {
            round_key_2[i] = 0;
        } else {
            round_key_2[i] = (r[i] - (-0.5)).round() as i64;
        }
        if round_key_2[i] > 1 {
            round_key_2[i] = 1;
        }
    }

    println!("   indice   :");
    for i in 0..K {
        print!(" {}", i + 1);
    }
    println!();

    println!("  round_key :");
    for i in 0..K {
        print!(" {}", round_key[i]);
    }
    println!();

    println!("round_key_2 :");
    for i in 0..K {
        print!(" {}", round_key_2[i]);
```

```
    }
    println!();

    println!("     vrai s :");
    for i in 0..K {
        print!(" {}", s[i]);
    }
    println!();

    println!("differences :");
    for i in 0..K {
        if round_key[i] - (s[i] as i64) == 0 {
            print!(" V ");
        } else {
            print!(" X ");
        }
    }
    println!();

    println!("differences2:");
    for i in 0..K {
        if round_key_2[i] - (s[i] as i64) == 0 {
            print!(" V ");
        } else {
            print!(" X ");
        }
    }
    println!();

    println!("compteur: {}", len);
    println!("nombre d'echantillons: {}", n);
}
```

References

CAMILA C. S. CAIADO, PUSHPA N. RATHIE. “POLYNOMIAL COEFFICIENTS AND DISTRIBUTIONOF THE SUM OF DISCRETE UNIFORM VARIABLES.” *Online*.