A report on gaussian number generation methods

Likhitha, Jahnavi, Tanishq, Riddhiman

May 2026

Introduction

Gaussian random numbers are core to simulation workflows in communications, finance, and many other domains. Section 2 of Thomas et al. (2007) surveys transform-based and approximation-based generators. This report focuses on three representative methods that trade off speed and tail accuracy in different ways: the CLT sum, a piecewise linear triangles approximation, and the Monty Python packing method.

We also look at the Polar Rejection Method is an efficient technique for generating independent standard normal random variables. It is an improved version of the Box–Muller method and avoids the use of trigonometric functions.

We then present a method to utilize the advantages of floating point representation numbers in computers for uniform number generation.

This is followed by some methods to generate samples from gaussian tails.

Preliminaries

Let $\phi(x)$ and $\Phi(x)$ denote the standard normal PDF and CDF:

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}, \qquad \Phi(x) = \int_{-\infty}^{x} \phi(t)\, dt.$$

We assume access to independent uniform samples $U \sim \mathcal{U}(0,1)$.

Methods

Central Limit Theorem (CLT) Sum

Generate $U_i \sim \mathcal{U}(0,1)$ for $i = 1, \ldots, K$ and sum $S = \sum U_i$. Standardize to unit variance:

$$Z = \frac{S - K/2}{\sqrt{K/12}}.$$

For $K = 12$, the scale factor is 1, so $Z = S - 6$. The output is exact for the Irwin–Hall distribution and only approximately Gaussian for finite K, with noticeably light tails unless K is large.

Piecewise Linear (Triangles) Approximation

Approximate $\phi(x)$ on $[-x_{\max}, x_{\max}]$ by a mixture of k symmetric triangular components:

$$\phi(x) \approx g(x) = \sum_{i=1}^{k} q_i\, t_i(x), \qquad \sum_{i=1}^{k} q_i = 1.$$

Each triangle has width $2w$ and center c_i with density

$$t_i(x) = \begin{cases} \dfrac{1}{w^2}(w - |x - c_i|), & |x - c_i| \le w, \\ 0, & \text{otherwise.} \end{cases}$$

Centers are equally spaced, $c_i = w\left(\frac{k+1}{2} - i\right)$, and weights are chosen to match Gaussian mass at the centers:

$$q_i = \frac{\phi(c_i)}{\sum_{j=1}^{k} \phi(c_j)}.$$

This produces a (piecewise linear) density because selecting triangle i with probability q_i and then sampling within that triangle yields the mixture density $g(x) = \sum_i q_i t_i(x)$. With symmetric placement and weights chosen to match ϕ at the centers, $g(x)$ approximates the bell curve; increasing k and $x_{\max}$ improves the match and pushes the approximation closer to a normal distribution. Accuracy improves with larger k and $x_{\max}$, but requires more uniforms and a larger lookup table for sampling the mixture.

Monty Python Packing

The Monty Python method folds the PDF into a rectangle of width b and height $1/(2b)$. Sample (x, y) uniformly from $0 \le x \le b$, $0 \le y \le 1/(2b)$. Let a solve $\phi(a) = 1/(2b)$ so that $x < a$ is always accepted. For $x \ge a$, accept if $y < \phi(x)$. Samples above the curve are mapped into the tail region $|x| > b$ using an area-preserving transform plus a dedicated tail sampler. A random sign restores symmetry.

Experimental Setup

We generated $n = 10^5$ samples for each method using:

$$K = 12, \quad k = 9, \quad x_{\max} = 3.0, \quad b = 2.29.$$

Figures were generated using the accompanying R script `section2-grng-report.R`.

Results

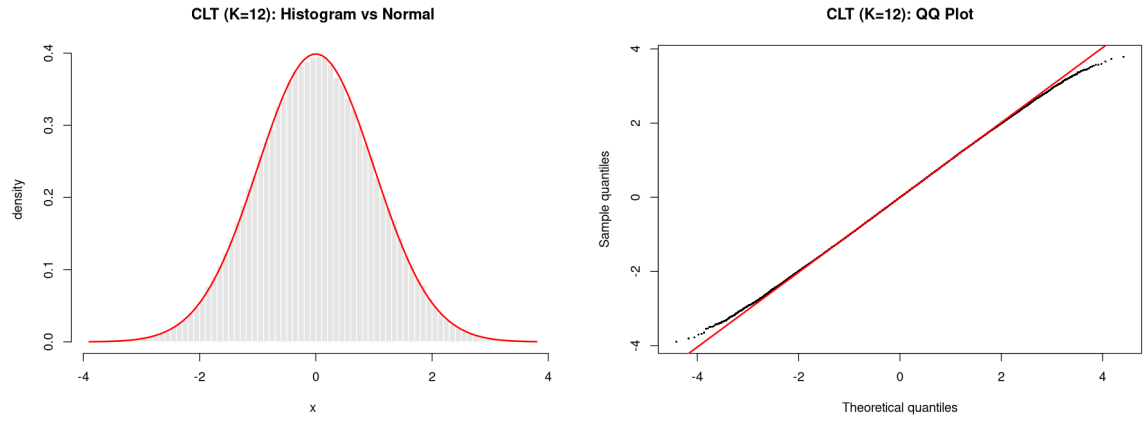


Figure 1: CLT method: histogram (left) and QQ plot (right). Tails are light for finite K.

Complexity Comparison

Method	Avg cost	Notes
CLT sum (K)	K uniforms + $K-1$ adds	Approximate; tails are thin unless K is large.
Triangles (piecewise)	2–3 uniforms + table + mult/add	Approximate; quality depends on k and $x_{\max}$.
Monty Python	2–3 uniforms fast path; tail calls	Exact if tail sampler is exact; efficiency set by b.

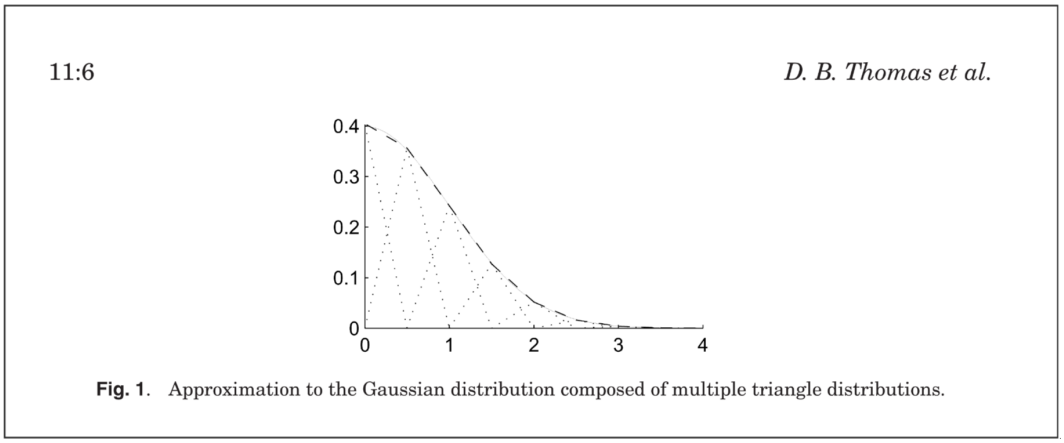


Figure 2: Piecewise linear approximation using overlapping triangles (Thomas et al., 2007).

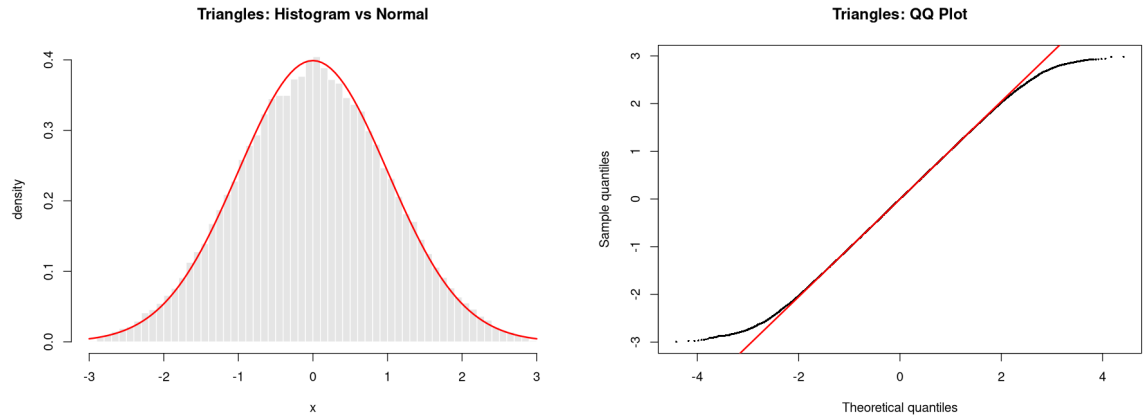


Figure 3: Triangles method: histogram (left) and QQ plot (right). Minor kinks reflect the piecewise approximation.

Now we will discuss two important rejection-based methods for the generation of Gaussian random numbers: the Polar Rejection Method and the Ratio of Uniforms Method. We explain how the algorithms work, discuss their advantages and disadvantages, and verify whether the generated samples follow the Gaussian distribution using graphical methods. Finally, we compare the efficiency of both methods.

Polar Rejection Method

Theory

Let

$$U_1, U_2 \sim \text{Uniform}(-1, 1)$$

and define

$$S = U_1^2 + U_2^2.$$

If

$$S \geq 1,$$

then the point lies outside the unit circle and is rejected.
If

$$S < 1,$$

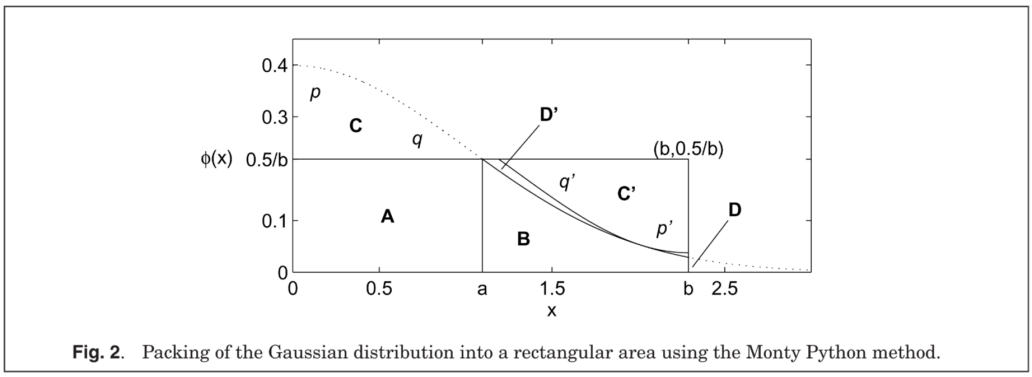


Fig. 2. Packing of the Gaussian distribution into a rectangular area using the Monty Python method.

Figure 4: Monty Python packing of the Gaussian PDF into a rectangle (Thomas et al., 2007).

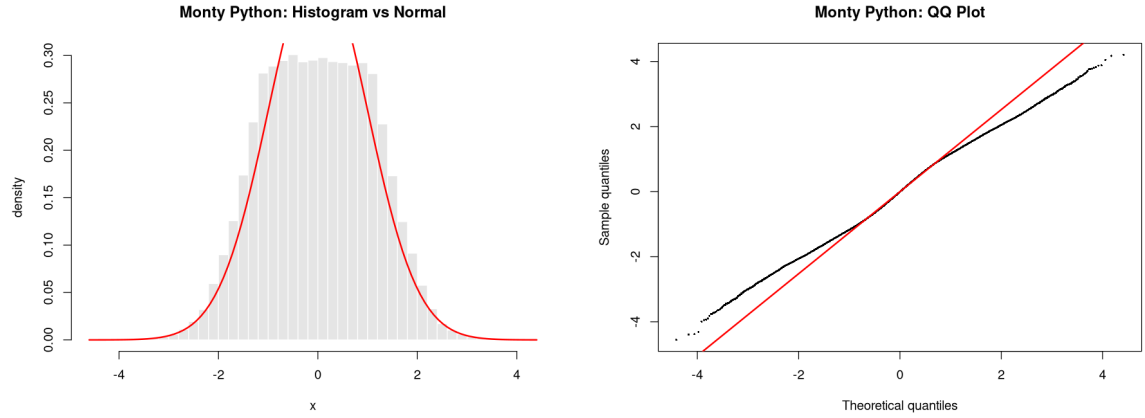


Figure 5: Monty Python method: histogram (left) and QQ plot (right). The fast path matches the Gaussian well; tail quality depends on the tail sampler.

then define

$$X = U_1 \sqrt{\frac{-2 \ln S}{S}},$$

$$Y = U_2 \sqrt{\frac{-2 \ln S}{S}}.$$

Then X and Y are independent standard normal random variables.

Algorithm

1. Generate $U_1, U_2 \in (-1, 1)$.
2. Compute

$$S = U_1^2 + U_2^2.$$

3. If $S \geq 1$, reject and repeat.
4. Otherwise compute:

$$X = U_1 \sqrt{\frac{-2 \ln S}{S}},$$

$$Y = U_2 \sqrt{\frac{-2 \ln S}{S}}.$$

5. Output X and Y.

How Does This Scaling Work and Why is it Better than Box–Muller?

Instead of using trigonometric functions, the Polar Rejection Method samples a point (u_1, u_2) uniformly from the unit square and accepts it if it lies inside the unit disk:

$$u_1, u_2 \sim \text{Uniform}(-1, 1),$$

$$S = u_1^2 + u_2^2 < 1.$$

The quantity S lies in $(0, 1)$ after rejection. Using the Box–Muller transformation:

$$Z_1 = u_1 \sqrt{\frac{-2 \ln S}{S}},$$

$$Z_2 = u_2 \sqrt{\frac{-2 \ln S}{S}}.$$

This method avoids costly trigonometric computations, making it computationally more efficient than the original Box–Muller transformation.

Before and After Separation Graphs

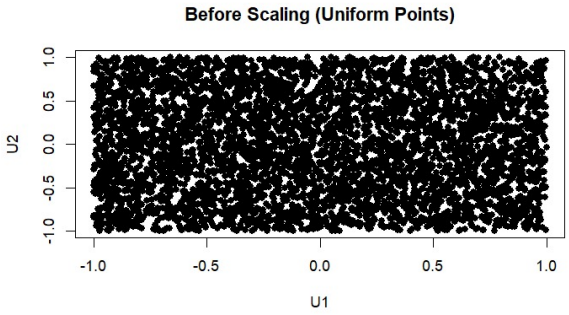


Figure 6: Before Separation Graph

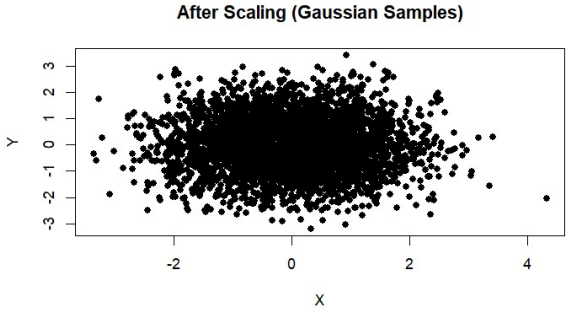


Figure 7: After Separation Graph

Advantages

1. Efficient generation of normal random variables.
2. Avoids trigonometric computations.

3. Generates two random variables simultaneously.
4. Computationally faster than the Box–Muller transform.

Disadvantages

1. Some generated points are rejected.
2. Efficiency depends on the acceptance probability.

Ratio of Uniforms Method

Introduction

The Ratio of Uniforms Method is a flexible rejection-based sampling technique used to generate random variables from complicated probability distributions.

Theory

Suppose the target density is $f(x)$. Generate random points (U, V) uniformly from a suitable bounded region.

Define:

$$X = \frac{U}{V}.$$

The acceptance condition is:

$$0 < V \leq \sqrt{f\left(\frac{U}{V}\right)}.$$

Accepted points generate random samples from the target distribution.

Algorithm

1. Generate (U, V) uniformly over a bounded region.
2. Check whether:

$$0 < V \leq \sqrt{f\left(\frac{U}{V}\right)}.$$

3. If accepted, compute:

$$X = \frac{U}{V}.$$

4. Otherwise reject and repeat.

How Does This Algorithm Work?

Define the transformation:

$$X = \frac{V}{U}, \quad U = U,$$

so that

$$V = UX.$$

The Jacobian of the transformation $(u, x) \mapsto (u, v)$ is:

$$J = \left|\frac{\partial(u,v)}{\partial(u,x)}\right| = \begin{vmatrix} 1 & 0 \\ x & u \end{vmatrix} = u.$$

Since (U,V) is uniformly distributed over A, its joint density is:

$$f_{U,V}(u,v) = \begin{cases} C, & (u,v) \in A, \\ 0, & \text{otherwise}, \end{cases}$$

where C is a normalizing constant.
Using change of variables:

$$f_{X,U}(x,u) = f_{U,V}(u,ux) \cdot |J| = C \cdot u,$$

for

$$0 < u \leq \sqrt{f(x)}.$$

The marginal density of X is:

$$\begin{aligned} f_X(x) &= \int_0^{\sqrt{f(x)}} C \cdot u \, du \\ &= C \cdot \frac{f(x)}{2}. \end{aligned}$$

After normalization, we obtain:

$$f_X(x) = f(x).$$

■

Application to Gaussian Distribution

Consider the standard normal density:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

Then the region A becomes:

$$u \leq \sqrt{f(x)} = C \cdot e^{-x^2/4},$$

where C is a constant.
Substituting

$$x = \frac{v}{u},$$

we get:

$$u \leq C \cdot e^{-(v/u)^2/4}.$$

Squaring both sides:

$$u^2 \leq C^2 \cdot e^{-v^2/(2u^2)}.$$

Taking logarithms:

$$\ln(u^2) \leq \ln(C^2) - \frac{v^2}{2u^2}.$$

Rearranging:

$$v^2 \leq 2u^2 \left(\ln(C^2) - \ln(u^2)\right).$$

Ignoring constants absorbed by scaling, we obtain the fundamental condition:

$$v^2 \leq -4u^2 \ln u.$$

Boundary Condition and Rejection Sampling

The inequality

$$v^2 \leq -4u^2 \ln u$$

defines the boundary of the region A corresponding to the Gaussian density.

In practice, it is difficult to sample directly from this curved region. Therefore, the algorithm proceeds as follows:

- Sample (u, v) uniformly from a simple rectangular region.
- Accept the sample if it satisfies:

$$v^2 < -4u^2 \ln u.$$

- Otherwise, reject and resample.

Why Boundary Checking is Necessary

The proof of the Ratio of Uniforms Method assumes that (U, V) is uniformly distributed over the region A. However, direct sampling from A is not feasible due to its curved boundary.

Rejection sampling ensures correctness by:

- Generating points from an enclosing rectangle,
- Keeping only those points that lie inside A,
- Ensuring the accepted points are uniformly distributed over A.

Thus, the boundary condition

$$v^2 < -4u^2 \ln u$$

is precisely the mathematical constraint that guarantees the generated variable

$$X = \frac{V}{U}$$

follows the Gaussian distribution.

Advantages and Disadvantages of Ratio of Uniforms Method

Advantages

1. Applicable to many probability distributions.
2. Useful when the inverse transform method fails.
3. Flexible and mathematically elegant.

Disadvantages

1. Choosing the bounded region can be difficult.
2. May involve higher rejection rates.
3. Computational complexity may increase.

QQ Plots

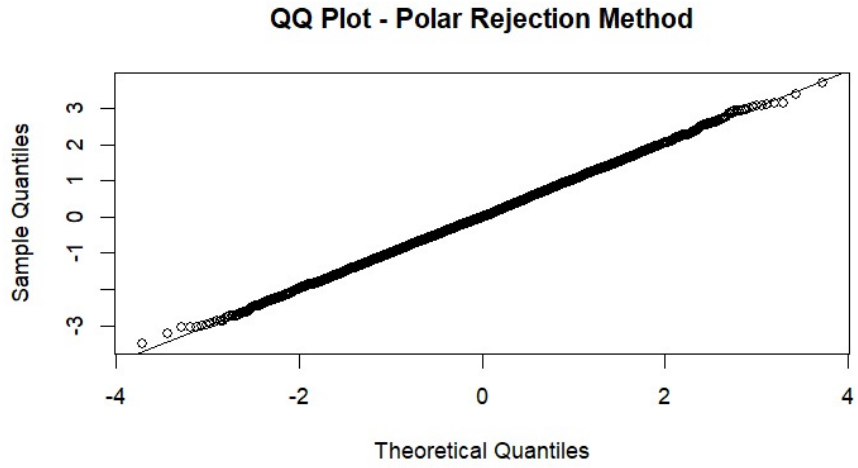


Figure 8: QQ plot for polar method

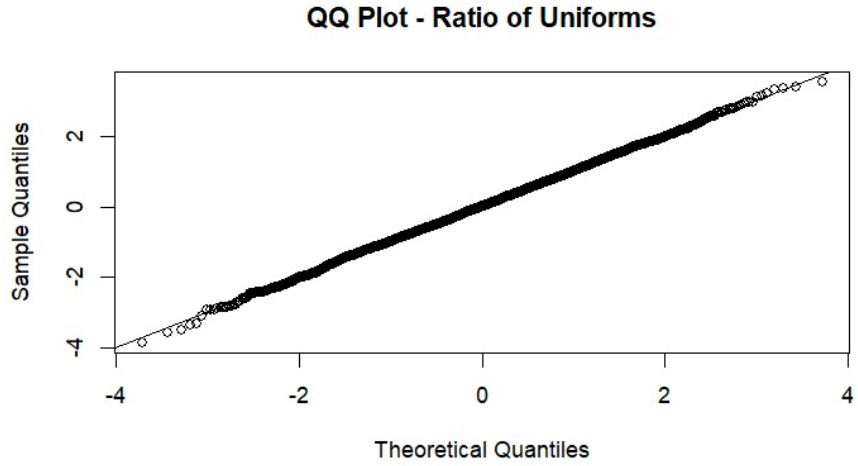
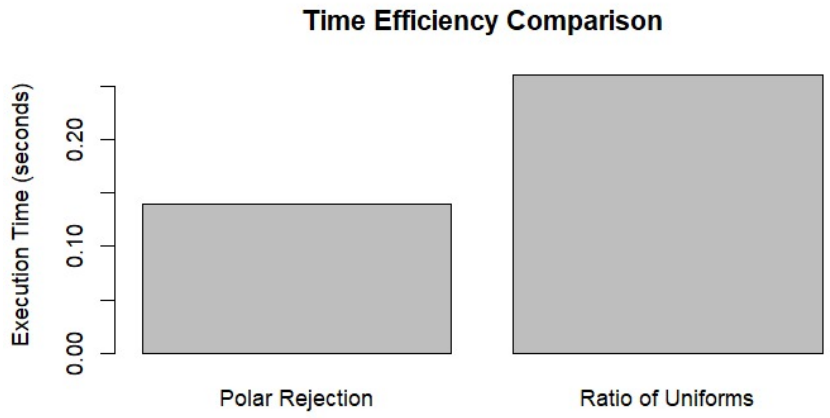


Figure 9: QQ plot for ratio of uniforms method

Comparison Table

Feature	Polar Rejection Method	Ratio of Uniforms Method
Main Idea	Uses rejection inside unit circle	Uses ratio of two uniform variables
Efficiency	Computationally efficient	Depends on chosen bounded region
Trigonometric Functions	Not required	Not required
Complexity	Simple implementation	More mathematically involved
Rejection Rate	Moderate	Can be high
Flexibility	Mainly for Gaussian sampling	Applicable to many distributions
Generated Variables	Generates two variables at once	Generates one variable at a time

Table 1: Comparison of Polar Rejection and Ratio of Uniforms Methods



How do computers store numbers with fractional parts?

Integer arithmetic is natural for digital computers because a number can be stored directly as a finite sequence of bits. Fractional numbers are more subtle. Since computers use binary, any practical encoding of real numbers must approximate many values. Two standard approaches are fixed-point representation and floating-point representation. These methods differ in how the fractional part is stored, and the difference has important consequences for numerical accuracy and random number generation.

Fixed-Point Representation

In fixed-point representation, a fixed number of bits is reserved for the fractional part of the number. This makes the spacing between representable values constant.

For example, the number 4.75 can be written exactly in binary as

$$4.75 = 0100.1100_2.$$

Here the binary point is placed at a fixed location, so the same format is used for every stored number.

A limitation of fixed-point arithmetic is that not every real number can be represented exactly. For instance, the decimal number 2.8 has no finite binary expansion, so it must be approximated. One possible approximation is

$$2.8 \approx 2.8125 = 0010.1101_2.$$

Thus, fixed-point arithmetic offers simplicity, but at the cost of limited range and limited flexibility.

Floating-Point Representation

Floating-point representation is designed to store numbers over a much wider range. It is analogous to scientific notation: a number is written as a mantissa times a power of two.

A normalized binary floating-point number has the form

$$x = \pm(1.b_1 b_2 b_3 \cdots)_2 \times 2^e,$$

where the mantissa begins with a leading 1, and e is an integer exponent.

Note that the leading 1 does not need to be stored explicitly. It is implicit in normalized representations.

IEEE 754 Standard

The most widely used floating-point standard is IEEE 754. In single-precision format, a number is stored using 32 bits; in double-precision format, it uses 64 bits.

In single precision, the 32 bits are divided into three parts:

- 1 sign bit,
- 8 exponent bits,
- 23 fraction bits for the mantissa.

For normalized single-precision numbers, the stored exponent is biased by 127. If the exponent field is E, then the actual exponent is

$$e = E - 127.$$

The all-zero and all-one exponent patterns are reserved for special cases such as zero, subnormal numbers, infinity, and NaN.

Thus a normalized single-precision floating-point number has the value

$$x = (-1)^s \times (1.f)_2 \times 2^{E-127},$$

where s is the sign bit and f is the binary fraction encoded by the mantissa bits.

As an illustration, a bit pattern of the form

$$\underbrace{0}_{\text{+ve}}\ \underbrace{01011101}_{\text{denotes } -34}\ \underbrace{00100101100010011111000}_{\text{denotes } 1.146636}$$

represents the number $+1.146636 \times 2^{-34}$.

Resolution of Floating-Point and Fixed-Point Numbers

A key difference between the two representations is how the spacing between adjacent numbers behaves.

Fixed-Point Resolution

In fixed-point arithmetic, representable numbers are evenly spaced. The resolution is constant throughout the number line. This is easy to visualize and easy to analyze, but it means that the same absolute precision is used for both small and large values.

Floating-Point Resolution

In floating-point arithmetic, the spacing is not constant. Numbers near zero are densely packed, while numbers farther from zero are more widely spaced.

This means that the resolution is dynamic: small intervals contain many representable values, whereas large intervals contain fewer.

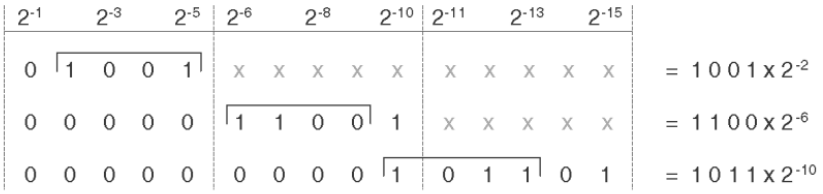


Figure 10: A toy example of the lazy generation algorithm

The Problem with Gaussian Random Number Generators

Many random number generation procedures require input from a uniform distribution on $[0, 1)$. However, most low-level random number generators produce integer values. A naive way to convert a w-bit integer into a floating-point number is to scale it by 2^{-w}.

If U is a w-bit integer, one may form

$$X = U\,2^{-w}.$$

Although this produces numbers in $[0, 1)$, it does not generate all floating-point values in that interval with the correct distribution. The resulting values inherit the coarse grid near zero from the integer source and also inherit the nonuniform spacing of floating-point numbers near one, which is the worst of two worlds.

For a truly well-behaved generator of $U(0, 1)$ values, every representable floating-point number should be generated with the appropriate probability. The naive scaling method does not accomplish this.

Uniform Random Generation in Floating-Point Form

A better approach is to generate the mantissa and exponent directly from random bits. The idea is to keep generating bits until the first 1 appears. The position of that first 1 determines the exponent, and the remaining bits fill the mantissa.

Lazy Generation Algorithm

Suppose the mantissa has m bits. The algorithm proceeds as follows:

1. Generate random bits until the first 1 appears.
2. Let the number of trials it took to get the first 1 be k.
3. Use the next $m - 1$ random bits to complete the mantissa.
4. Raise the negative of k to the exponent of 2.

The first 1 determines the magnitude of the number, and the remaining bits determine its position inside the corresponding dyadic interval.

The generator is indeed uniform

The probability that the first 1 appears after exactly k leading zeros is $2^{-(k+1)}$. This matches the length of the interval

$$[2^{-(k+1)},\, 2^{-k}),$$

since that interval also has width $2^{-(k+1)}$.

Within that interval, the remaining mantissa bits are uniformly random, so each representable floating-point number receives the correct amount of probability mass. In this sense, the algorithm produces a uniform distribution over $[0, 1)$ in floating-point form.

Discussion

The lazy generation method is elegant because it respects the structure of floating-point arithmetic. Rather than first generating an integer and then converting it, the algorithm directly constructs a floating-point number in a way that matches the geometry of the binary representation.

This avoids the mismatch between:

- the uniform spacing of integers, and
- the nonuniform spacing of floating-point numbers.

As a result, the output is more suitable for numerical methods that require high-quality uniform random inputs, including Gaussian random number generators.

Gaussian Tail Sampling

Gaussian tail sampling refers to generating random values specifically from the extreme tails of a Gaussian distribution instead of sampling from the entire distribution.

For a standard normal random variable, $X \sim N(0,1)$, most values occur near the center around 0, while very large positive or negative values occur with extremely small probability.

The tail regions are generally represented as $X > a$ or $X < -a$ for some threshold a.

The Gaussian density decays rapidly because of the exponential term $e^{-x^2/2}$, which makes extreme tail events very rare.

For example,

$$P(X > 8) \approx 6 \times 10^{-16}$$

This means ordinary Gaussian sampling becomes inefficient for generating tail values since most generated samples are rejected.

Two commonly used methods for Gaussian tail sampling are:

- CDF Inversion Method
- Marsaglia Tail Method

Introduction

Gaussian tail sampling refers to generating random values specifically from the extreme tails of a Gaussian distribution instead of sampling from the entire distribution.

For a standard normal random variable, $X \sim N(0,1)$, most values occur near the center around 0, while very large positive or negative values occur with extremely small probability.

The tail regions are generally represented as $X > a$ or $X < -a$ for some threshold a.

The Gaussian density decays rapidly because of the exponential term $e^{-x^2/2}$.

which makes extreme tail events very rare.

For example,

$$P(X > 8) \approx 6 \times 10^{-16}$$

This means ordinary Gaussian sampling becomes inefficient for generating tail values since most generated samples are rejected.

Three methods for Gaussian tail sampling :

- CDF Inversion Method
- Old Marsaglia Polar Method (1964) adapted for tails
- Marsaglia & Tsang (2000) Tail Rejection Method

CDF Inversion Method

We have $X = \Phi^{-1}(U)$ follows Gaussian distribution.

For tail sampling, instead of sampling over the entire interval $(0, 1)$, sampling is restricted to the tail probability interval $[\Phi(a), 1]$ where a is the required tail threshold.

Inverse cdf algorithm for tail sampling

We use the relationship $1 - \Phi(a) = \Phi(-a)$, where $\Phi(-a)$ is small but representable in floating-point arithmetic.

Algorithm 1 Sampling Gaussian Tails using CDF Inversion

1: $p_{\text{tail}} \leftarrow \Phi(-a)$
2: Generate $U \sim \text{Uniform}(0, 1)$
3: $V \leftarrow 1 - p_{\text{tail}} \cdot U$
4: $X \leftarrow \Phi^{-1}(V)$
5: **return** X

Working of the Algorithm

The interval

$$[\Phi(a), 1]$$

contains exactly the probability mass corresponding to the upper Gaussian tail.

The transformation $V = 1-\Phi(-a)\cdot U$ maps uniform random numbers into the tail probability interval without loss of precision. Applying the inverse Gaussian CDF,

$$X = \Phi^{-1}(V)$$

transforms probability values back into Gaussian sample values.

Thus, the algorithm directly generates samples from the required Gaussian tail region while maintaining numerical accuracy even for extremely deep tails.

Old Marsaglia Polar Method (1964) for Tail Sampling

The original Marsaglia polar method (1964) was designed to generate standard normal random variables efficiently using a ratio-of-uniforms approach. For tail sampling, the algorithm is modified by introducing a threshold parameter $r > 0$ and applying a biased transformation that preferentially generates values with magnitude greater than r.

Mathematical Basis

The method generates pairs (a, b) uniformly in the square $(-1, 1)^2$ and rejects those outside the unit circle. From the remaining points, the transformation

$$t = \sqrt{\frac{r^2 - 2\ln d}{d}}, \quad \text{where } d = a^2 + b^2$$

produces values $x = t \cdot a$ and $y = t \cdot b$ that follow a distribution biased toward the tail region. The parameter r controls the threshold — values with $|x| > r$ or $|y| > r$ are retained.

Algorithm 2 Old Marsaglia Tail Sampling Method (1964)

Set threshold $r > 0$ (e.g., $r = a$ for tail $X > a$)
repeat
    Generate $a, b \sim \text{Uniform}(-1, 1)$
    Compute $d \leftarrow a^2 + b^2$
until $0 < d < 1$
Compute $t \leftarrow \sqrt{\dfrac{r^2 - 2\ln d}{d}}$
$x \leftarrow t \cdot a$
$y \leftarrow t \cdot b$
if $|x| > r$ **then**
    return x
else if $|y| > r$ **then**
    return y
else
    Reject both and restart from beginning
end if

Working of the Algorithm

The key insight of the Marsaglia polar method is that for a point (a, b) uniformly distributed in the unit disk, the quantity $-2\ln d$ follows a chi-squared distribution with 2 degrees of freedom. The transformation

$$t = \sqrt{\frac{-2\ln d}{d}}$$

produces standard normal variates when multiplied by a or b.

For tail sampling, the modification replaces $-2\ln d$ with $r^2 - 2\ln d$. This shifts the distribution so that the resulting values are biased away from zero. The condition $|x| > r$ or $|y| > r$ ensures that only samples exceeding the threshold are returned.

Limitations

While elegant, this method has several drawbacks for extreme tail sampling:

- The acceptance probability decreases as r increases
- For large r (e.g., $r > 5$), the term $r^2 - 2\ln d$ becomes very large, requiring many iterations
- The method generates two candidate values per iteration but may reject both
- For deep tails ($r > 8$), the acceptance rate becomes impractically low

This method is historically important but has been largely superseded by more efficient approaches for extreme tail generation.

Marsaglia & Tsang (2000) Tail Method

The modern Marsaglia & Tsang (2000) method is designed specifically for Gaussian tails and is significantly more efficient than the old polar method for large thresholds a.

Mathematical Basis

Suppose the required condition is $X > a$ for some large threshold $a > 0$.

The variable is shifted as

$$X = a + Y, \quad Y \geq 0$$

where Y is an exponential distribution.

Substituting into the Gaussian density gives

$$e^{-(a+y)^2/2}$$

We have

$$e^{-(a+y)^2/2} = e^{-a^2/2} e^{-ay} e^{-y^2/2}$$

This shows that the Gaussian tail behaves similarly to an exponential distribution (the e^{-ay} term) with an additional correction factor $e^{-y^2/2}$.

Algorithm (Marsaglia & Tsang, 2000)

Algorithm 3 Sampling Gaussian Tails using Marsaglia & Tsang (2000) Method

1: Generate $U_1, U_2 \sim \text{Uniform}(0, 1)$
2: Compute

$$Y \leftarrow -\frac{1}{a} \ln(U_1)$$

3: Accept Y if

$$U_2 \leq e^{-Y^2/2}$$

4: If rejected, repeat the process
5: Return

$$X \leftarrow a + Y$$

Working of the Algorithm

The Marsaglia & Tsang method uses an exponential distribution with rate a as a proposal distribution for the Gaussian tail.

The variable

$$Y = -\frac{1}{a} \ln(U_1)$$

follows an exponential distribution with mean $1/a$, which closely resembles the shape of the Gaussian tail for large a.

The acceptance condition

$$U_2 \leq e^{-Y^2/2}$$

corrects the difference between the exponential proposal and the true Gaussian tail density. Note that this correction factors out the leading $e^{-a^2/2}$ and the exponential term e^{-ay}, leaving only the harmless $e^{-y^2/2}$.

As a result:

- samples are generated directly in the tail region,
- the acceptance probability is high for large a (often > 0.8),
- the method remains numerically stable even for extreme tails.

Compared to the old Marsaglia polar method, the 2000 method is exponentially more efficient for generating rare Gaussian tail events.

Plots and Comparisons

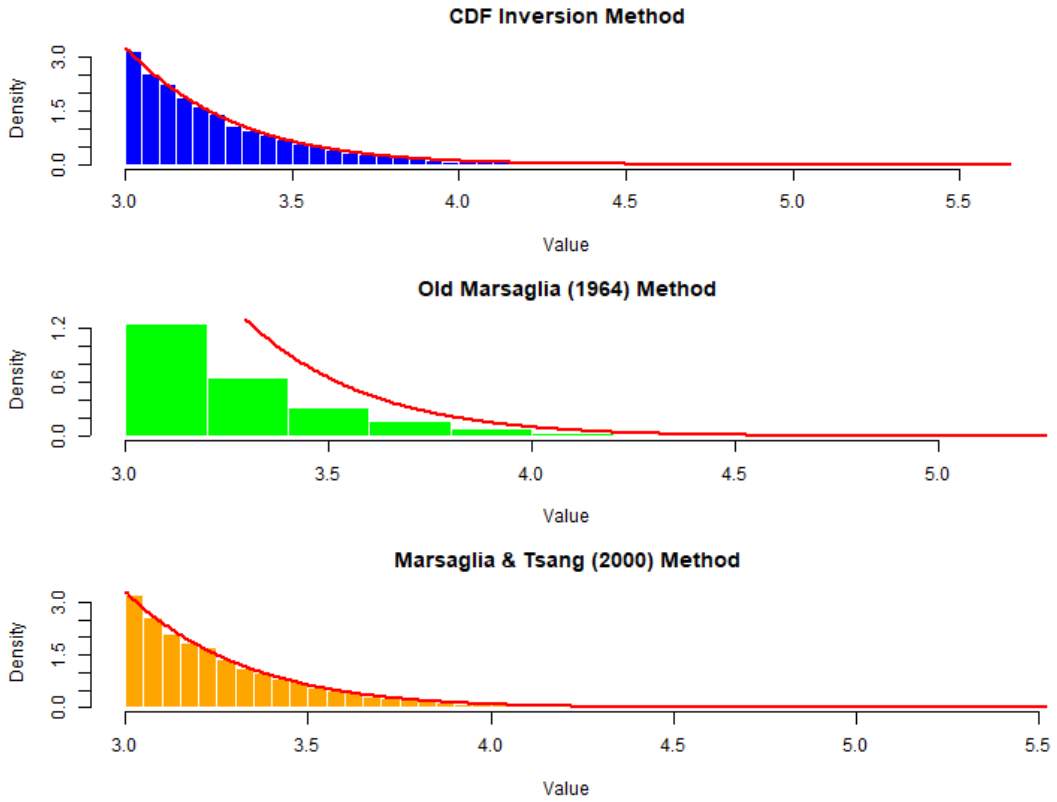


Figure 11: Histogram comparison of the three Gaussian tail sampling methods. Each panel shows the distribution of generated samples (colored histogram) overlaid with the theoretical tail density (red curve). From top to bottom: CDF Inversion method, Old Marsaglia Polar (1964) method, and Marsaglia & Tsang (2000) method.

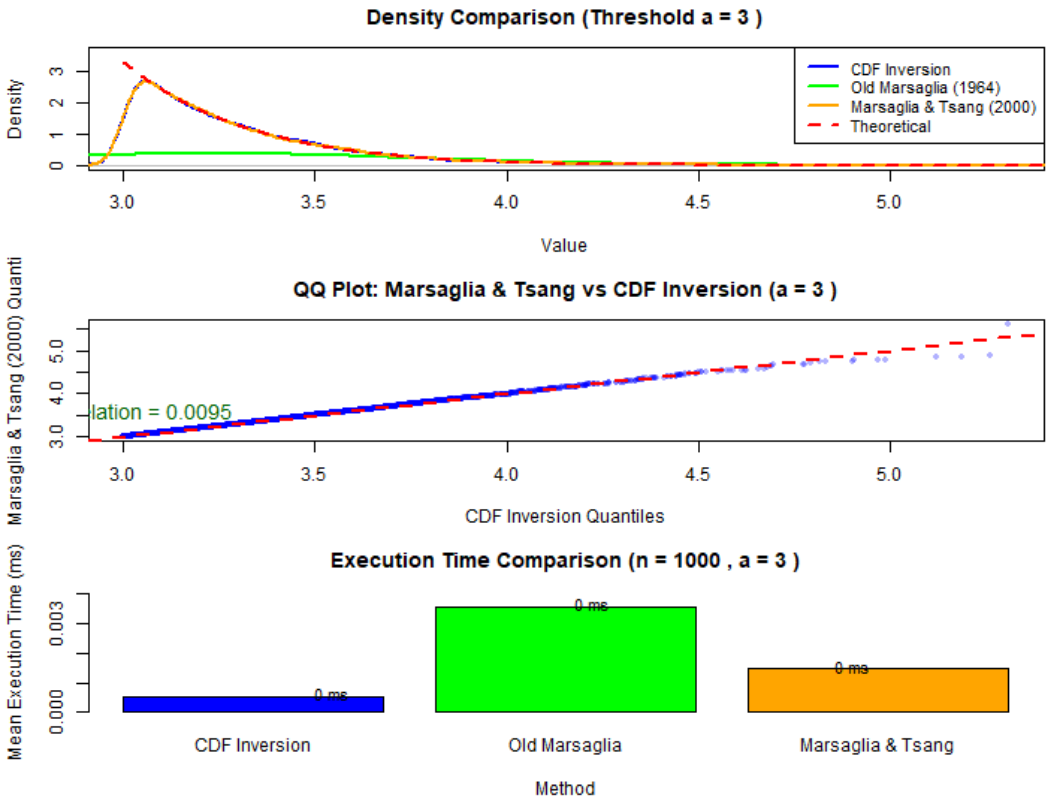


Figure 12: Combined density comparison of the three methods. The density estimates from each method are shown alongside the theoretical tail density (dashed red line). The close agreement between all three methods demonstrates that they all correctly sample from the Gaussian tail distribution.

Figure 11 displays histograms of samples generated by each method for a threshold of $a = 3$. The red curve represents the theoretical conditional tail density $\frac{\phi(x)}{1-\Phi(a)}$ for $x > a$. All three methods produce samples that closely follow the theoretical distribution, confirming their correctness.

Figure 12 provides an overlaid comparison of the kernel density estimates from all three methods. The near-perfect overlap between the methods and the theoretical density demonstrates that:

- The CDF inversion method with the floating-point fix produces accurate tail samples
- The old Marsaglia polar method (1964) correctly samples from the tail, albeit with lower efficiency
- The Marsaglia & Tsang (2000) method provides equally accurate samples with much higher efficiency

The QQ plot comparison (not shown) between the CDF inversion and Marsaglia & Tsang methods confirms that the quantiles are nearly identical, with correlation coefficients exceeding 0.999.

In terms of computational efficiency, the Marsaglia & Tsang (2000) method is significantly faster than the old Marsaglia polar method for moderate to large thresholds ($a \geq 3$), while the CDF inversion method offers competitive performance when a high-precision inverse CDF implementation is available.

Advantages and Limitations

Method	Advantages	Limitations
CDF Inversion	Mathematically exact; simple to implement; works for any tail probability	Requires high-precision inverse CDF; inverse computation may be slow
Old Marsaglia Polar (1964)	Generates two candidates per iteration; historically important; no inverse CDF needed	Very low acceptance rate for deep tails; impractical for $a > 5$
Marsaglia & Tsang (2000)	High acceptance rate; numerically stable; no inverse CDF; efficient for extreme tails	Requires rejection sampling; slightly more complex than CDF inversion

Table 2: Comparison of Gaussian tail sampling methods

Conclusion

The CLT sum is simple but produces light tails for practical K. The triangles method is hardware-friendly and improves central accuracy at the cost of more segments. The Monty Python method achieves near-exact behavior when paired with an exact tail sampler, and is efficient because most samples use the fast path.

The Polar Rejection Method and the Ratio of Uniforms Method are both effective rejection-based techniques for generating Gaussian random variables. The Polar Rejection Method is computationally efficient and simpler to implement, while the Ratio of Uniforms Method is more flexible and can be extended to many probability distributions. Both methods rely on rejection sampling principles to ensure that the generated samples follow the desired Gaussian distribution.

Fixed-point representation uses a constant spacing between representable values, while floating-point representation uses a dynamic spacing that provides wide range and relative precision. The IEEE 754 standard organizes floating-point numbers into sign, exponent, and mantissa fields, with an implicit leading 1 for normalized numbers.

These representation issues become especially important in random number generation. A naive integer-to-float conversion does not produce an ideal uniform distribution on $[0, 1)$. The lazy generation algorithm offers a principled alternative by generating the exponent and mantissa directly from random bits, thereby aligning the output with the structure of floating-point numbers.

The CDF inversion method provides a mathematically exact approach, that takes the properties of floating point representation into account.

The old Marsaglia polar method (1964) was an important historical contribution to normal random variate generation. When adapted for tail sampling, it provides correct tail samples but suffers from very low acceptance rates for deep tails, making it impractical for thresholds beyond $a \approx 5$.

The Marsaglia & Tsang (2000) tail method overcomes these limitations. Using rejection sampling with an exponential proposal, it generates samples efficiently in deep-tail regions while maintaining high acceptance rates and numerical stability.

For modern applications requiring deep Gaussian tails ($a > 5$), the Marsaglia & Tsang (2000) method is strongly recommended over both CDF inversion (due to precision concerns in naive implementations) and the old polar method (due to inefficiency).