Published in MTI Technology · Follow

SIMULATION

How to generate Gaussian samples

Part 3: Central limit theorem

- *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

The goal of this project is to generate Gaussian samples in 2-D from uniform samples, the latter of which can be readily generated using built-in random number generators in most computer languages.

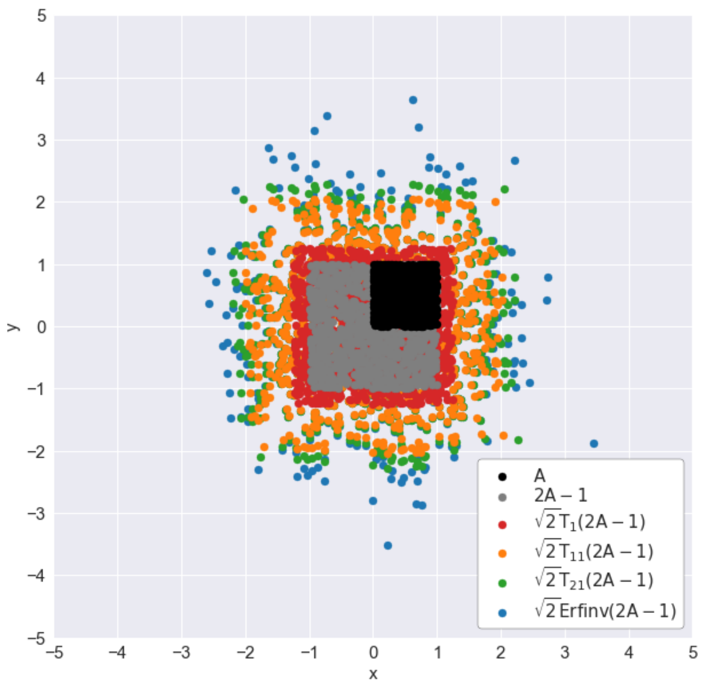
In part 1 of the project, the inverse transform sampling was used to convert each uniform sample into respective x and y coordinates of our Gaussian samples, which are themselves independent standard normal (having mean of 0 and standard deviation of 1):

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

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

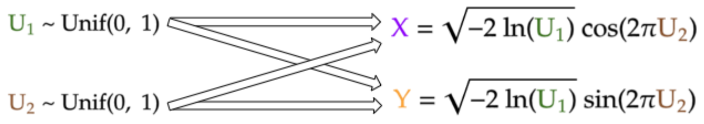
Generate x and y coordinates from uniform samples (U_1 and U_2) using inverse transform sampling

However, this method uses the inverse cumulative distribution function (CDF) of the Gaussian distribution, which is not well-defined. Therefore, we approximated this function using the simple Taylor series. 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 ith-degree Taylor series approximation of the inverse Gaussian CDF: √2 Erfinv(2A-1)

In part 2 of the project, we used the Box-Muller transform, a more direct method to transform the uniform samples into Gaussian ones. The implementation of the algorithm is quite simple, as seen below, but its derivation requires some clever change of variables: instead of sampling Gaussian x and y coordinates for each point, we will sample a uniformly-distributed angle (from 0 to 2π) and an exponentially-distributed random variable that represents half of squared distance of the sample to the origin.


$U_1 \sim \text{Unif}(0, 1)$
$U_2 \sim \text{Unif}(0, 1)$
$X = \sqrt{-2\ln(U_1)}\cos(2\pi U_2)$
$Y = \sqrt{-2\ln(U_1)}\sin(2\pi U_2)$

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

In this part of the project, I will present an even simpler method than the above two methods. Even better, this method is one that every statistics students are already familiar with.

The Central Limit Theorem

It turns out, we can rely on the most

fundamental principle of statistics to help us generate Gaussian samples: the **central limit theorem**. In very simple terms, the central limit theorem states that:

> Given n independent random samples from the same distribution, their sum will converge to a Gaussian distribution as n gets large.

Therefore, to generate a Gaussian sample, we can just generate many independent uniform samples and add them together! We then repeat this routine to generate the next standard Gaussian sample until we get enough samples for our x-coordinates. Finally, we just repeat the same steps to generate the y coordinates.

$$U_1, U_2, \ldots, U_n \sim \text{Unif}(0, 1) \Longrightarrow X = \sum_{i=1}^{n} U_i$$

$$U'_1, U'_2, \ldots, U'_n \sim \text{Unif}(0, 1) \Longrightarrow Y = \sum_{i=1}^{n} U'_i$$

Note that this method will work even if

the samples that we start with are not uniform — they are Poisson-distributed, for example. This is because the central limit theorem holds for virtually all probability distributions **cough let's not talk about the Cauchy distribution cough**.

Generate Gaussian samples by central limit theorem

To generate, say, 1000 Gaussian sums (`n_sums = 1000`) where each is the sum of 100 uniform samples (`n_additions = 100`):

1. First, we generate a 100 x 1000 matrix — `uniform_matrix` — in which every element is a uniformly-distributed sample between 0 and 1.
2. Then, calculating the 1000 sums is nothing but summing the uniform samples down each column of this matrix, hence the `axis=0` argument. The result is a NumPy array `gaussians`, which contains the 1000 Gaussian samples.

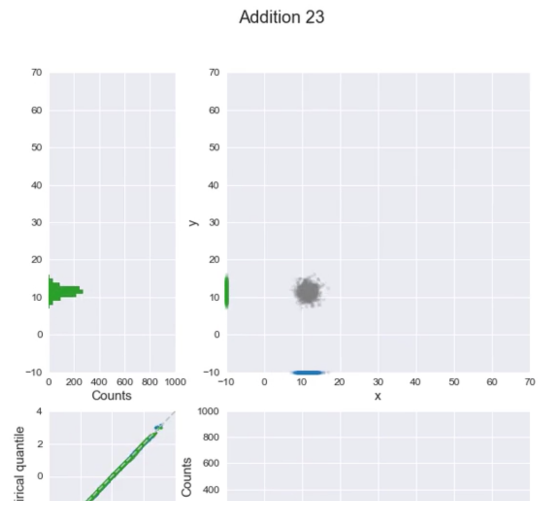
```
n_additions = 100
n_points = 1000
```

```
# 0. Initialize random number
generator
rng =
np.random.RandomState(seed=24)

# 1. Generate matrix of uniform
samples
uniform_matrix = rng.uniform(size=
(n_additions, n_points))

# 2. Sum uniform elements down
each column to get all Gaussian
sums
gaussians =
uniform_matrix.sum(axis=0)
```

We can apply the above method to generate Gaussian samples for each coordinate, using different random number generator for x and for y to ensure that the coordinates are independent from each other. Visualizing the intermediate sums after each addition, we see that:



2.25 s SD

2.2K views gfycat

Generate 1000 Gaussian samples in 2-D using central limit theorem

- The 2-D samples start out confined in a length-one block near the origin, since their coordinates all began as uniform samples between 0 and 1. However, as more and more samples are added on top of each uniform sample, they start to "grow" toward the familiar Gaussian blob in the xy-plane.
- This is also reflected in the histogram of the sampled x and y coordinates: they start out as single peaks that represent the Unif(0, 1) distribution. As more samples are added, they level out towards the familiar bell curve of the Gaussian distribution.
- Lastly, the QQ-plot of the samples first followed the S-shaped pattern of lighter-tailed distributions , which include the uniform. Then, it quickly conforms to the diagonal line, even when only 10 samples have been added. In other words, adding even just a few uniform samples already gives us sums that are characteristically Gaussian.

In short, the central limit theorem allows us to easily generate Gaussian samples in 2-D, whose x and y coordinates are the Gaussian sums of many uniform samples. However, we still need to rescale these x and y coordinates so that they return to standard normal (mean of 0 and standard deviation of 1).

Rescale Gaussian samples

Rescaling the Gaussian samples means we have to subtract each sum by its mean, and divide by its standard deviation.

- For a sum of n uniform samples, the mean of the sum is simply the sum of all the uniform means, which are 1/2 each. Therefore, the mean of the generated Gaussian sample is n/2.
- Furthermore, since these uniform samples are independent, the variance of their sum is simply the sum of the uniform variances, which are 1/12 each. Hence, the variance of the generated Gaussian sample is n/12, and its standard deviation is √(n/12).

$$X = \sum_{i=1}^{n} U_i \xrightarrow{n\to\infty} X \sim N\left(\frac{n}{2}, \frac{n}{12}\right)$$

$$Y = \sum_{i=1}^{n} U_i' \xrightarrow{n\to\infty} Y \sim N\left(\frac{n}{2}, \frac{n}{12}\right)$$

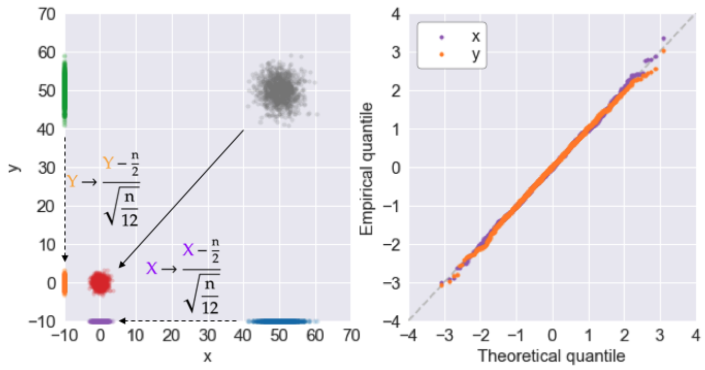
As a result, the Gaussian samples that represents the x and y coordinates can be normalized as follows:

$$X \to \frac{X - \frac{n}{2}}{\sqrt{\frac{n}{12}}} \qquad Y \to \frac{Y - \frac{n}{2}}{\sqrt{\frac{n}{12}}}$$

```
from math import sqrt

gaussian_mean = n_additions / 2
gaussian_std = sqrt(n_additions /
12)
normalized_gaussians = (gaussians
- gaussian_mean) / gaussian_std
```

The result of such rescaling is seen below:



In the above graph, the rescaled x and y coordinates are still normal, as evidenced by their diagonal QQ-plots. Furthermore, the coordinates are still independent from each other, since they were generated from different uniform samples in the beginning. This means that they are also uncorrelated, as evidenced by the nice round blob of Gaussian samples at the origin.

Transform Gaussian samples

Once we have our standard normal sample, we can extend the problem to generating Gaussian samples with *any* mean and variance in the x and y directions, as well as with *any* covariance between x and y.

We can do so by first looking at the vector

z (also called the Gaussian vector) that contains the standard normal x and y coordinates of a given sample:

$$\vec{Z} = \begin{bmatrix} X \\ Y \end{bmatrix} \sim \mathrm{N}\left(\overrightarrow{\mu_Z}, \mathbb{C}\mathrm{ov}(Z)\right) - \left[\begin{array}{l} \textbf{Mean vector}: \overrightarrow{\mu_Z} = \begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \textbf{Covariance matrix}: \mathbb{C}\mathrm{ov}(Z) = \begin{bmatrix} \mathrm{Var}(X) & \mathrm{Cov}(X, Y) \\ \mathrm{Cov}(Y, X) & \mathrm{Var}(Y) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{array}\right.$$

This Gaussian vector is characterized by:

- A **mean vector**, which represents the mean for each element (x and y). For our Gaussian sample, the mean vector is a zero vector of length two, since both its x and y coordinates have zero mean.
- A symmetric **covariance matrix**, whose diagonal entries represent the variance for each coordinate, while the remaining entries represent the covariance between coordinates (how much x varies when y varies, and vice versa). In this case, the covariance matrix is a 2x2 identity matrix: the two diagonal entries of the matrix are 1, since the x and y coordinates have variance of 1, while the other two entries are 0, since the coordinates are independent, hence uncorrelated.

Linear transformation of a Gaussian vector

Given our Gaussian vector z, the linear transformation of z — multiplying z by a matrix A and add a vector b on top — is also a Gaussian vector. Furthermore:

- Its mean is the same linear transformation applied to the mean vector of z,
- Its covariance matrix is the original covariance matrix multiplied by A on the left, and A^T (transpose of A) on the right.

$$A\,\vec{Z} + \vec{b} \sim N\left(A\overrightarrow{\mu_Z} + \vec{b}\,,\; A\,\mathbb{C}\text{ov}(Z)\,A^T\right)$$

As a result, transforming each standard normal sample z to have any given mean, variance, and covariance between x and y coordinates simply boils down to finding the right A and b to apply to each sample.

Example

Here's a concrete example to demonstrate how this works: given our current standard normal samples, we want to transform them into Gaussian samples whose x coordinate has a mean of 7 and a variance of 4, whose y coordinate has a mean of 9 and a variance of 3, and whose covariance between x and y is 2.

Given: $\vec{Z} = \begin{bmatrix} X \\ Y \end{bmatrix} \sim N\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right)$

Goal: Find A and $\vec{b}$ such that
$A\vec{Z} + \vec{b} \sim N\left(\begin{bmatrix} 7 \\ 9 \end{bmatrix}, \begin{bmatrix} 4 & 2 \\ 2 & 3 \end{bmatrix}\right)$

From the transformation rules outlined earlier:

- The new mean vector [7, 9] will be the linear transformation of the original mean vector [0, 0]. This straight away gives us the value for the vector b, which is also [7, 9].

$$A\overrightarrow{\mu_Z} + \vec{b} = A\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \vec{b} = \begin{bmatrix} 7 \\ 9 \end{bmatrix} \Longrightarrow \vec{b} = \begin{bmatrix} 7 \\ 9 \end{bmatrix}$$

- The new covariance vector M will be the identity matrix — the original covariance matrix — multiplied by A on the left and A^T on the right. As a result, the only thing left is to find a matrix A such that $AA^T = M$.

$$A\,\mathrm{Cov}(Z)\,A^T = A\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}A^T = \begin{bmatrix}4 & 2\\ 2 & 3\end{bmatrix} \Longrightarrow AA^T = \underbrace{\begin{bmatrix}4 & 2\\ 2 & 3\end{bmatrix}}_{M}$$

Finding A

Cholesky decomposition

Those well-versed in linear algebra will immediately recognize that the problem of finding A already has a well-known solution: A can be found from the Cholesky decomposition of M. Thankfully, NumPy comes installed with its own Cholesky decomposition method, which we can use to decompose the new covariance matrix into A:

```
M = np.array([[4, 2], [2, 3]])
A = np.linalg.cholesky(M)

# array([[2.        , 0.        ],
#        [1.        ,
1.41421356]])
```

Get started Sign In

Search

Khanh Nguyen

180 Followers

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

Follow

Related

Support Vector Machine: Theory and Practice

With A and b known, we can just apply them to any Gaussian sample z in 2-D (a NumPy array of length two) to get our transformed Gaussian sample.

```
b = np.array([7, 9])
z = np.array([0.276, -0.394])
z_transformed = A @ z + b

# array([7.552, 8.71879986])
```

Singular value decomposition

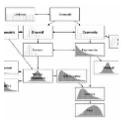
Another method to get the matrix A is through singular value decomposition (SVD) of the covariance matrix M. This decomposition gives us 3 matrices:

1. **U**: matrix whose columns are eigenvectors of the MM^T matrix
2. **S**: diagonal matrix whose diagonal entries are square roots of the eigenvalues of M^TM (or MM^T). These square roots are also called the singular values of M.
3. **V**: matrix whose rows are eigenvectors of the M^TM matrix

In this case:

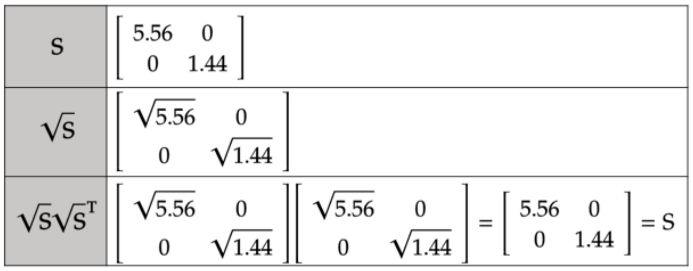
Understand SVM, one of the…

Statistical languages (mostly) don't matter.
Are you looking to start a…


Continuous Probability Distribution
Definition

Understanding decision trees (CART)
When starting your…

- Since M is a symmetric matrix ($M^T = M$), the matrices MM^T and M^TM are one and the same. As a result, the matrices U and V are simply transpose of each other: $V = U^T$.
- We can split the S matrix into a product of √S and $(\sqrt{S})^T$, where √S is a diagonal matrix whose diagonal entries are square root of the diagonal entries in S.



S	$\begin{bmatrix} 5.56 & 0 \\ 0 & 1.44 \end{bmatrix}$
$\sqrt{S}$	$\begin{bmatrix} \sqrt{5.56} & 0 \\ 0 & \sqrt{1.44} \end{bmatrix}$
$\sqrt{S}\sqrt{S}^T$	$\begin{bmatrix} \sqrt{5.56} & 0 \\ 0 & \sqrt{1.44} \end{bmatrix}\begin{bmatrix} \sqrt{5.56} & 0 \\ 0 & \sqrt{1.44} \end{bmatrix} = \begin{bmatrix} 5.56 & 0 \\ 0 & 1.44 \end{bmatrix} = S$

As a result, the SVD can be rewritten as:

$$\begin{aligned} M &= U\,S\,V \\ &= U\,S\,U^T \\ &= U\sqrt{S}\sqrt{S}^T U^T \\ &= \underbrace{U\sqrt{S}}_{A}\underbrace{\left(U\sqrt{S}\right)^T}_{A^T} \end{aligned}$$

Written this way, it is clear that our matrix

A is just U multiplied by √S:

```
M = np.array([[4, 2], [2, 3]])
U, S, V = np.linalg.svd(M)
A = U @ np.diag(S)**(1/2)

# array([[-1.85882053,
-0.73809637],
#        [-1.45132321,
0.94533641]])
```

We can then apply this A, along with the vector b, to the standard normal vector to get our transformed Gaussian sample in 2-D.

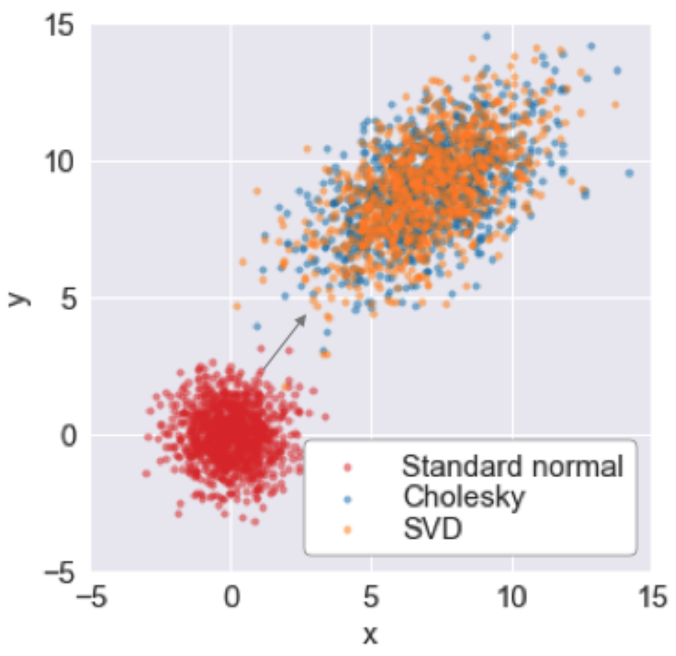
```
b = np.array([7, 9])
z = np.array([0.276, -0.394])
z_transformed = A @ z + b

# array([6.7777755, 8.22697225])
```

Comparison between Cholesky decomposition and SVD

From the results above, notice that the Cholesky decomposition and SVD return different A matrices: the Cholesky decomposition always returns A as a lower triangular matrix, while no such restriction exists for SVD. However, these two matrices still satisfy $AA^T = M$.

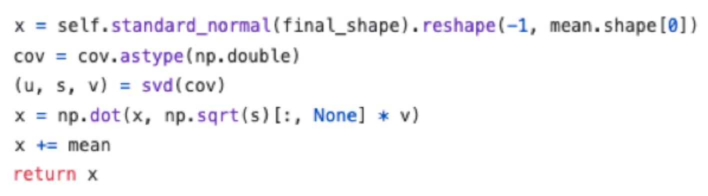
As a result of these different A matrices, the transformed sample from each method is also slightly different from the other method. Indeed, when plotting 1000 transformed Gaussian samples from both methods, we see that these methods do not transform each standard normal samples into exactly the same point in 2-D:



Nevertheless, the two transformed "blobs" are overall the same. In other words, the samples will still have the desired mean, variances, and covariance between the x and y coordinates regardless of which transformation method we use.

Transformation in action: NumPy's random.multivariate_normal

Even though the Cholesky decomposition seems like the more straight-forward method, the reason why I mentioned the SVD is because it is used by `np.random.multivariate_normal` (doc, source code): it first generates a standard normal vector — using the Box-Muller transform of `standard_normal` — then uses SVD to transform the vector to have the desired mean, variances, and covariances. The only difference with our method above is that their matrix A is now √SV instead of our U√S, but the effect is still the same.

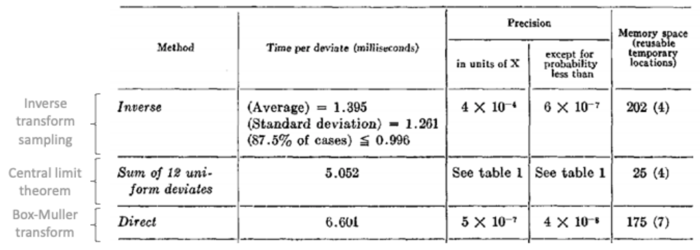


```
x = self.standard_normal(final_shape).reshape(-1, mean.shape[0])
cov = cov.astype(np.double)
(u, s, v) = svd(cov)
x = np.dot(x, np.sqrt(s)[:, None] * v)
x += mean
return x
```

Implementation of np.random.multivariate_normal. Some lines removed for clarity

Final remarks

For comparisons between the 3 methods to generate Gaussian samples — inverse transform sampling in part 1, Box-Muller transform in part 2, and central limit theorem in this part — Muller (of the Box-Muller fame) did such a comparison all the way back in 1959. In fact, it is through this paper that I first discovered the ingenious method of using the central limit theorem to generate Gaussian samples, and ended up with the biggest case of "why haven't I thought of that before?!".



	Method	Time per deviate (milliseconds)	Precision		Memory space (reusable temporary locations)
			in units of X	except for probability less than	
Inverse transform sampling	*Inverse*	(Average) = 1.395 (Standard deviation) = 1.261 (87.5% of cases) ≦ 0.996	4×10^{-4}	6×10^{-7}	202 (4)
Central limit theorem	*Sum of 12 uniform deviates*	5.052	See table 1	See table 1	25 (4)
Box-Muller transform	*Direct*	6.601	5×10^{-7}	4×10^{-8}	175 (7)

Muller 1959 "A Comparison of Methods for Generating Normal Deviates on Digital Computers"

I'm not well-versed in numerical analysis to really comment on the relative performance, in speed or in accuracy, between these 3 methods. However, in my opinion, the Box-Muller transform strikes the perfect balance between simplicity and elegance, as it’s the only method that generates the Gaussian samples “directly” in just a few lines of code.

In contrast, the other two methods still have to resort to approximations: the inverse sampling method approximates the inverse Gaussian CDF, while under the central limit theorem, the sum of *finite* uniform samples is only an approximation of a Gaussian sample.

I’m planning to research even more methods to generate Gaussians in the future, so please stay tuned! In the mean time, thank you for taking the time to read the first three parts of my project. I hope you learned a few interesting things about generating Gaussians like I did during my research for this project 🎉