Published in MTI Technology · Follow

SIMULATION

# How to generate Gaussian samples

## Part 2: Box-Muller transform

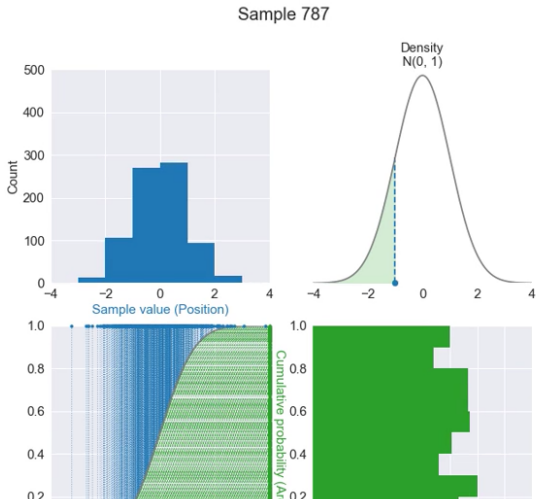
- *To see the code I wrote for this project, you can check out its Github repo*
- *For other parts of the project: part 1, part 2, part 3*

### Background

In part 1 of this project, I've shown how to generate Gaussian samples using the common technique of inversion sampling:

1. First, we sample from the uniform distribution between 0 and 1 — green points in the below animation. These uniform samples represent the cumulative probabilities of a Gaussian distribution i.e. the area under the distribution to the left of some point.
2. Next, we apply the inverse Gaussian cumulative distribution function (CDF) to these uniform samples. This

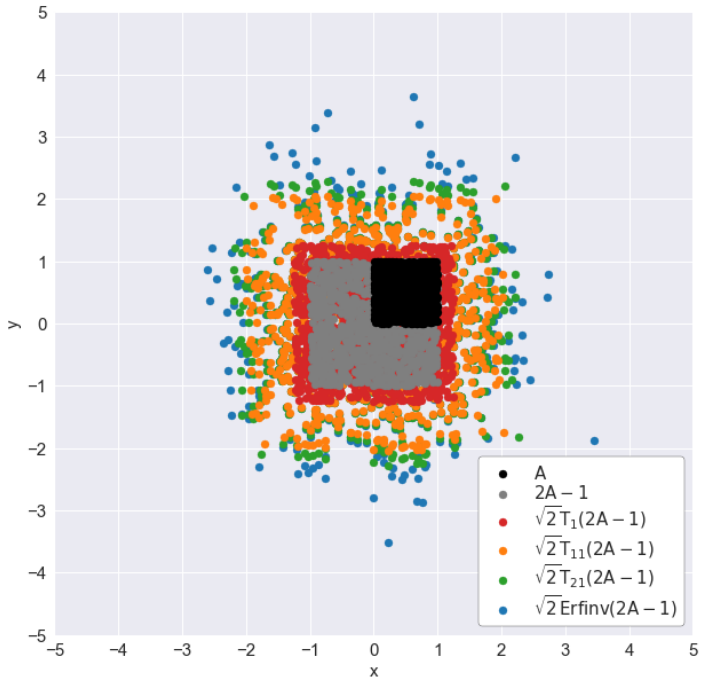
will transform them to our desired Gaussian samples — blue points in the below animation.




39.29 s SD

21.9K views gfycat

The bad news: the Gaussian inverse CDF is not well-defined, so we have to approximate that function, and the simple Taylor series was used. However, this only samples accurately near the Gaussian mean, and under-samples more extreme values at both ends of the distribution.




**A:** left-side area (uniformly sampled). **$T_i$:** the $i^{th}$-degree Taylor series approximation of the inverse Gaussian CDF: √2 Erfinv(2A-1)

Therefore, in this part of the project, we will investigate a more "direct" sampling method that does not depend on the approximation of the Gaussian inverse CDF. This method is called the **Box-Muller transform**, after the two mathematicians who invented the method in 1958: the British George E. P. Box and the American Mervin E. Muller.


**Left:** George E. P. Box (1919–2013). **Right:** Mervin E. Muller (1928–2018)

## How does the Box-Muller transform work?

For this project, my goal is to generate Gaussian samples in two dimensions i.e. generating samples whose x and y coordinates are independent standard normals (Gaussian with zero mean and standard deviation of 1). In part 1, I used the inverse Gaussian CDF to generate separately the x and y coordinates from their respective uniform samples ($U_1$ and $U_2$):

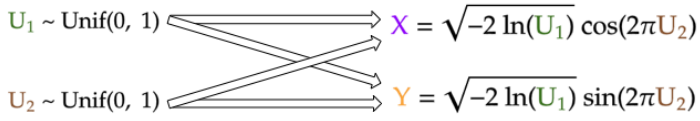
$$U_1 \sim \text{Unif}(0,\ 1) \Longrightarrow X = \sqrt{2}\,\text{erfinv}(2U_1 - 1)$$

$$U_2 \sim \text{Unif}(0,\ 1) \Longrightarrow Y = \sqrt{2}\,\text{erfinv}(2U_2 - 1)$$

Generate x and y coordinates from uniform samples ($U_1$ and $U_2$) using inverse transform sampling

For the Box-Muller transform, I will also

start with the same two uniform samples. However, I will transform these uniform samples into the x and y coordinates using much simpler formulas:



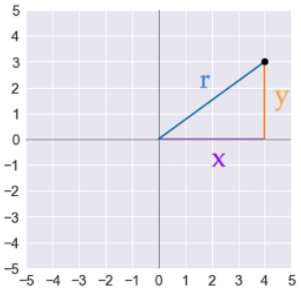

Generate x and y coordinates from uniform samples ($U_1$ and $U_2$) using Box-Muller transform

Despite the strong coupling between U1 and U2 in each of the two formulas above, the generated x and y coordinates, which are both standard Gaussians, are still surprisingly independent from each other! In the derivation of the Box-Muller transform that follows, I will demonstrate why this is indeed this case.

## Derivation of Box-Muller transform

We know that for any two independent random variables x and y, the **joint probability density f(x, y)** is simply the product of the individual density functions: f(x) and f(y). Furthermore, Pythagorus theorem allows us to combine the $x^2$ and $y^2$ term in each of the Gaussian

density function. This results in the $-r^2/2$ term in the exponential of the joint distribution, where **r** is the distance from the origin to the 2-D Gaussian sample.



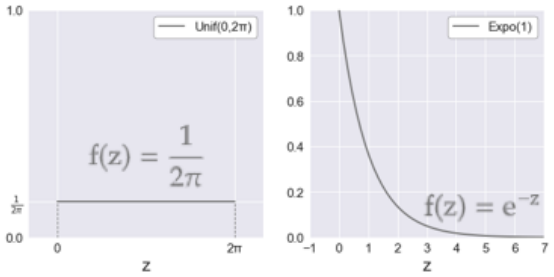

$$
\begin{aligned}
f(x, y) &= f(x) \times f(y) \\
&= \frac{1}{\sqrt{2\pi}} e^{\frac{-x^2}{2}} \frac{1}{\sqrt{2\pi}} e^{\frac{-y^2}{2}} \\
&= \frac{1}{2\pi} e^{-\frac{(x^2+y^2)}{2}} \\
&= \frac{1}{2\pi} e^{-\frac{r^2}{2}}
\end{aligned}
$$

To make it simpler, we then define a variable **s** that is equal to $r^2/2$. In other words, s is simply the **half of squared distance from the origin** of our Gaussian sample. Written this way, the joint PDF is simply the product of a constant ($1/2\pi$) and an exponential ($e^{-s}$).

$$\boxed{\text{Let } s = \frac{r^2}{2}} \iff \boxed{r = \sqrt{2s}}$$

$$\Rightarrow f(x, y) = \frac{1}{2\pi} e^{-\frac{r^2}{2}}$$

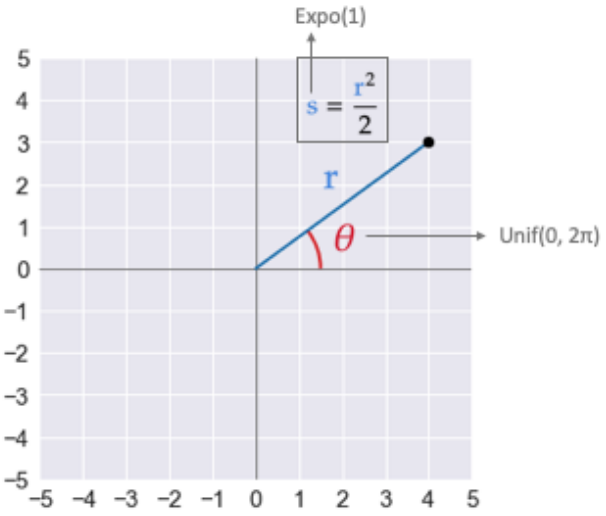
$$= \underbrace{\frac{1}{2\pi}}_{\text{Unif}(0,\,2\pi)} \underbrace{e^{-s}}_{\text{Expo}(1)}$$




This is where it gets interesting, since:

- We can think of the $1/2\pi$ constant as a uniform distribution between 0 and $2\pi$: **Unif(0, 2π)**.
- On the other hand, the exponential term is nothing but an exponential distribution with the parameter $\lambda=1$: **Expo(1)**.
- Furthermore, since these two terms are multiplied together in the joint distribution, the uniform and the exponential distributions are also independent from each other (see accompanying image).

## Geometric interpretation




Geometrically, the uniform distribution between 0 and 2π represents the angle **θ** that our 2-D Gaussian sample makes with the positive x-axis. This aligns with our intuition about Gaussian samples in 2-D: they are sampled with equal chance at *any* angle between 0 and 360 degrees, hence the characteristic round "blob" that they often make.

In contrast, these blobs are always concentrated near the origin, which fits well with the fact that **s** (half of squared distance from origin) follows an exponential distribution: you are more likely to encounter a sample as you move closer to the origin.

This leads us to the underlying principle



**Khanh Nguyen**

180 Followers

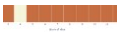
Data scientist based in Ho Chi Minh City, Vietnam | dknguyen.com

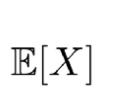
of the Box-Muller transform:

> Instead of sampling independent Gaussians for the x and y coordinates of the samples, we sample their independent uniform angles and exponential half-of-squared distances to origin.

### Caveat for changing variables

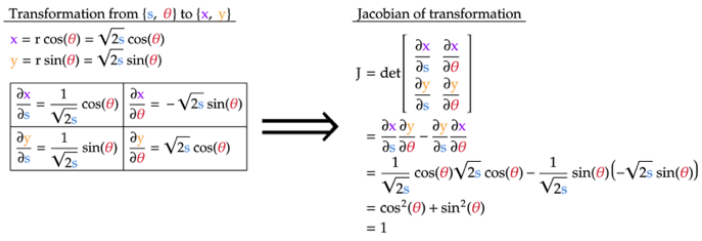
Technically, when changing random variables of a joint distribution to different ones, we need to multiply the new distribution with the Jacobian of the variable transformation (see here for more details). This ensures that the new distribution is still a valid probability density function i.e. that the "area" under the distribution is still 1. Thankfully, the Jacobian of the transformation from {s, θ} to {x, y} is 1, which greatly simplifies our problem.





What is the expected value?
A (really) simple explanation

Prediction Metrics: Recall vs. Precision with some anthropomorphic (human-like) interpretations.



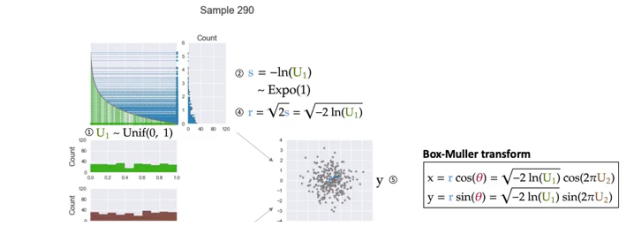

## Sampling 2-D Gaussians with Box-Muller transform

Once we transform the original problem into sampling a uniform angle and an exponential half-of-squared-distance, the remaining steps are much easier. This is because both the uniform and the exponential distribution have very simple inverse CDF, as outlined in the table below. As a result, we can apply these inverse CDFs to any uniform sample from 0 to 1 to transform it to our desired uniform and exponential sample (see part 1 of the project for an explanation of inverse transform sampling).

| Distribution | PDF | CDF | Inverse CDF |
|---|---|---|---|
| Unif(0, 2π) | $f(z) = \frac{1}{2\pi}$ | $A = \frac{1}{2\pi}z$ | $z = 2\pi A$ |
| Expo(1) | $f(z) = e^{-z}$ | $A = 1 - e^{-z}$ | $z = -\ln(1 - A)$ |

The animation below walks through all the steps needed to generate the Gaussian

samples in 2-D:




14.32 s SD

13.2K views gfycat

1. **Sample from two separate uniform sample generators** ($U_1$ and $U_2$)
2. **Apply the inverse CDF of the exponential distribution with $\lambda=1$ to $U_1$** to get half of squared distance from origin of the sample (s). For simplicity, the inverse CDF is modified from $-\ln(1-U_1)$ to $-\ln(U_1)$. As a result, this modified function is technically no longer the inverse CDF of the exponential, but it will still output samples that are exponentially distributed. This is because $U_1$ and $1-U_1$ are both uniform samples between 0 and 1.
3. **Apply the inverse CDF of the uniform distribution between 0 and $2\pi$ to $U_2$** to get the angle of the sample ($\theta$)

4. **Calculate the distance r from origin for each sample** from its half of squared distance: $r=\sqrt{(2s)}$.
5. Lastly, **the x and y coordinates are found by simple trigonometry**: $r\cos(\theta)$ and $r\sin(\theta)$ respectively. Combining the formulas from step 2 to 5 gives us the formulas seen earlier to transform $U_1$ and $U_2$ into x and y.

Coding-wise, generating Gaussian samples using the Box-Muller transform can't be any easier, as seen in this Python implementation for the 5 steps mentioned above to generate 1000 Gaussian samples in 2-D:

Note that the relevant NumPy math functions — `np.log`, `np.sqrt`, `np.sin`, and `np.cos` — are all vectorized to work on NumPy arrays (such as `u1s` and `u2s`). As a result, the 1000 Gaussian samples, whose coordinates belong to `x1s` and `y1s`, can be generated all at the same time.

## Box-Muller transform in action: NumPy's random.standard_normal

Looking under NumPy's source code, it appears that the `np.random.standard_normal` function to generate standard Gaussians indeed uses the Box-Muller transform. This is because it refers to the C function `legacy_gauss`, which uses a method based on this type of transform:

```c
double legacy_gauss(aug_bitgen_t *aug_state) {
  if (aug_state->has_gauss) {
    const double temp = aug_state->gauss;
    aug_state->has_gauss = false;
    aug_state->gauss = 0.0;
    return temp;
  } else {
    double f, x1, x2, r2;

    do {
      x1 = 2.0 * legacy_double(aug_state) - 1.0;
      x2 = 2.0 * legacy_double(aug_state) - 1.0;
      r2 = x1 * x1 + x2 * x2;
    } while (r2 >= 1.0 || r2 == 0.0);

    /* Polar method, a more efficient version of the Box-Muller approach. */
    f = sqrt(-2.0 * log(r2) / r2);
    /* Keep for next call */
    aug_state->gauss = f * x1;
    aug_state->has_gauss = true;
    return f * x2;
  }
}
```

numpy/numpy/random/src/legacy/legacy-distributions.c

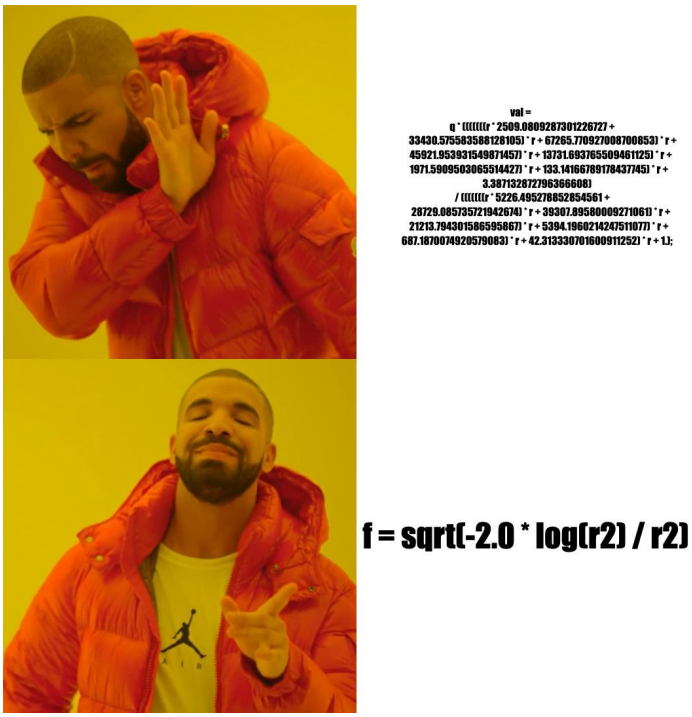
There are 3 things to note for this implementation:

1. It implements the polar form of the Box-Muller transform, which is also called the Marsaglia polar method. This method uses some clever geometry to avoid computing the sine and cosine function of the traditional Box-Muller transform, thus makes it faster. I'm planning to implement this polar method in the near future, which I will add to this blog post.
2. The NumPy's implementation discards one of the two generated Gaussians from the Box-Muller transform. As a result, only one Gaussian sample is returned, hence the `return f * x2` line in `legacy_gauss`.

3. NumPy's newer random number generator class, the appropriately-named `Generator`, does not use the Box-Muller transform anymore. Instead, according to this changelog, it uses the Ziggurat algorithm (wiki, implementation) to generate the samples, which are much faster than both the Box-Muller transform or the Marsaglia polar method.

## Comparison to R's rnorm

As detailed in part 1, R uses inversion transform sampling to generate its Gaussian samples, as opposed to the Box-Muller transform used by Python's NumPy. I have no authority to comment on the relative accuracy or computational efficiency of either approach. However, in terms of elegance and readability, Python's approach is clearly superior to the R's hard-coded mess in the previous part. This is yet another proof that Python is clearly the better programming language *#kidding #notreally*.




**Top:** implementation of R's qnorm. **Bottom:** implementation of NumPy's random.standard_normal

Nevertheless, in the next part of the project, I will demonstrate another technique to generate Gaussian samples that are even simpler than the Box-Muller, and that every statistics student are already familiar with! Please stay tuned.

## Reference

All the derivation for the Box-Muller transform in this blog post is taken from the "Introduction to Probability" book by Blitzstein and Hwang (available for free online, page 373–374).