Discrete Ziggurat: A Time-Memory Trade-Off for Sampling from a Gaussian Distribution over the Integers

Johannes Buchmann, Daniel Cabarcas,
Florian Göpfert(✉), Andreas Hülsing, and Patrick Weiden

Technische Universität Darmstadt, Darmstadt, Germany
{buchmann,fgoepfert,pweiden}@cdc.informatik.tu-darmstadt.de,
dcabarc@unal.edu.co, a.t.huelsing@tue.nl

Abstract. Several lattice-based cryptosystems require to sample from a discrete Gaussian distribution over the integers. Existing methods to sample from such a distribution either need large amounts of memory or they are very slow. In this paper we explore a different method that allows for a flexible time-memory trade-off, offering developers freedom in choosing how much space they can spare to store precomputed values. We prove that the generated distribution is close enough to a discrete Gaussian to be used in lattice-based cryptography. Moreover, we report on an implementation of the method and compare its performance to existing methods from the literature. We show that for large standard deviations, the Ziggurat algorithm outperforms all existing methods.

Keywords: Lattice-based cryptography · Gaussian sampling · Practicality · Implementation

1 Introduction

The object of study of this paper is the discrete Gaussian probability distribution over the integers. Sampling elements from such a distribution is widely used in lattice-based cryptography [GPV08, LP11, BGV12, GGH12]. It is a critical technical challenge to sample from a discrete Gaussian over the integers accurately and efficiently. Weiden et al. [WHCB13] report that sampling from it takes more than 50 % of the running time of the signing algorithm in their implementation of Lyubashevsky's signature scheme [Lyu12].

All existing methods to sample from a Gaussian distribution over the integers either need large amounts of memory or they are very slow. For example, Galbraith and Dwarakanath estimate that Peikert's sampler [Pei10] requires around 12 MB of storage [GD12] for some parameters. Such a large memory requirement might be acceptable on a PC but not on the diversity of devices that demand cryptographic solutions today.

In this paper we explore a different alternative for sampling from a Gaussian distribution over the integers that offers a flexible trade-off between speed and

T. Lange, K. Lauter, and P. Lisoněk (Eds.): SAC 2013, LNCS 8282, pp. 402–417, 2014.
DOI: 10.1007/978-3-662-43414-7_20,

memory. Moreover, for big standard deviations, this method beats commonly used methods. We call the method discrete Ziggurat because it adapts the Ziggurat algorithm [MT00] for the discrete case.

The discrete Ziggurat is specially appealing for cryptographic applications because of its flexibility. The method uses precomputed rectangles of equal 'size' to cover the area under the probability density function (PDF). Increasing the number of rectangles increases speed but also increases the memory used. Therefore, it offers an easy-to-tune trade-off between speed and memory. This is a desirable property because developers of cryptographic primitives can easily adjust it to fit the particular characteristics of different devices. On memory constraint devices like smartcards or microcontrollers they could use a low-memory low-speed setting, while on a high performing server they could use a high-memory high-speed configuration.

Originally, the Ziggurat sampler was developed for a continuous distribution. In order to adapt it to the discrete case some care must be taken. In particular the notion of 'size' of a rectangle must be redefined from the narrow concept of 'area' to the more general "probability to sample points inside the rectangle". We discuss the implications of this generalization.

It is also challenging to analyze the quality of the discrete Ziggurat because of the subtleties of an actual implementation. In this paper we provide a careful analysis that takes into consideration the loss of precision due to the tail-cut, the precision in sampling from the y-axis and the precision in calculating the PDF. The techniques used and the way they are combined in this analysis might show valuable for the analysis of other samplers. For developers we explain how to achieve a desired accuracy by setting the precision for representing numbers.

We implemented the discrete Ziggurat in C++ using the Number Theory Library (NTL) [Sho]. The implementation can be downloaded at the authors' homepage[1]. We compare the efficiency of the discrete Ziggurat with existing methods and analyze the speed-memory trade-off. For example, we used the parameters proposed by Galbraith and Dwarakanath [GD12] for the normal distribution in Lyubashevsky's signature scheme [Lyu12]. For this illustrative setting, the discrete Ziggurat produces about 1.13 million samples per second, using only 524 KB of memory. In comparison, Peikert's sampler outputs 281,000 samples per second for a memory usage of 33.55 MB. The Knuth-Yao algorithm is only slightly faster (it produces about 4 % more samples), but increases the memory-consumption by a factor of more than 400.

Related Work. We briefly survey existing alternatives to sample from a discrete Gaussian probability distribution over the integers, denoted D_σ. For parameter $\sigma > 0$, D_σ assigns $x \in \mathbb{Z}$ a probability proportional to $\rho_\sigma(x) = \exp(-\frac{1}{2}x^2/\sigma^2)$. It is important to note that sampling from D_σ is different to sampling from

[1] In particular at https://www.cdc.informatik.tu-darmstadt.de/~pschmidt/implementations/ziggurat/ziggurat-src.zip.

a (continuous) normal distribution [TLLV07]. Another related problem is that of sampling from a Gaussian distribution over a generic lattice, a more complex problem, whose solutions often require sampling from D_σ as a subroutine [GPV08, Pei10, DN12, AGHS12].

For cryptographic applications it is sufficient to sample from the bounded subset $B := \mathbb{Z} \cap [-t\sigma, t\sigma]$, where the tailcut $t > 0$ is chosen large enough to guarantee a desired precision [GPV08]. One alternative to sample from D_σ is to do rejection sampling on B. Another alternative is to precompute the cumulative distribution function (CDF) for $x \in B$, sample a uniform element $y \in [0, 1)$ and perform a binary search on the CDF table to output the "inverse CDF" of y [Pei10]. To the best of our knowledge, no work analyzes the accuracy or efficiency of any of these methods in detail.

Yet another alternative, explored by Galbraith and Dwarakanath [GD12], is the Knuth-Yao algorithm. The algorithm first precomputes a binary tree with leaves labeled by the elements of B. For $x \in B$, if the probability of sampling x has a one in the i-th place of its binary representation, there is a leaf labeled x at height i of the tree. Then it samples by walking down the tree using one uniform bit at each step to decide which of the two children to move to. Galbraith and Dwarakanath present a very detailed analysis of the accuracy of the sampler and of the number of random bits it uses. They also propose ways to reduce the memory requirements. However, they do not assess the speed of the sampler.

Ducas and Nguyen propose an enhancement for rejection sampling. They observe that the sampler can compute at a low precision by default and only use high precision computation when a certain threshold is reached [DN12]. To the best of our knowledge, no work evaluates the effect of this enhancement in detail.

Organization. In Sect. 2 we explain the Ziggurat algorithm and we describe in detail its discrete variant. In Sect. 3 we analyze the quality of the distribution. Finally, in Sect. 4 we report on experimental results.

2 The Discrete Ziggurat Algorithm

The Ziggurat algorithm belongs to the class of rejection sampling algorithms and was introduced by Marsaglia and Tsang for sampling from a continuous Gaussian distribution [MT00]. Here we adapt it for the discrete case. After explaining the setting, we give a short overview over Ziggurat in the continuous case and shortly explain how to control the trade-off. Afterwards, we show how we adapted it to the discrete case and explain how to perform the necessary precomputing. Subsequently, we discuss the implementation-details and finish the section with further improvements.

2.1 Setting

We are concerned with sampling from a discrete Gaussian distribution centered at zero with bounded support $B := [-t\sigma, t\sigma] \cap \mathbb{Z}$ for some parameter $t > 0$. This bounded support is sufficient for the application in lattice-based cryptography as long as t is chosen large enough. Moreover, we show in Sect. 3.2 how to select parameters such that the sampled distribution is within a certain statistical distance to a (truly) discrete Gaussian distribution. The assumption that the distribution is centered at zero is also fine, as we can add a constant offset to transform samples into a distribution centered around any other integer.

2.2 Intuition

We briefly review the continuous Ziggurat for the above setting to give some intuition. As the target distribution is symmetric, we can proceed as follows. We use the method to sample a value $x \leq t\sigma$ within $\mathbb{R}_0^+$. Afterwards, if $x = 0$ we accept with probability $1/2$. Otherwise, we sample a sign $s \in \{-1, 1\}$ and return the signed value sx.

Now, how do we sample x within $\mathbb{R}_0^+$? During set-up, we enclose the area of the probability density function (PDF) in an area A consisting of m horizontal rectangles with equal area as shown in Fig. 1. How the rectangles are computed is described below. Next, we store the coordinates (x_i, y_i) of the lower right corner of each rectangle R_i, $1 < i < m - 1$. Please note that each rectangle R_i can be split into a left rectangle R_i^l that lies completely within the area of the PDF and a right rectangle R_i^r that is only partially covered by the PDF. For an example, see R_3 in Fig. 1.

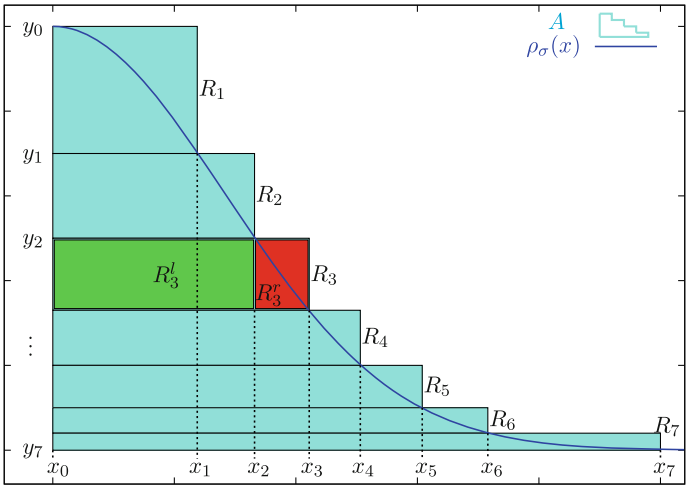


Fig. 1. Ziggurat for $m = 7$ with covering area A and the partition into rectangles.

Now, to actually sample a value $x \leq t\sigma$ within $\mathbb{R}_0^+$ we first sample an integer $1 \leq i \leq m$ uniformly at random, to select a random rectangle. Next, we sample an x-coordinate inside R_i, by sampling a uniformly random x' within $[0, x_i]$. If $x' \leq x_{i-1}$, i.e. if x' is inside R_i^l, we directly accept and return x'. Otherwise, x' lies within R_i^r. In this case, we do rejection sampling. Namely, we sample a value γ within $[y_{i+1}, y_i]$ uniformly at random. Then, if $\gamma + y_{i+1} \leq \rho_\sigma(x')$, i.e. we hit a point in the area below the PDF, we accept and return x'. Otherwise, we reject and restart the whole process by sampling a new i.

In order to understand why this sampling-algorithm works, think of it as an efficient implementation of rejection-sampling in the area A. More precisely, the implementation of the first step (sampling a point in the enclosing area) is improved. Since all the rectangles have equal size, the probabilities of sampling a point in a given rectangle are equal. Therefore, one can sample the rectangle first and a point in the rectangle afterwards.

The expensive part of the algorithm is computing $\rho_\sigma(x')$ if x' does not lie within R_i^l. It becomes even more expensive whenever a value is rejected. For this reason Ziggurat provides a time-memory trade-off, which is controlled by the number of rectangles used, as follows. If we use more rectangles, the ratio between the left and the right rectangle within one rectangle is changed in such a way that the left rectangle becomes comparatively bigger. Hence, we accept an x' without computing $\rho_\sigma(x')$ with higher probability. Moreover, using more rectangles, the area A tighter encloses the area C below the PDF. Thereby, the area $A \setminus C$ that leads to a rejection shrinks and with it the overall probability of a rejection. However, for each additional rectangle the coordinates of one additional point have to be stored, increasing the memory requirements.

2.3 Adaption to the Discrete Case

In the discrete case, the algorithm works quite similar. The whole pseudocode can be found in Appendix A.1. As before, a sign s, a rectangle with index i and a potential sample x' are sampled. If x' lies in a left rectangle and is non-zero, sx' is returned immediately. If x' equals zero, it is returned with probability $1/2$, like in the continuous case. If not, exactly the same rejection sampling procedure as in the continuous case is used to decide whether sx' is returned or the whole process is restarted.

In contrast to the continuous case, the notion of 'size' defined using the area of a rectangle can not be used in the discrete case. We have seen in the last section that the size of a rectangle has to be proportional to the probability to sample a point in it. In the discrete case, we therefore define the size of a rectangle as the number of integer x-coordinates in the rectangle times its height. For instance, the rectangle R_3 has size $(1 + \lfloor x_3 \rfloor) \cdot (y_2 - y_3)$.

The second difference between the continuous and the discrete case is the way the rectangles are computed. While we did not explain how this is done in the continuous case, as it would go beyond the scope of this work, we give a

description for the discrete case. We explain how to obtain a partition for the Ziggurat algorithm for a given number of m rectangles where each rectangle has exactly the same 'size' S. Therefore, we set

$$y_m := 0, \qquad x_0 := 0 \qquad \text{and} \qquad x_m := t\sigma,$$

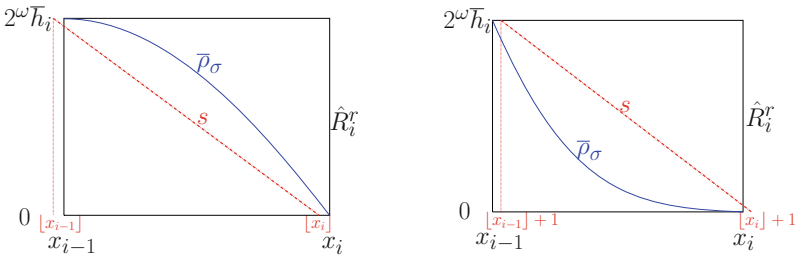
and we iteratively compute a possible partition "from right to left" via

$$\begin{aligned} &y_{m-1} = \frac{S}{1+\lfloor x_m \rfloor}, && x_{m-1} = \rho_\sigma^{-1}(y_{m-1}), \\ &\text{for } i = m-2, \ldots, 1: && y_i = \frac{S}{1+\lfloor x_{i+1} \rfloor} + y_{i+1}, && x_i = \rho_\sigma^{-1}(y_i), \\ &y_0 = \frac{S}{1+\lfloor x_1 \rfloor} + y_1. \end{aligned}$$

Recall that ρ_σ is a scaled density function with $\rho_\sigma(0) = 1$. Therefore, a valid partition for Ziggurat requires $y_0 \geq 1$, since only then the partition completely covers the area under the curve ρ_σ on the support $B_0^+ := [0, t\sigma] \cap \mathbb{Z}_0^+$. Since the value y_0 depends on the 'size' S of the rectangles, any value of S for which $y_0 \geq 1$ leads to a valid partition. We heuristically determine S as follows. We set $S = \sigma/(m \cdot \sqrt{\pi/2}) \cdot c$ with initial value $c = 1$, compute the corresponding partition, and increase c stepwise as long as $y_0 < 1$. (To improve the quality of the input partition, i.e. minimizing $y_0 - 1$, one can perform a binary search for S in $[\sigma/(m \cdot \sqrt{\pi/2}), t\sigma + 1]$.) In the case that no valid partition is found, we increase x_m by one and restart the whole process. Reaching $x_m = (t+1)\sigma$, we abort. We note that this method ended with no partition being output in only about 1.3 % of our computations. In these cases, i.e. when no valid partition is found, one can re-run the procedure with one or more of the following changed: number of rectangles m, Gaussian parameter σ (if possible), or upper bound on x_m.

2.4 Implementation

For an implementation, we have to analyze the effect of computing with limited precision. We use a dash over numbers or functions to indicate the use of their corresponding n-bit fixed-point approximation, e.g. $\overline{y}$ and $\overline{\rho}_\sigma$ denote the n-bit approximation of $y \in \mathbb{R}$ and the function ρ_σ, respectively. Since we can exactly calculate $\overline{\rho}_\sigma$, we can find a partition such that the rectangles have exactly the same 'size' and represent it with the vertical bounds $\overline{y}_i$ (which we store with n bits fixed point precision) and the rounded horizontal borders $\lfloor x_i \rfloor$. The last problem is to sample uniformly at random in the infinite sets $[\overline{y}_i, \overline{y}_{i-1}]$. Our solution is to discretize the set: We define $\overline{h}_i := \overline{y}_{i-1} - \overline{y}_i$ to be the height of the i-th rectangle, sample $y' \xleftarrow{\$} \{0, 1, \ldots, 2^\omega - 1\}$ for a parameter $\omega \in \mathbb{Z}_0^+$ and transform the samples to $\overline{y} = \overline{h}_i y' \in [0, 2^\omega \overline{h}_i]$. Instead of transforming $\overline{y}$ into



(a) The concave-down case (b) The concave-up case

Fig. 2. Optimization to discrete Ziggurat ($\hat{R}_i^r$ is R_i^r vertically shifted and stretched)

the interval $[\overline{y}_i, \overline{y}_{i-1}]$ we replace the condition $\overline{y} \leq \overline{\rho}_\sigma(x)$ for $\overline{y} \in [\overline{y}_i, \overline{y}_{i-1}]$ with $\overline{y} \leq 2^\omega(\overline{\rho}_\sigma(x) - \overline{y}_i)$ for $\overline{y} \in [0, 2^\omega \overline{h}_i]$. We show in Sect. 3 how to choose the parameters t, ω and n in order to bound the statistical distance between the distribution defined by our algorithm and D_σ by a given value.

2.5 Further Improvement

Since the most time-consuming part of the discrete Ziggurat is the computation of $\overline{\rho}_\sigma$, we want to avoid it as often as possible. As mentioned above, it is only necessary if (x, y) is contained in a right rectangle R_i^r. But even in this case, depending on the shape of $\overline{\rho}_\sigma$ inside of R_i^r, we can avoid the evaluation of $\overline{\rho}_\sigma$ in nearly half of the cases and more easily reject or accept x as follows.

We divide R_i^r by connecting its upper left and lower right corner by a straight line s. Since $\overline{\rho}_\sigma$ has inflection point σ, it is concave-down for $x \leq \sigma$, and concave-up otherwise. In the concave-down case ($x_i \leq \sigma$) all points (x, y) in R_i^r below s implicitly fulfill the acceptance condition, thus x is instantly output. In the concave-up case ($\sigma \leq x_{i-1}$) all points above s lead to immediate rejection. In all other cases we have to evaluate $\overline{\rho}_\sigma(x)$ and check the acceptance condition. For the discrete Ziggurat we have to adjust this approach to our way of sampling $\overline{y}_i$ and our use of the values $\lfloor x_i \rfloor$ instead of x_i (for an idea how to accomplish this see Fig. 2).

3 Quality of Our Sampler

In this section, we show how to choose parameters for the algorithm such that it achieves a given quality in the sense of statistical distance to a discrete normal distribution. We begin with a theorem that bounds the statistical distance between the distribution produced by the sampler and a discrete normal distribution. Afterwards, we show as an example how to select parameters such that the statistical distance is smaller than 2^{-100}. The methodology can be used to select parameters for any given statistical distance.

3.1 Statistical Distance Between Sampled and Gaussian Distribution

No practical algorithm outputs samples exactly distributed according to a Gaussian distribution. Therefore, it is important to understand how much the produced output distribution differs from the normal distribution. This difference is measured by the statistical distance. Recall that t determines the tailcut and ω the precision of the sampled y-values. As explained before, we use numbers with n-bit fixed-point precision. Similar to the definition of the support $B_0^+ = [0, t\sigma] \cap \mathbb{Z}_0^+$, we define $B^+ := [0, t\sigma] \cap \mathbb{Z}^+$. The next theorem gives a lower bound on the quality of our sampler depending on the used parameters.

Theorem 1. *The statistical distance between the discrete Gaussian distribution D_σ and the distribution $\overline{D}_\sigma$ output by our algorithm is bounded by*

$$\Delta(D_\sigma, \overline{D}_\sigma) < te^{(1-t^2)/2} + \frac{\left|B_0^+\right|}{\overline{\rho}_\sigma(B^+) + \frac{1}{2}}(2^{-\omega+1} + 2^{-n}). \tag{1}$$

Because of the restricted space, we omit the proof of the result here. It can be found in the full version of this paper[2]. The main idea of the proof is to introduce intermediary distributions. The first intermediary distribution differs from a Gaussian distribution by the tailcut. The second intermediary distribution takes the limited precision of the stored numbers and the sampled y-values into consideration. After bounding the statistical distances between the consecutive distributions, we apply the triangle inequality to show the main result.

3.2 Parameter Selection

We now show how to choose t, n and ω such that the statistical distance of our distribution and the discrete Gaussian distribution is below 2^{-100} for $\sigma = 10$. We choose t to be the smallest positive integer such that $t\exp((1-t^2)/2) < 2^{-101}$, which is $t = 13$. Furthermore, we choose $\omega = n + 1$ and obtain $2^{-\omega+1} + 2^{-n} = 2^{-n+1}$. We can now find an n such that the second addend of inequality (1) is bounded by 2^{-101}. Since this calculation is a little bit complex, we try to get a feeling for the expected result first. Since t was chosen such that the probability of sampling an element in the tail is extremely small, we obtain

$$\overline{\rho}_\sigma(B^+) + \frac{1}{2} \approx \rho_\sigma(B^+) + \frac{1}{2} \approx \rho_\sigma(\mathbb{Z}_0^+) \approx \int_0^\infty \rho_\sigma(x)dx = \sigma\sqrt{\frac{\pi}{2}}$$

and expect

$$2^{-n+1}\frac{\left|B_0^+\right|}{\overline{\rho}_\sigma(B^+)} \approx 2^{-n+1}\frac{t\sigma}{\sigma\sqrt{\pi/2}} \approx 2^{-n+1}t \approx 2^{-n+5}.$$

The smallest n satisfying $5 - n \leq -101$ is $n = 106$. An exact calculation shows indeed that $n = 106$ suffices.

[2] The full version is available at http://eprint.iacr.org/2013/510.pdf.

4 Experiments and Results

In this section we discuss the performance of our implementation of the discrete Ziggurat algorithm. We first describe the experiments we performed to test the efficiency, then present their results and analyze the gathered data. Furthermore, we compare our sampler to implementations of existing samplers for discrete Gaussians.

4.1 Environment and Setup

The experiments were carried out on a Sun XFire 4400 server with 16 Quad-Core AMD Opteron 8356 CPUs running at 2.3 GHz (we only used one CPU), having in total 64GB of RAM and running a 64 bit Debian 7.1. All implementations use the Number Theory Library (NTL, cf. [Sho]) with precision $n = 106$ bits in consistency to our choice of parameters in Sect. 3.2 to assure a statistical distance for Ziggurat of at most 2^{-100}. Furthermore, we used the tailcut $t = 13$ and the discrete Gaussian parameters $\sigma \in \{10, 32, 1000, 1.6 \cdot 10^5\}$. The value $\sigma = 32$ maintains the worst-to-average-case reduction [Reg05] in several schemes for a certain parameter set, and the rather large value $\sigma = 1.6 \cdot 10^5$ is chosen according to Galbraith and Dwarakanath [GD12]. The other two values $\sigma = 10, 1000$ were chosen arbitrarily inbetween and at the lower end to allow a better comparison.

We queried each algorithm iteratively 1 million times to output a single sample per call. These experiments were applied to the discrete Ziggurat with the

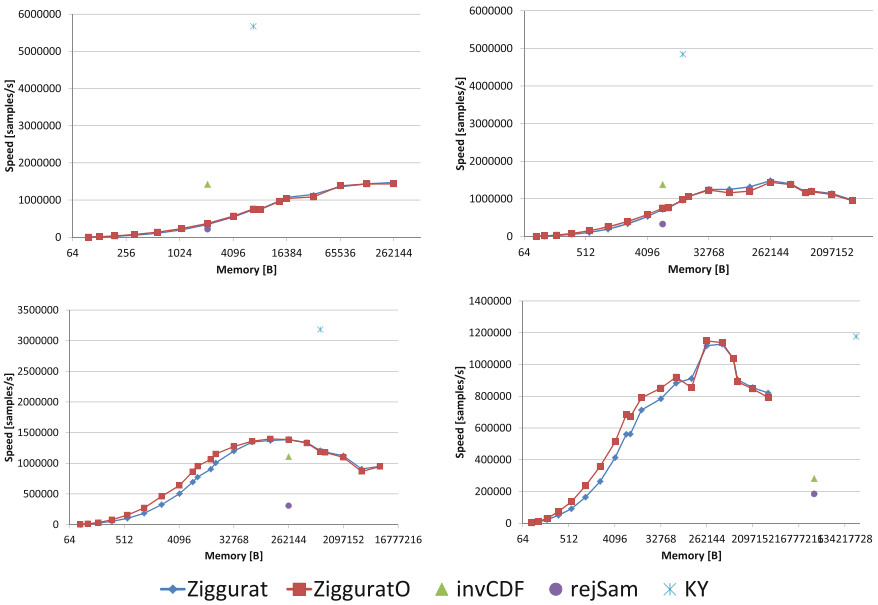


Fig. 3. Results for inverse CDF, rejection sampling, Knuth-Yao, and discrete Ziggurat with and without optimization for parameters $\sigma = 10, 32, 1000, 1.6 \cdot 10^5$, respectively.

optimization using the straight line s (ZigguratO), discrete Ziggurat without optimization (Ziggurat), inverse CDF (invCDF), rejection sampling with precomputed lookup-table (rejSam), and Knuth-Yao (KY). Furthermore we tested both Ziggurat algorithms with a precomputed lookup-table for the support B_0^+ (ZigguratOP and ZigguratP, respectively).

For each algorithm we measured the running time using the (Linux-internal) function `clock_gettime` with clock `CLOCK_PROCESS_CPUTIME_ID`. In order to have non-distorted results we excluded all pre- and post-computations (e.g. setting up lookup-tables) from the measurements. Regarding the memory, we did not perform per-runtime analyses, but computed the amount of memory by adding up the number of fixed variables in regard to their types in NTL. For our choice of parameters, in Ziggurat(O) the values on the x-axis need 8 bytes and on the y-axis 24 bytes of memory. With m rectangles the total amount of memory is thus $32(m+2)$ bytes (including σ, t, ω, m). For both invCDF and rejSam we need to store a lookup-table of $t\sigma$ values à 16 bytes, resulting in 2080 bytes for $\sigma = 10$. The same amount of memory is used by Ziggurat(O) with $m = 63$ rectangles. The size of Knuth-Yao is approximated by $(\#\text{intermediates} + \#\text{leaves})/2$ bits, where $\#\text{intermediates} = n \cdot 2^{\lceil \log \log(n \cdot t\sigma) \rceil}$ and $\#\text{leaves} = n \cdot 2^{\lceil \log \log(t\sigma) \rceil}$ for precision $n = 106$ bits as above.

4.2 Results

Figure 3 shows results for inverse CDF, rejection sampling, Knuth-Yao, and discrete Ziggurat with and without optimizations for different numbers of rectangles. It shows four different graphs for different values of σ. For small values of σ, the inverse CDF method outperforms both discrete Ziggurat and rejection sampling for the same fixed amount of memory. For example, our implementation invCDF samples about 1.37 million samples per second for $\sigma = 32$. On the other hand, rejection sampling is quite slow due to a large rejection area. Even with a precomputed lookup-table, rejSam only achieves about 327,000 samples per second, which is a factor 4.2 slower than invCDF. The naïve approach without lookup-table solely achieves 2,500 samples per second, being a factor 558 slower than invCDF. For the same amount of memory, ZigguratO achieves an overall number of about 753,000 samples per second, while Ziggurat outputs 718,000 samples per second. Compared to the other two methods, Ziggurat is 1.91 times slower than invCDF and a factor 2.19 faster than rejSam. Our implementation of Knuth-Yao outperforms all the other methods by at least a factor of 3.53, outputting 4.85 million samples per second. This comes at the cost of nearly doubled memory usage.

In the extreme case $\sigma = 1.6 \cdot 10^5$, the fastest instantiation of Ziggurat outputs 1.13 million samples per second with a memory usage of 524 KB. Inverse CDF creates 281,000 samples per second while using 33.55 MB, thus being about a factor 4 slower than Ziggurat. For rejSam the situation is even worse: Using the same amount of memory as invCDF, it only outputs 185,000 samples per second – a factor 6.1 slower than Ziggurat. The Knuth-Yao algorithm still performs

better than Ziggurat, but only by 4.26 %. On the other hand, KY needs more than 424 times the memory storage of Ziggurat. Concluding we state that for larger values of σ the Ziggurat algorithm beats both inverse CDF and rejection sampling. Compared to Knuth-Yao, Ziggurat achieves almost the same speed but reduces the memory consumption by a factor of more than 400.

Figure 3 shows that we can beat invCDF in terms of speed and compete with Knuth-Yao. The reason for this is the simplicity of the algorithm. If many rectangles are stored, the rejection-probability gets very small. Likewise, the probability to sample an x in a right rectangle R_i^r gets very small. Therefore, the algorithm only samples a rectangle and afterwards samples a value within this rectangle, which can be done very fast.

As one can furthermore see in Fig. 3, the discrete Ziggurat algorithm shows a large flexibility in regard to the speed-memory trade-off. For a small amount of memory (i.e. number of rectangles) it is quite slow, e.g. for $\sigma = 32$ and 8 rectangles it obtains about 57,000 samples per second. For increasing memory allocation the speed of Ziggurat(O) increases. This statement holds for all values of σ we tested. As can be seen by the graphs, the speed of Ziggurat decreases for increasing number of rectangles. This was first surprising to us. Further analysis showed that this is due to the fact that with increasing number of rectangles (i.e. amount of allocated memory) the processor cannot keep the partition table in the fast caches, but has to obtain requested memory addresses from slower caches on demand. In addition, the large number of rectangles requires more bits to be sampled in a single step of the algorithm.

The trade-off provided by the Ziggurat-algorithms is indeed a property the other approaches do not share. InvCDF assigns every possible value to an interval on the y-axis. Consequently, one has to store at least the borders if the intervals. Decreasing the precision of the borders will decrease the memory consumption, but as well decrease the quality of the sampler. Increasing the precision

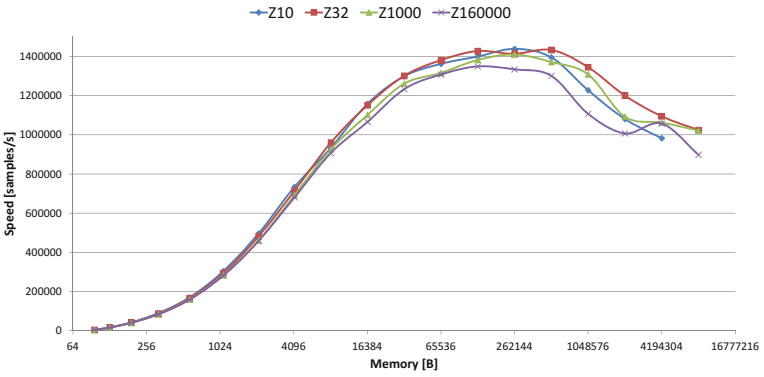
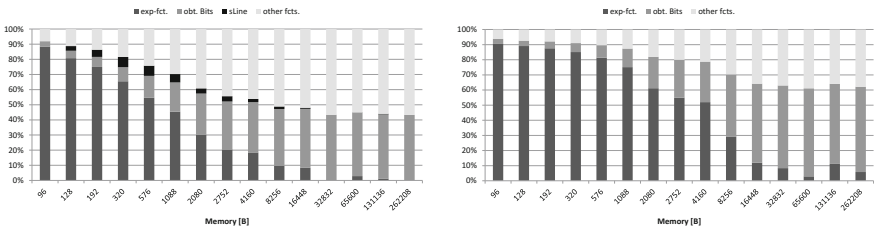


Fig. 4. Discrete Ziggurat for different parameters σ (ZX denotes Ziggurat for $\sigma = X$).



(a) ZigguratO (b) Ziggurat

Fig. 5. Time-split of discrete Ziggurat with and without optimization

or storing intermediate values, on the other hand, will not decrease the running time. The same happens to rejection sampling if the precision of the precomputed values is changed. Knuth-Yao stores for every element in the support the probability to sample this element. Decreasing the precision of the stored probabilities would (like for invCDF) decrease the quality of the sampler. While there might be efficient ways to store those values, there is a minimal amount of space required to store this information. Knuth-Yao as well as invCDF and rejection sampling therefore only provide a trade-off between quality and speed/memory consumption.

In Fig. 4 we draw the time-memory trade-off for the Ziggurat algorithm for different values of σ. One can see that the performance of the Ziggurat algorithm decreases for larger σ. What is interesting in the graph is that the Ziggurat algorithm for $\sigma = 10$ is slower for a large amount of rectangles than for $\sigma = 32$. This is puzzling as we cannot directly explain the described behaviour. We want to state that during our experiments we saw quite large fluctuations for several runs of the algorithm. Maybe this could explain the better performance for $\sigma = 32$ in comparison to $\sigma = 10$.

We also compared ZigguratO and Ziggurat in regard to speed.[3] The improvement rate increases up to 30 % for a total memory of 320 bytes, then decreases to around 6 % for 2080 bytes, until finally for 130KB and bigger there is no improvement. Overall, the described behaviour is not surprising since for increased memory the number of rectangles gets larger, so that the rejection area is very small. This leads to nearly no evaluations in the right sub-rectangles R_i^r and therefore to no computation of the straight line s (or even $\overline{\rho}_\sigma$).

Additionally, we compared ZigguratO to ZigguratOP, which operates with at least 2176 bytes of memory. ZigguratOP is slower until about 2.5 KB of memory, but then it beats ZigguratO with a speedup of up to 40 %, until for memory larger than 262 KB there seems to be no speedup at all. This behaviour is reasonable since the lookup-table requires more storage, but simultaneously affects the speed due to replacing $\overline{\rho}_\sigma$ by table-lookups.

At last, we give insights on the time-split for our implementations Ziggurat and ZigguratO. We used the tool suite `Valgrind` with the tool `Callgrind`

[3] For additional Figures see Appendix A.2.

to obtain the measurements and analyzed them using the Callee Graph in the `Callgrind`-GUI `KCachegrind`. Figure 5 shows the percentages for both algorithms. We chose the most interesting sub-routines, i.e. the exponential function (called inside $\bar{\rho}_\sigma$ in R_i^r), the generation of random bits, the computation of the straight line s (in ZigguratO), and 'other' sub-routines. One can see that for a small amount of memory the computation of the exponential function takes most part of the running time, e.g. for 104 bytes (two rectangles) its computation consumes 80–90 % of the total running time. As the memory increases, the rejection area gets smaller, i.e. the percentage of the right sub-rectangles R_i^r compared to their super-rectangles R_i. Thus, the number of integers sampled inside the R_i^r's decreases. Additionally, the exponential function has to be called less often. Nevertheless, the graphs show that the use of the straight line s decreases the use of the exponential function (or call to $\bar{\rho}_\sigma$) in ZigguratO in comparison to Ziggurat considerably, while at the same time the computational complexity of s is not high (at most 6.77 %).

A Appendix

In this Appendix we present the pseudocode for the discrete Ziggurat algorithm and give additional Figures in regard to our experimental results.

A.1 Pseudocode for Discrete Ziggurat

In Fig. 6 we present the pseudocode for our implementation of the discrete Ziggurat algorithm of Sect. 2. In particular, we give the pseudocode for ZigguratO. From this, one obtains pseudocode for Ziggurat by removing lines 11–17, 19 and 20.

A.2 Additional Figures Regarding Results

In Fig. 7 we present the rate of improvement of Ziggurat with optimization (ZigguratO) over Ziggurat without the straight line approach. For a small amount of memory, the improvement using the straight line approach is quite good (around 20–30 % for memory usage between 128 and 576 bytes), while for larger memory, i.e. higher number of rectangles, the improvement vanishes due to nearly no rejection area.

Figure 8 shows the speed of ZigguratO and its corresponding variant ZigguratOP with precomputed lookup-table. ZigguratOP can perform only with memory larger or equal to 2176 bytes due to the size of the lookup-table. Thus, given a small amount of memory, it is not possible to apply ZigguratOP. But for available memory larger than 2.5 KB ZigguratOP outperforms ZigguratO up to 40 %.

Algorithm 1: ZigguratO

Input: m, σ, $\lfloor x_1 \rfloor, \ldots, \lfloor x_m \rfloor$, $\overline{y}_0, \overline{y}_1, \ldots, \overline{y}_m$, ω
Output: number distributed according to a discrete Gaussian distribution

while *true* **do**
  $i \xleftarrow{\$} \{1, \ldots, m\}$, $s \xleftarrow{\$} \{-1, 1\}$, $x \xleftarrow{\$} \{0, \ldots, \lfloor x_i \rfloor\}$;
  `// choose rectangle, sign and value`
  if $0 < x \leq \lfloor x_{i-1} \rfloor$ **then return** sx;
  else
    if $x = 0$ **then**
      $b \xleftarrow{\$} \{0, 1\}$;
      if $b = 0$ **then return** sx;
      else continue;
    else
      `// in rejection area` R_i^r `now`
      $y' \xleftarrow{\$} \{0, \ldots, 2^\omega - 1\}$, $\overline{y} = y' \cdot (\overline{y}_{i-1} - \overline{y}_i)$;
      if $\lfloor x_i \rfloor + 1 \leq \sigma$ **then**
        `// in concave-down case`
        if
        $\overline{y} \leq 2^\omega \cdot sLine(\lfloor x_{i-1} \rfloor, \lfloor x_i \rfloor, \overline{y}_{i-1}, \overline{y}_i; x) \vee \overline{y} \leq 2^\omega \cdot (\overline{\rho}_\sigma(x) - \overline{y}_i)$
        then return sx;
        else continue;
      else if $\sigma \leq \lfloor x_{i-1} \rfloor$ **then**
        `// in concave-up case`
        if
        $\overline{y} \geq 2^\omega \cdot sLine(\lfloor x_{i-1} \rfloor, \lfloor x_i \rfloor, \overline{y}_{i-1}, \overline{y}_i; x-1) \vee \overline{y} > 2^\omega \cdot (\overline{\rho}_\sigma(x) - \overline{y}_i)$
        then continue;
        else return sx;
      else
        if $\overline{y} \leq 2^\omega \cdot (\overline{\rho}_\sigma(x) - \overline{y}_i)$ **then return** sx;
        else continue;
      end
    end
  end
end

Algorithm 2: sLine$(\lfloor x_{i-1} \rfloor, \lfloor x_i \rfloor, \overline{y}_{i-1}, \overline{y}_i; x)$

if $\lfloor x_i \rfloor = \lfloor x_{i-1} \rfloor$ **then return** -1;
Set $\hat{y}_i = \overline{y}_i$ and $\hat{y}_{i-1} = \begin{cases} \overline{y}_{i-1} & i > 1 \\ 1 & i = 1 \end{cases}$
return $\dfrac{\hat{y}_i - \hat{y}_{i-1}}{\lfloor x_i \rfloor - \lfloor x_{i-1} \rfloor} \cdot (x - \lfloor x_i \rfloor)$

Fig. 6. The discrete Ziggurat algorithm with optimization (ZigguratO)

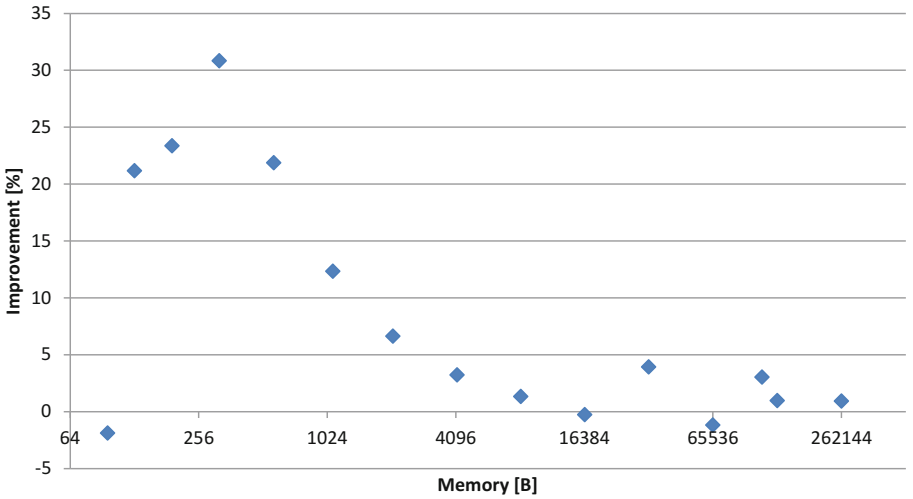


Fig. 7. Improvement rate of ZigguratO over Ziggurat

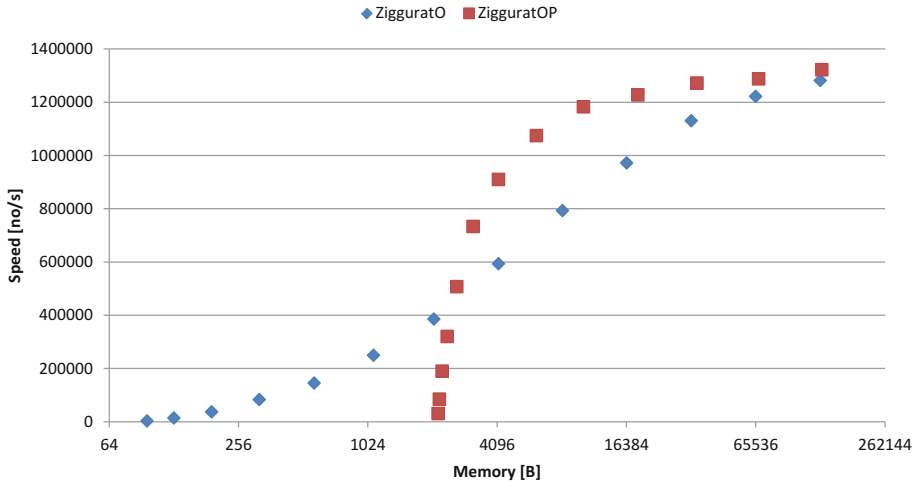


Fig. 8. Comparison of ZigguratO and ZigguratOP

References

[AGHS12] Agrawal, S., Gentry, C., Halevi, S., Sahai, A:. Discrete Gaussian leftover hash lemma over infinite domains. Cryptology ePrint Archive, Report 2012/714 (2012). http://eprint.iacr.org/

[BGV12] Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS '12, pp. 309–325. ACM, New York (2012)

[DN12] Ducas, L., Nguyen, P.Q.: Faster Gaussian lattice sampling using lazy floating-point arithmetic. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 415–432. Springer, Heidelberg (2012)

[GD12] Galbraith, S.D., Dwarakanath, N.C.: Efficient sampling from discrete Gaussians for lattice-based cryptography on a constrained device. Preprint (2012)

[GGH12] Garg, S., Gentry, C., Halevi, S.: Candidate multilinear maps from ideal lattices. Cryptology ePrint Archive, Report 2012/610 (2012). http://eprint.iacr.org/

[GPV08] Gentry, C., Peikert, C., Vaikuntanathan, C.: Trapdoors for hard lattices and new cryptographic constructions. In: Ladner, R.E., Dwork, C. (ed.) 40th ACM STOC Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, 17–20 May 2008, pp. 197–206. ACM Press (2008)

[LP11] Lindner, R., Peikert, Ch.: Better key sizes (and attacks) for LWE-based encryption. In: Kiayias, A. (ed.) CT-RSA 2011. LNCS, vol. 6558, pp. 319–339. Springer, Heidelberg (2011)

[Lyu12] Lyubashevsky, V.: Lattice signatures without trapdoors. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 738–755. Springer, Heidelberg (2012)

[MT00] Marsaglia, G., Tsang, W.W.: The Ziggurat method for generating random variables. J. Stat. Softw. **5**(8), 1–7, 10 (2000)

[Pei10] Peikert, Ch.: An efficient and parallel Gaussian sampler for lattices. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 80–97. Springer, Heidelberg (2010)

[Reg05] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing, STOC '05, pp. 84–93. ACM, New York (2005)

[Sho] Shoup, V.: Number Theory Library (NTL) for C++. http://www.shoup.net/ntl/

[TLLV07] Thomas, D.B., Luk, W., Leong, P.H.W.: Gaussian random number generators. ACM Comput. Surv. **39**(4), 11:1–11:38 (2007)

[WHCB13] Weiden, P., Hülsing, A., Cabarcas, D., Buchmann, J.: Instantiating treeless signature schemes. Cryptology ePrint Archive, Report 2013/065 (2013). http://eprint.iacr.org/