Random Numbers and Distributions

A. Ridder

June 6, 2022

Abstract

This note describes random number generating in Python, and it gives a summary of a few well-known discrete and continuous distributions and their usage in simulations. Python code is included for non-trivial cases.

Contents

1 Random Numbers and Simulation **2**
- 1.1 Random Number Generator . . . 2
- 1.2 Random Data and Distributions . . . 3
- 1.3 Random Sample from a Finite Distribution . . . 4
- 1.4 Random Permutations . . . 4
- 1.5 Random Finite Probability Distribution . . . 4
- 1.6 Legacy Random Generation . . . 5

2 Special discrete distributions **5**
- 2.1 Bernoulli $\mathsf{Ber}(p)$ . . . 5
- 2.2 Uniform $\{a, a+1, \ldots, b\}$ . . . 5
- 2.3 Binomial $\mathsf{Bin}(n, p)$ . . . 5
- 2.4 Geometric $\mathsf{Geo}(p)$ . . . 6
- 2.5 Negative Binomial $\mathsf{NBin}(p, r)$ . . . 6
- 2.6 Poisson $\mathsf{Poi}(\lambda)$ . . . 7
  - 2.6.1 Python . . . 8

3 Special continuous distributions **8**
- 3.1 Uniform $\mathsf{U}(a, b)$ . . . 8
- 3.2 Exponential $\mathsf{Exp}(\lambda)$ . . . 8
  - 3.2.1 Python . . . 8
- 3.3 Laplace (λ) . . . 9
  - 3.3.1 Python . . . 9
- 3.4 Erlang $\mathsf{Erl}(k, \lambda)$ . . . 9
  - 3.4.1 Python . . . 10
- 3.5 Hyperexponential $(p, \lambda_1, \lambda_2)$ . . . 10
- 3.6 Coxian $\mathsf{Cox}(p, \lambda_1, \lambda_2)$ . . . 10
- 3.7 Gamma $\mathsf{Gam}(\alpha, \lambda)$ . . . 11
  - 3.7.1 Python . . . 13
- 3.8 Beta $\mathsf{Beta}(\alpha, \beta)$ . . . 13
  - 3.8.1 Python . . . 14
- 3.9 Weibull $\mathsf{Wei}(\alpha, \lambda)$ . . . 14

3.9.1 Python . . . 15
3.10 Normal $\mathsf{N}(\mu, \sigma^2)$ . . . 15
3.10.1 Python . . . 16
3.11 Lognormal $\mathsf{LN}(\mu, \sigma^2)$ . . . 16
3.11.1 Python . . . 16
3.12 Pareto $\mathsf{Par}(\alpha, \lambda)$ . . . 17
3.12.1 Python . . . 17

4 Multivariate Normal **17**
4.1 Cholesky Factorization . . . 18
4.2 Python . . . 18

1 Random Numbers and Simulation

The module `random` in NumPy is convenient for generating random numbers for simulation purposes. Documentation in

https://numpy.org/doc/stable/reference/random/index.html

```
import numpy.random as rnd
```

1.1 Random Number Generator

Stochastic simulation is build upon the availability of random independent uniform (0,1) numbers $U_1, U_2, \ldots$. By the inverse transform method (or other algorithms) these uniform numbers are transformed to random numbers with other distributions, and these are used in stochastic processes.

For several reasons (see below), you do not wish true random numbers in computer simulation. There are algorithms that produce so-called pseudo-random numbers $u_1, u_2, \ldots$ that can be considered to be a realisation of the (perfect) $U_1, U_2, \ldots$. Such an algorithm is called *random number generator* (RNG). There are many RNG's. A good RNG satisfies the following criteria.

1. The produced sequence of numbers $u_1, u_2, \ldots$ passes successfully statistical tests on uniformity and independence.
2. The algorithm is very fast, e.g., 10M numbers in one second.
3. The sequence should be repeatable!
4. Ability of producing separate streams.
5. The algorithm is portable (works the same on different computers and operating systems).
6. The period of the sequence should be large. Explanation: computers are finite machines, thus at the end any (mathematically) infinitely long sequence repeats itself after what is called *the period*.

The default RNG in the module `random` of Numpy is the *PCG-64* which satisfies all these criteria with a period of 2^{128}.

The first number that is produced by the RNG depends on how the RNG is started. This is defined by an integer, called *seed*. Given the same seed, you get the same sequence of random numbers. Python's RNG starts up with a fresh unpredictable entropy pulled from the

operating system. Thus, you need to set the seed by your self when you wish to reproduce the same sequence. In this way you can repeat a simulation experiment with the same sequence of random numbers! This is very important in scenario analysis, design optimization, etc. Here is an example.

```
import numpy.random as rnd

seed = 745795 # just any positive integer will do
rng = rnd.default_rng(seed)
# do an experiment, e.g.
x = rng.random(100)<0.5 # 100 tosses with fair coin

rng = rnd.default_rng(seed)
y = rng.random(100)<0.6 # 100 tosses with unfair coin
```

The 100 outcomes of the fair and unfair coins used the same random numbers. Thus for sure, if $x_i = 1$, then $y_i = 1$.

1.2 Random Data and Distributions

Using the random number generator you can sample from some well known distributions, check https://numpy.org/doc/stable/reference/random/generator.html

Table 1: Some available functions to generate random numbers.

function	description
`random`	uniform in $[0, 1)$
`integers`	random integers
`choice`	from a discrete distribution
`standard_normal`	from the standard Gaussian distribution
`binomial`	from binomial distribution
`gamma`	from Gamma distribution

Here are examples how to use these.

```
rng = rnd.default_rng()
u = rng.random() # one (0,1)-random number
u = rng.random(4) # array of four (0,1)-random numbers
u = rng.random([4,5]) # 4 x 5 matrix of (0,1)-random numbers
u = rng.random([4,5,3,3]) # 4 x 5 x 3 x 3 matrix
```

```
rng = rnd.default_rng()
x = rng.integers(5) # random in {0,1,2,3,4}
x = rng.integers(3,8) # random in {3,4,5,6,7}
x = rng.integers(3,8,size=10) # 10 times random in {3,4,5,6,7}
x = rng.integers(7,size=(4,5,3,3)) # 4 x 5 x 3 x 3 matrix
                    # of random integers in {0,1,...,6}
```

```
rng = rnd.default_rng()
z = rng.standard_normal() # one random Gaussian number
z = rng.standard_normal(4) # array of four random Gaussian numbers
z = rng.standard_normal([4,5]) # 4 x 5 matrix of random Gaussian numbers
```

```
z = rng.standard_normal([4,5,3,3]) # 4 x 5 x 3 x 3 matrix
```

1.3 Random Sample from a Finite Distribution

Assume that a finite distribution (say of size n) is given on state space $S = \{s_1, \ldots, s_n\}$ by a probabilities $p_1, \ldots, p_n$. Suppose that you want to generate m i.i.d. samples from this distribution; i.e., with replacement. The method for this is `choice()` which you call with arguments s, m, p. Example (as showing how to call another RNG)

```
prob = np.array([1,2,3,4,5,5,4,3,2,1])
prob = prob / np.sum(prob)
n = len(prob)
s = np.linspace(-np.pi,np.pi,n)
m = 100
rng = rnd.Generator_rng(rnd.MT19937) # Mersenne Twister
x = rng.choice(s, size=m, p=prob)
```

When you do not specify p, the samples are generated according to the uniform distribution on S. When the state space is $S = \{0, 1, \ldots, n-1\}$ it suffices to just give the size n:

```
prob = np.array([1,2,3,4,5,5,4,3,2,1])
prob = prob / np.sum(prob)
n = len(prob)
m = 100
x = rng.choice(n, size=m, p=prob)
```

You generates samples without replacement by the argument `replace=False`:

```
prob = np.array([1,2,3,4,5,5,4,3,2,1])
prob = prob / np.sum(prob)
n = len(prob)
m = 5
x = rng.choice(n, size=m, replace=False, p=prob)
```

1.4 Random Permutations

`permutation(n)` produces a random permutation of $0, 1, \ldots, n-1$. `shuffle(x)` shuffles randomly the elements of array x.

```
>>> rng = np.random.default_rng()
>>> rng.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6]) # random
```

1.5 Random Finite Probability Distribution

Suppose that you want to generate a random probability distribution $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$ on $\{0, 1, \ldots, n-1\}$. Intuitively it is clear what is meant: the $p_i \geq 0$ are random such that they add up to 1. Formally it is a bit more involved to give a precise definition. The intuitive view leads to a simple algorithm: generate $q_0, q_1, \ldots, q_n$ i.i.d. random from the uniform $(0, 1)$ distribution, and then divide by their sum.

```
rng = np.random.default_rng()
q = rng.random(n)
```

```
s = np.sum(q)
p = q/s
```

1.6 Legacy Random Generation

The class `RandomState` is NumPy's legacy generator based on the Mersenne Twister *MT19937* which is considered to be slow in comparison to *PCG-64*. It remains available but will not get improvements. It has the same functionalities as the newer the `Generator` class. Seeding and calls to random uniforms, integers, standard normals, finite distributions goes as follows.

```
import numpy as np
import numpy.random as rnd

seed = 12345
rnd.seed(seed) # sets the seed of the MT to <seed>

u = rnd.rand() # single
u = rnd.rand(100) # array of size 100
j = rnd.randint(5) # random in {0,1,2,3,4}
j = rnd.randint(3,8) # random in {3,4,5,6,7}
j = rnd.randint(3,8,size=10) # 10 times random in {3,4,5,6,7}
z = rnd.randn() # single
z = rnd.randn(2000) # array of size 2000
x = rnd.choice(np.array([-1,0,1]), size=5, p=np.array([0.2,0.3,0.5]))
# 5 times with replacements from {-1,0,1} according to p
```

2 Special discrete distributions

Set $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0,1)$, which can be generated by the RNG as above. Conveniently we use $U \overset{\mathcal{D}}{=} 1-U$.

2.1 Bernoulli $\mathsf{Ber}(p)$

Parameter $p \in [0,1]$. Pmf $\mathbb{P}(X=1)=p$, $\mathbb{P}(X=0)=1-p$, $\mathbb{E}[X]=p$, $\mathbb{V}ar[X]=p(1-p)$. Inverse transform:

$$X = 1\{1-U<p\} \overset{\mathcal{D}}{=} 1\{U<p\}.$$

2.2 Uniform $\{a, a+1, \ldots, b\}$

Suppose $a,b \in \mathbb{Z}, a<b$. $\mathbb{E}[X]=(a+b)/2$, $\mathbb{V}ar[X]=((b-a+1)^2-1)/12$. Inverse transform:

$$X = a + \text{floor}((b-a+1)U).$$

2.3 Binomial $\mathsf{Bin}(n,p)$

Parameters $n \in \mathbb{N}, p \in [0,1]$. X is the number of successes in n iid Bernoulli trials where each trial has success probability p. Pmf:

$$\mathbb{P}(X=k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k=0,\ldots,n.$$

As a sum of iid Bernoulli's: $X = \sum_{i=1}^{n} B_i$. $\mathbb{E}[X] = np$, $\mathbb{V}ar[X] = np(1-p)$. Simulation:

$$X = \sum_{i=1}^{n} 1\{U_i < p\}.$$

2.4 Geometric $\mathsf{Geo}(p)$

Parameter $0 < p < 1$. X is the number of failures in a series of iid Bernoulli trials until the first success. Each trial has success with probability p. Pmf and moments

$$\mathbb{P}(X = k) = (1-p)^k p,\ k = 0, 1, \ldots;\ \mathbb{E}[X] = \frac{1-p}{p},\ \mathbb{V}ar[X] = \frac{1-p}{p^2}.$$

Inverse transform; because $F(k) = 1 - (1-p)^{k+1}$ you can derive

$$\begin{aligned}
& X = k \ \Leftrightarrow\ F(k-1) < U \le F(k) \\
\Leftrightarrow\ & (1-p)^{k+1} \le 1 - U < (1-p)^k \\
\Leftrightarrow\ & (k+1)\log(1-p) \le \log(1-U) < k\log(1-p) \\
\Leftrightarrow\ & k < \frac{\log(1-U)}{\log(1-p)} \le k+1 \\
\Leftrightarrow\ & X = \mathrm{ceil}\Big(\frac{\log(1-U)}{\log(1-p)}\Big) - 1 \overset{\mathcal{D}}{=} \mathrm{ceil}\Big(\frac{\log(U)}{\log(1-p)}\Big) - 1.
\end{aligned}$$

2.5 Negative Binomial $\mathsf{NBin}(p, r)$

X is the number of successes in a sequence of iid Bernoulli trials until the r-th failure. Each trial has success with probability p. Pmf:

$$\mathbb{P}(X = k) = \binom{k+r-1}{k}(1-p)^r p^k, \quad k = 0, 1, \ldots.$$

Equivalently, X is the sum of r iid geometric variables. Thus, $\mathbb{E}[X] = pr/(1-p)$, $\mathbb{V}ar[X] = pr/(1-p)^2$. Simulation:

1. As a sum of r geometric variables.

2. Repetitive Bernoulli trials until r-th failure:

```
def rannegbin(p,r):
    x = 0
    f = 0
    while f<r:
        s = (rng.random()<p)
        if s==True:
            x += 1
        else:
            f += 1
    return x
```

3. As Gamma-Poisson mixture. One can show that $X \overset{\mathcal{D}}{=} \mathsf{Poi}(Y)$, where $Y \overset{\mathcal{D}}{\sim} \mathsf{Gam}(r, (1-p)/p)$ (shape $\alpha = r$ and scale $\lambda = p/(1-p)$). Thus:

```
def rannegbin(p,r):
    lamda = (1-p)/p
    y = rangamma(r,lamda)
    return ranpoisson(y)
```

This algorithm can also be used in case r is not an integer. The next topic is on simulating the Poisson variate. The Gamma variable comes later.

2.6 Poisson $\mathsf{Poi}(\lambda)$

Probabilities and moments are:

$$\mathbb{P}(X=k)=e^{-\lambda}\lambda^k/k!,\ k=0,1,\ldots,\mathbb{E}[X]=\lambda,\mathbb{V}ar[X]=\lambda.$$

Use the fact that X can be interpreted as the number of events in a unit interval of a Poisson process with rate λ. Let $Y_1,Y_2,\ldots$ be the consecutive IID interarrival times of these events; they have an exponential (λ) distribution. Hence

$$X=k \ \Leftrightarrow\ \sum_{j=1}^{k} Y_j \le 1 < \sum_{j=1}^{k+1} Y_j. \tag{1}$$

The algorithm is (see next section for generating an exponential variate):

1: $X=0$;
2: Generate $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0,1)$;
3: $S=-\log(U)/\lambda$;
4: **while** $S<1$ **do**
5: $\quad X=X+1$;
6: $\quad$ Generate $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0,1)$;
7: $\quad S=S-\log(U)/\lambda$;
8: **end while**
9: **return** X.

Notice that this algorithm implements (1) with $Y_j=-\log(U_j)/\lambda$. To avoid the calculation of these logarithms, one sets

$$\sum_{j=1}^{k} Y_j \le 1 < \sum_{j=1}^{k+1} Y_j \ \Leftrightarrow\ -\sum_{j=1}^{k} \log(U_j)/\lambda \le 1 < -\sum_{j=1}^{k+1} \log(U_j)/\lambda$$
$$\Leftrightarrow\ \prod_{j=1}^{k+1} U_j < e^{-\lambda} \le \prod_{j=1}^{k} U_j.$$

Hence the algorithm becomes

1: $X=0$;
2: Generate $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0,1)$;
3: Set $p=U$;
4: **while** $p \ge e^{-\lambda}$ **do**
5: $\quad X=X+1$;
6: $\quad$ Generate $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0,1)$;
7: $\quad p=p*U$;
8: **end while**
9: **return** X.

2.6.1 Python

The distribution is `poisson(k,shape,loc=0)` available in `scipy.stats`, where `shape`$= \lambda$, and which has default support $\{0, 1, \ldots\}$. The location parameter can be used to shift the default support to $\{$`loc`,`loc`$+1, \ldots\}$. For instance, generating $n = 100$ random $\mathsf{Poi}(\lambda)$ for $\lambda = 2.4$ on support $\{0, 1, \ldots\}$:

```
from scipy.stats import poisson

lamda, n = 2.4, 100
x = poisson.rvs(lamda, size=n)
```

3 Special continuous distributions

Definition: (squared) coefficient of variation of nonnegative rv X:

$$c^2[X] = \frac{\mathbb{V}ar[X]}{(\mathbb{E}[X])^2} = \frac{\mathbb{E}[X^2]}{(\mathbb{E}[X])^2} - 1$$

Let be given data $x_1, \ldots, x_n$, and suppose that we fit a distribution. The parameters of the distrbution might be computed by a two-moment fit; or by MLE (maximum likelihood). Two-moment fit means: given the data-average m, data-variance v, and data-squared coefficient of variation c^2, find the parameters of the distribution. For generating, we set $U \stackrel{\mathcal{D}}{=} \text{Uniform}(0, 1)$, which can be generated by the RNG. Conveniently we use $U \stackrel{\mathcal{D}}{=} 1 - U$.

3.1 Uniform $\mathsf{U}(a, b)$

Moments:

$$\mathbb{E}[X] = \frac{a+b}{2},\ \mathbb{V}ar[X] = \frac{(b-a)^2}{12},\ c^2[X] = \frac{1}{3}\left(\frac{b-a}{b+a}\right)^2$$

Two moment fit:

$$a = m(1 - c\sqrt{3}),\ b = m(1 + c\sqrt{3}).$$

Generation via inverse transform: $X = a + (b - a)U$.

3.2 Exponential $\mathsf{Exp}(\lambda)$

Pdf $f(x) = \lambda e^{-\lambda x}$, cdf $F(x) = 1 - e^{-\lambda x}$, $x \geq 0$. Moments:

$$\mathbb{E}[X] = \frac{1}{\lambda},\ \mathbb{V}ar[X] = \frac{1}{\lambda^2},\ c^2[X] = 1.$$

First moment fit (also MLE): $\lambda = 1/m$.
Generation via inverse transform:

$$X = -\frac{1}{\lambda}\log(1 - U) \stackrel{\mathcal{D}}{=} -\frac{1}{\lambda}\log(U).$$

3.2.1 Python

The distribution is `expon(x,loc=0,scale=1)` available in `scipy.stats`, where `scale`$= 1/\lambda$, and which has default support $[0, \infty)$. The location parameter can be used to shift the default support to $[$`loc`$, \infty)$. For instance, generate $n = 100$ random $\mathsf{Exp}(\lambda)$ for $\lambda = 2.4$ on support $[0, \infty)$.

```
from scipy.stats import expon

lamda, n = 2.4, 100
x = expon.rvs(scale = 1.0/lamda, size = n)
```

3.3 Laplace (λ)

PDF with scale parameter $\lambda > 0$,

$$f(x) = \frac{\lambda}{2} e^{-\lambda|x|}, \quad x \in \mathbb{R}.$$

CDF

$$F(x) = \begin{cases} \frac{1}{2} e^{\lambda x} & \text{for } x \leq 0; \\ 1 - \frac{1}{2} e^{-\lambda x} & \text{for } x \geq 0. \end{cases}$$

Moments:

$$\mathbb{E}[X] = 0, \ \mathbb{V}ar[X] = \frac{2}{\lambda^2}.$$

MLE:

$$\frac{1}{\lambda} = \frac{1}{n} \sum_{i=1}^{n} |x_i|.$$

Generation via inverse transform of the $Y \overset{\mathcal{D}}{\sim} \mathsf{Exp}(\lambda)$, and setting randomly $X = Y$ or $X = -Y$:

1: Generate $Y \overset{\mathcal{D}}{\sim} \mathsf{Exp}(\lambda)$;
2: Generate $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0, 1)$;
3: **if** $U < 0.5$ **then**
4: Set $X = Y$;
5: **else**
6: Set $X = -Y$;
7: **end if**
8: **return** X.

3.3.1 Python

The distribution is `laplace(x,loc=0,scale=1)` available in `scipy.stats`, where `scale`$= 1/\lambda$, and with 0 as default symmetric center. The location parameter can be used to shift the center to `loc`. For instance, generating $n = 100$ random Laplace (λ) for $\lambda = 2.4$ with center 0.

```
from scipy.stats import laplace

lamda, n = 2.4, 100
x = laplace.rvs(scale = 1.0/lamda, size = n)
```

3.4 Erlang $\mathsf{Erl}(k, \lambda)$

This is Gamma with an integer shape parameter. Use the fact that X can be interpreted as the sum of k IID exponential (λ) random variables, $X = \sum_{j=1}^{k} Y_j$, with $Y_j \overset{\mathcal{D}}{=} \mathsf{Exp}(\lambda)$ (independent). Moments:

$$\mathbb{E}[X] = \frac{k}{\lambda}, \ \mathbb{V}ar[X] = \frac{k}{\lambda^2}, \ c^2[X] = \frac{1}{k}.$$

Two-moment fit ($c^2 \in \{1/2, 1/3, \ldots\}$):

$$k = \frac{1}{c^2},\ \lambda = \frac{k}{m}.$$

Apply convolution and inverse transform for the exponentials:

$$X \overset{\mathcal{D}}{=} -\frac{1}{\lambda}\sum_{j=1}^{k}\log(U_j) = -\frac{1}{\lambda}\log\Big(\prod_{j=1}^{k}U_j\Big).$$

3.4.1 Python

The distribution is `erlang(x,shape,loc=0,scale=1)` available in `scipy.stats`, where `shape`= k and `scale`= $1/\lambda$, and which has default support $[0, \infty)$. The location parameter can be used to shift the default support to $[\texttt{loc}, \infty)$. For instance, generating $n = 100$ random $\mathsf{Erl}(k, \lambda)$ for $k = 3$ and $\lambda = 2.4$ on support $[0, \infty)$.

```
from scipy.stats import erlang

k, lamda, n = 3, 2.4, 100
x = erlang.rvs(k, scale = 1.0/lamda, size = n)
```

3.5 Hyperexponential $(p, \lambda_1, \lambda_2)$

This is a mixture of two exponentials $F(x) = pG_1(x) + (1-p)G_2(x)$ with $G_j(x)$ the CDF of $\mathsf{Exp}(\lambda_j)$ random variable. Can be generalised to a mixture of n exponentials. Moments:

$$\mathbb{E}[X] = \frac{p}{\lambda_1} + \frac{1-p}{\lambda_2},\ \mathbb{E}[X^2] = \frac{2p}{\lambda_1^2} + \frac{2(1-p)}{\lambda_2^2},\ c^2[X] \geq 1.$$

The parameter fit with balanced means, i.e., $p/\lambda_1 = (1-p)/\lambda_2$:

$$p = \frac{1}{2}\Big(1 + \sqrt{\frac{c^2-1}{c^2+1}}\Big),\ \lambda_1 = \frac{2p}{m},\ \lambda_2 = \frac{2(1-p)}{m}.$$

Apply the composition method for generating.

3.6 Coxian $\mathsf{Cox}(p, \lambda_1, \lambda_2)$

This is a generalisation of Erlang. Let $Y_j \overset{\mathcal{D}}{=} \mathsf{Exp}(\lambda_j)$, $j = 1, 2$ be independent, and $J \in \{1, 2\}$ a discrete random variable with $p = P(J = 2)$ (and $1 - p = P(J = 1)$). Then

$$X = 1\{J = 1\}Y_1 + 1\{J = 2\}(Y_1 + Y_2) = Y_1 + 1\{J = 2\}Y_2.$$

Moments:

$$\mathbb{E}[X] = \frac{1}{\lambda_1} + \frac{p}{\lambda_2},\ \mathbb{E}[X^2] = \frac{2}{\lambda_1^2} + \frac{2p}{\lambda_2}\left(\frac{1}{\lambda_1} + \frac{1}{\lambda_2}\right),\ c^2[X] \geq \frac{1}{2}.$$

The parameter fit with Gamma normalisation, i.e., the third moment is the same as the Gamma distribution with mean m and sq. coef. var. c^2:

$$\lambda_1 = \frac{2}{m}\Big(1 + \sqrt{\frac{c^2-\frac{1}{2}}{c^2+1}}\Big),\ \lambda_2 = \frac{4}{m} - \lambda_1,\ p = \frac{\lambda_2}{\lambda_1}(\lambda_1 m - 1).$$

The generation algorithm is

1: Generate $Y_1 \overset{\mathcal{D}}{\sim} \mathsf{Exp}(\lambda_1)$;
2: Set $Y_2 = 0$;
3: Generate $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0,1)$;
4: **if** $U < p$ **then**
5: Generate $Y_2 \overset{\mathcal{D}}{\sim} \mathsf{Exp}(\lambda_2)$;
6: **end if**
7: **return** $Y_1 + Y_2$.

Can be generalised to include more exponentials and mixtures.

3.7 Gamma $\mathsf{Gam}(\alpha, \lambda)$

The PDF with scale parameter $\lambda > 0$ and shape parameter $\alpha > 0$ is:

$$f(x) = \frac{\lambda^\alpha \, x^{\alpha-1}}{\Gamma(\alpha)} \, e^{-\lambda x}, \quad x \geq 0, \tag{2}$$

where $\Gamma(\cdot)$ is the complete gamma function :

$$\Gamma(\alpha) = \int_0^\infty t^{\alpha-1} e^{-t} \, dt.$$

Moments:

$$\mathbb{E}[X] = \frac{\alpha}{\lambda}, \; \mathbb{V}ar[X] = \frac{\alpha}{\lambda^2}, \; c^2[X] = \frac{1}{\alpha} > 0.$$

A parameter fit is simple:

$$\alpha = \frac{1}{c^2}, \; \lambda = \frac{\alpha}{m}.$$

MLE: α by a root finding of

$$\frac{\Gamma'(\alpha)}{\Gamma(\alpha)} - \log \alpha = \frac{1}{n} \sum_{i=1}^{n} \log x_i - \log m,$$

and $\lambda = \alpha / m$.

By substitution $\lambda x = t$ you get $\int_0^\infty t^{\alpha-1} e^{-t} \, dt = \int_0^\infty \lambda^\alpha \, x^{\alpha-1} e^{-\lambda x} \, dx$. In other words, $\Gamma(\alpha)$ is the normalizing constant in (2).

Generating X is more complicated because there is no inverse transform formula, and because $\alpha < 1$ and $\alpha > 1$ require different methods. The first step is to see that $X \overset{\mathcal{D}}{=} X'/\lambda$, with X' a Gamma $(\alpha, 1)$ distribution. Thus, it suffices to generate X with a Gamma $(\alpha, 1)$ distribution.

- $\alpha > 1$: the acceptance-rejection method with an exponentially (μ) distributed random variable Y. The optimal $\mu = 1/\alpha$ maximizes the acceptance probability. The complete algorithm is

1: **repeat**
2: Generate $Y \overset{\mathcal{D}}{\sim} \mathsf{Exp}(1/\alpha)$;
3: Generate $U \overset{\mathcal{D}}{\sim}$ U(0,1);
4: **until** $\log U < (\alpha - 1)\Big(1 - \frac{Y}{\alpha} + \log \frac{Y}{\alpha}\Big)$.
5: **return** Y/λ.

Observe that

$$Y \overset{\mathcal{D}}{\sim} \mathsf{Exp}(1/\alpha) \;\; \Rightarrow \;\; Y/\alpha \overset{\mathcal{D}}{\sim} \mathsf{Exp}(1).$$

Thus we can speed up the algorithm:

```
1: repeat
2:     Generate Y ~ Exp(1);
3:     Generate U ~ U(0,1);
4: until log U < (α − 1) × (1 − Y + log Y).
5: return αY/λ.
```

Bad for larger α.

- $\alpha > 1$: the ratio-of-uniforms method. Idea: define $h(x) = x^{\alpha-1}\, e^{-x}$, and

$$\Omega = \left\{(v_1, v_2) \,:\, 0 < v_1 \leq \sqrt{h(v_2/v_1)}\right\} \subset (0, \infty) \times (-\infty, \infty).$$

If we can generate (V_1, V_2) uniform on Ω, then $X = V_2/V_1 \overset{\mathcal{D}}{\sim} \mathsf{Gam}(\alpha, 1)$. Uniform (V_1, V_2) are obtained by the acceptance rejection method from a bounding rectangle around Ω. Leading to the algorithm, where

$$a = \left(\frac{\alpha - 1}{e}\right)^{(\alpha-1)/2} \quad b = \left(\frac{\alpha + 1}{e}\right)^{(\alpha+1)/2}.$$

```
1: repeat
2:     Generate U1, U2 ~ U(0,1);
3:     Set V1 = aU1, V2 = bU2;
4:     Set X = V2/V1;
5: until 2 log V1 < (α − 1) log X − X.
6: return X/λ.
```

Also bad for larger α.

- $\alpha > 1$: acceptance rejection with the standard norm. Let $d = \alpha - 1/3$, and $c = 1/\sqrt{9d}$. Define the function

$$h(z) = d(\log y - y + 1), \quad \text{where } \; y = (1 + cy)^3.$$

Then $e^{h(z)} \leq \sqrt{2\pi}\, \phi(z)$. Which leads to,

```
1: repeat
2:     Generate Z ~ N(0, 1);
3:     Generate U ~ U(0,1);
4: until Z > −1/c and log U ≤ h(Z) + ½Z².
5: return d(1 + cZ)³/λ.
```

Very good for large α.

- $\alpha > 1$: decomposition. Write $\alpha = k + \delta$, where $k = \lfloor \alpha \rfloor \in \mathbb{N}$, and $0 < \delta < 1$. Then $X \overset{\mathcal{D}}{=} X_1 + X_2$ where $X_1 \overset{\mathcal{D}}{\sim} \mathsf{Erl}(k, 1)$ and $X_2 \overset{\mathcal{D}}{\sim} \mathsf{Gam}(\delta, 1)$ are independent.

- $0 < \alpha < 1$: use that if $X \overset{\mathcal{D}}{\sim} \mathsf{Gam}(1+\alpha, 1)$ and $U \overset{\mathcal{D}}{\sim} \mathsf{U}(0,1)$ independent, then $XU^{1/\alpha} \overset{\mathcal{D}}{\sim} \mathsf{Gam}(\alpha, 1)$. Thus the algorithm becomes

> 1: Generate $Y \overset{\mathcal{D}}{\sim} \mathsf{Gam}(1+\alpha, 1)$;
> 2: Generate $U \overset{\mathcal{D}}{\sim}$ U(0,1);
> 3: **return** $YU^{1/\alpha}/\lambda$.

- $0 < \alpha < 1$: Ahrens-Dieter algorithm. Acceptance-rejection with proposal pdf

$$g(x) = \begin{cases} \frac{\alpha x^{\alpha-1}}{b}, & 0 < x \leq 1 \\ \frac{\alpha e^{-x}}{b}, & x > 1, \end{cases}$$

where $b = (e+\alpha)/e$. One can show that $f(x) \leq cg(x)$ for all $x > 0$ if $c = b/(\alpha\Gamma(\alpha))$. Generating from $g(x)$ is possible by the inverse transform method. Algorithm

> 1: **repeat**
> 2: Generate $U_1, U_2 \overset{\mathcal{D}}{\sim}$ U(0,1);
> 3: Set $V = bU_1$;
> 4: **if** $V \leq 1$ **then**
> 5: Set $X = V^{1/\alpha}$ and $Y = e^{-X}$;
> 6: **else**
> 7: Set $X = -\log(b - V)/\alpha$ and $Y = X^{\alpha-1}$;
> 8: **end if**
> 9: **until** $U_2 < Y$.
> 10: **return** X/λ.

3.7.1 Python

The distribution is `gamma(x,shape,loc=0,scale=1)` available in `scipy.stats`, where `shape`= α and `scale`= $1/\lambda$, and which has default support $[0, \infty)$. The location parameter can be used to shift the default support to $[\texttt{loc}, \infty)$. For instance, generating $n = 100$ random $\mathsf{Gam}(\alpha, \lambda)$ for $\alpha = 0.6$ and $\lambda = 2.4$ on support $[0, \infty)$.

```
from scipy.stats import gamma

alpha, lamda, n = 0.6, 2.4, 100
x = gamma.rvs(alpha, scale = 1.0/lamda, size = n)
```

3.8 Beta $\mathsf{Beta}(\alpha, \beta)$

PDF with shape parameters $\alpha > 0$ and $\beta > 0$ is

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)}, \quad x \in (0, 1),$$

where $B(\cdot)$ is the beta function :

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}.$$

$$\mathbb{E}[X] = \frac{\alpha}{\alpha+\beta}, \ \mathbb{V}ar[X] = \frac{\alpha\beta}{(\alpha+\beta+1)(\alpha+\beta)^2}, \ c^2[X] = \frac{\beta}{\alpha(\alpha+\beta+1)}.$$

If $\alpha \geq 1$ and $\beta \geq 1$, acceptance rejection with a uniform $(0,1)$ density is applicable. Generally, one can apply the property that $X \stackrel{\mathcal{D}}{=} Y_1/(Y_1+Y_2)$ with $Y_1 \stackrel{\mathcal{D}}{=} \mathsf{Gam}(\alpha,1)$ and $Y_2 \stackrel{\mathcal{D}}{=} \mathsf{Gam}(\beta,1)$, independent of Y_1.

3.8.1 Python

The distribution is `beta(x,a,b,loc=0,scale=1)` available in `scipy.stats`, where `a,b` are the shape parameters α, β, and which has default support $(0,1)$. The location and scale parameters can be used to shift and/or scale. The support becomes (`loc`,`loc+scale`). For instance, generating $n = 100$ random $\mathsf{Beta}(\alpha,\beta)$ for $\alpha = 0.6$ and $\beta = 2.4$ on support $(0,1)$.

```
from scipy.stats import beta

alpha, beta, n = 0.6, 2.4, 100
x = beta.rvs(alpha, beta, size = n)
```

3.9 Weibull $\mathsf{Wei}(\alpha,\lambda)$

PDF with scale parameter $\lambda > 0$ and shape parameter $\alpha > 0$ is,

$$f(x) = \alpha\lambda(\lambda x)^{\alpha-1}e^{-(\lambda x)^{\alpha}}, \quad x \geq 0.$$

Distribution function:

$$F(x) = 1 - e^{-(\lambda x)^{\alpha}}, \quad x \geq 0.$$

Moments:

$$\mathbb{E}[X] = \frac{1}{\lambda}\Gamma(1+(1/\alpha)) \tag{3}$$

$$\mathbb{V}ar[X] = \frac{1}{\lambda^2}\left(\Gamma(1+(2/\alpha)) - \Gamma^2(1+(1/\alpha))\right) \tag{4}$$

$$c^2[X] = \frac{\Gamma(1+(2/\alpha))}{\Gamma^2(1+(1/\alpha))} - 1 > 0. \tag{5}$$

Parameter fit is not possible with closed formulae, it must be done numerically; rewrite (5) and take logarithms:

$$\log\Gamma(1+(2/\alpha)) - 2\log\Gamma(1+(1/\alpha)) = \log(c^2+1). \tag{6}$$

The lefthand side is a decreasing positive function $H(\alpha)$ with $H(\alpha) \uparrow \infty$ as $\alpha \downarrow 0$, and $H(\alpha) \downarrow 0$ as $\alpha \to \infty$. Thus bisection easily solves (6) for α (for any $c^2 > 0$). Substitute the solution in (3):

$$\lambda = \frac{\Gamma(1+(1/\alpha))}{m}.$$

MLE: also a numerical root finding for α by

$$\frac{1}{\alpha} + \frac{1}{n}\sum_{i=1}^{n}\log x_i - \frac{\sum x_i^{\alpha}\log(x_i)}{\sum_{i=1}^{n} x_i^{\alpha}} = 0.$$

And,

$$\lambda = \left(\frac{n}{\sum x_i^{\alpha}}\right)^{1/\alpha}$$

Generation via inverse transform is simple: using $U \stackrel{\mathcal{D}}{=} 1-U$,

$$X = \frac{1}{\lambda}\big(-\log U\big)^{1/\alpha}.$$

3.9.1 Python

The distribution is `weibull_min(x,shape,loc=0,scale=1)` available in `scipy.stats`, where `shape`= α and `scale`= $1/\lambda$, and which has default support $(0, \infty)$. The location parameter can be used to shift the default support to $[\texttt{loc}, \infty)$. For instance, generating $n = 100$ random $\mathsf{Wei}(\alpha, \lambda)$ for $\alpha = 0.6$ and $\lambda = 2.4$ on support $(0, \infty)$.

```
from scipy.stats import weibull_min

alpha, lamda, n = 0.6, 2.4, 100
x = weibull_min.rvs(alpha, scale = 1.0/lamda, size = n)
```

3.10 Normal $\mathsf{N}(\mu, \sigma^2)$

The pdf is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}, \; x \in \mathbb{R}.$$

MLE:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i; \; \sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2.$$

Recall that $X = \mu + \sigma Z$ with $Z \stackrel{\mathcal{D}}{=}$ Normal $(0, 1)$. Thus it suffices to generate a standard normal. There are many algorithms to do this.

- Box-Muller transformations: if U_1 and U_2 are IID uniform $(0, 1)$, then

  $$Z_1 = \sqrt{-2\log(U_1)}\,\cos(2\pi U_2), \;\text{ and }\; Z_2 = \sqrt{-2\log(U_1)}\,\sin(2\pi U_2)$$

  are two IID standard normals.

- The polar method improves this (in speed):

  1: **repeat**
  2: Generate $V_1 \stackrel{\mathcal{D}}{\sim} \mathsf{U}(-1, 1)$;
  3: Generate $V_2 \stackrel{\mathcal{D}}{\sim} \mathsf{U}(-1, 1)$;
  4: **until** $V_1^2 + V_2^2 < 1$.
  5: $S = V_1^2 + V_2^2$;
  6: **return** $V_1\sqrt{(-2\log S)/S}$.

- Ratio-of-uniforms method. The same idea asd explained in the Gamma section. For the bounding rectangle we can take $a = 1$, and $b_- = -\sqrt{2/e}$ and $b_+ = \sqrt{2/e}$.

  1: **repeat**
  2: Generate $U_1, U_2 \stackrel{\mathcal{D}}{\sim}$ U(0,1);
  3: Set $V_1 = U_1$, $V_2 = b_- + (b_+ - b_-)U_2 = (2U_2 - 1)\sqrt{2/e}$;
  4: Set $X = V_2/V_1$;
  5: **until** $X^2 \le -4\log V_1$.
  6: **return** X.

3.10.1 Python

The distribution is `norm(x,loc=0,scale=1)` available in `scipy.stats`, where `loc`$= \mu$ and `scale`$= \sigma$. For instance, generating $n = 100$ random $\mathsf{N}(\mu, \sigma^2)$ for $\mu = 0.6$ and $\sigma^2 = 2.4$.

```
from scipy.stats import norm
import numpy as np

mu, sigma2, n = 0.6, 2.4, 100
x = norm.rvs(mu, scale = np.sqrt(sigma2), size = n)
```

3.11 Lognormal $\mathsf{LN}(\mu, \sigma^2)$

PDf with parameters $\sigma > 0$ and $\mu \in \mathbb{R}$ is,

$$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\log(x)-\mu)^2/\sigma^2}, \quad x > 0.$$

Moments

$$\mathbb{E}[X] = e^{\mu + \frac{1}{2}\sigma^2}, \ \mathbb{V}ar[X] = \left(e^{\sigma^2} - 1\right) e^{2\mu + \sigma^2}, \ c^2[X] = e^{\sigma^2} - 1 > 0.$$

Parameter fit:

$$\sigma^2 = \log(c^2 + 1), \ \mu = \log m - \tfrac{1}{2}\sigma^2.$$

MLE:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} \log x_i; \quad \sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (\log x_i - \mu)^2 = \frac{1}{n}\sum_{i=1}^{n} (\log x_i)^2 - \mu^2.$$

Since $\log(X) \overset{\mathcal{D}}{=} \mathrm{N}(\mu, \sigma^2)$, generating X follows by

1: Generate $Z \overset{\mathcal{D}}{\sim} \mathsf{N}(0, 1)$;
2: **return** $e^{\mu+\sigma Z}$.

3.11.1 Python

The distribution is `lognorm(x,shape,loc=0,scale=1)` available in `scipy.stats`, and where `shape`$= \sigma$, and `scale`$= e^{\mu}$. Actually, the latter is the median of the distribution. The location parameter can be used to shift the default support $(0, \infty)$ to $(\texttt{loc}, \infty)$. For instance, generating $n = 100$ random $\mathsf{LN}(\mu, \sigma^2)$ for $\mu = 0.6$ and $\sigma^2 = 2.4$ on support $(0, \infty)$.

```
from scipy.stats import lognorm
import numpy as np

mu, sigma2, n = 0.6, 2.4, 100
x = lognorm.rvs(np.sqrt(sigma2), loc = 0,
                scale = np.exp(mu),  size = n)
```

3.12 Pareto $\mathsf{Par}(\alpha, \lambda)$

PDF with scale parameter $\lambda > 0$ and shape parameter $\alpha > 0$ is,

$$f(x) = \alpha\lambda(1 + \lambda x)^{-(\alpha+1)}, \quad x \geq 0.$$

Distribution function

$$F(x) = 1 - (1 + \lambda x)^{-\alpha}.$$

Moments (assume $\alpha > 2$):

$$\mathbb{E}[X] = \frac{1}{\lambda(\alpha - 1)}, \ \mathbb{V}ar[X] = \frac{\alpha}{\lambda^2(\alpha - 1)^2(\alpha - 2)}, \ c^2[X] = \frac{\alpha}{\alpha - 2} > 1.$$

Parameter fit:

$$\alpha = \frac{2c^2}{c^2 - 1}, \ \lambda = \frac{1}{(\alpha - 1)m}.$$

Furthermore it is easy to see that $\alpha \leq 1 \ \Rightarrow \ \mathbb{E}[X] = \infty$, and $\alpha \leq 2 \ \Rightarrow \ \mathbb{V}ar[X] = \infty$. For generating Pareto variates you can apply inverse transform:

$$F(X) = U \ \Leftrightarrow \ X = \frac{1}{\lambda}\Big((1 - U)^{-1/\alpha} - 1\Big).$$

Use $U \overset{\mathcal{D}}{=} 1 - U$ for generating Pareto by $X = (U^{-1/\alpha} - 1)/\lambda$.

3.12.1 Python

The distribution is `pareto(x,shape,loc=0,scale=1)` available in `scipy.stats`, where `shape`= α, and `scale`= $1/\lambda$, and which has default support $[1, \infty)$. The location parameter can be used to shift the default support to $[$`loc`+`scale`$, \infty)$. For instance, generating $n = 100$ random $\mathsf{Par}(\alpha, \lambda)$ for $\alpha = 3.6$ and $\lambda = 2.4$ on support $[0, \infty)$.

```
from scipy.stats import pareto

alpha, lamda, n = 3.6, 2.4, 100
x = pareto.rvs(alpha, loc = -1.0/lamda, scale = 1.0/lamda, size = n)
```

4 Multivariate Normal

A d-dimensional normal r.v. $\boldsymbol{X} \overset{\mathcal{D}}{\sim} \mathsf{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \in \mathbb{R}^d$ is given by its mean vector $\boldsymbol{\mu} \in \mathbb{R}^d$ and its variance-covariance matrix $\boldsymbol{\Sigma} \in \mathbb{R}^d \times \mathbb{R}^d$, which is a positive semi-definite symmetric matrix. The pdf is

$$f(\boldsymbol{x}) = \frac{1}{(\sqrt{2\pi})^d \sqrt{\det(\boldsymbol{\Sigma})}} \exp\left(-\tfrac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{x} - \boldsymbol{\mu})\right), \quad \boldsymbol{x} \in \mathbb{R}^d.$$

The marginals are normal: $X_i \overset{\mathcal{D}}{\sim} \mathsf{N}(\mu_i, \sigma_i^2)$, where μ_i is the i-th entry of the $\boldsymbol{\mu}$ vector, and σ_i^2 is the i-th diagonal element $\boldsymbol{\Sigma}_{ii}$. The off-diagonal elements $\boldsymbol{\Sigma}_{ij} = \mathbb{C}ov[X_i, X_j]$, and the correlations are

$$\rho_{ij} = \frac{\boldsymbol{\Sigma}_{ij}}{\sigma_i \sigma_j}.$$

4.1 Cholesky Factorization

From the linear transformation rules we know that for any matrix $\boldsymbol{A} \in \mathbb{R}^d \to \mathbb{R}^d$:

$$\boldsymbol{Z} \overset{\mathcal{D}}{\sim} \mathsf{N}(\boldsymbol{0}, \boldsymbol{I}) \text{ and } \boldsymbol{X} = \boldsymbol{\mu} + \boldsymbol{A}\boldsymbol{Z} \;\Rightarrow\; \boldsymbol{X} \overset{\mathcal{D}}{\sim} \mathsf{N}(\boldsymbol{\mu}, \boldsymbol{A}\boldsymbol{A}^T).$$

Thus, given $\boldsymbol{X} \overset{\mathcal{D}}{\sim} \mathsf{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, we find a matrix $\boldsymbol{A}$ such that $\boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{\Sigma}$. For instance by Cholesky factorization.

1: Compute Cholesky factorization $\boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{\Sigma}$;
2: Generate $Z_1, \ldots, Z_d \overset{\mathcal{D}}{\sim} \mathsf{N}(0,1)$ independently;
3: Set $\boldsymbol{Z} = (Z_1, \ldots, Z_d)^T$ (column vector);
4: **return** $\boldsymbol{X} = \boldsymbol{\mu} + \boldsymbol{A}\boldsymbol{Z}$.

4.2 Python

Generate $n = 100$ multivariate normal variables $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n \in \mathbb{R}^3$ given mean vector $\boldsymbol{\mu} = (2.6, -0.4, -3.6)$ and correlation matrix

$$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & 0.5 & -0.6 \\ 0.5 & 1 & 0.3 \\ -0.6 & 0.3 & 1 \end{pmatrix}$$

A check gave all positive eigenvalues.

```
import numpy as np
import numpy.linalg as la
from scipy.stats import norm

mu = np.array([2.6,-0.4,-3.6]).reshape(3,1)
Sigma = np.array([[1,0.5,-0.6],[0.5,1,0.3],[-0.6,0.3,1]])
lam = la.eigvals(Sigma)
if any(lam<0):
    print('error: Sigma is not positive semidefinite')
else:
    A = la.cholesky(Sigma)
    Z = norm.rvs(size = (3,100))
    X = mu + np.dot(A,Z)
```

Or,

```
import numpy as np
import scipy.stats as stats

mu = np.array([2.6,-0.4,-3.6])
Sigma = np.array([[1,0.5,-0.6],[0.5,1,0.3],[-0.6,0.3,1]])
lam = la.eigvals(Sigma)
if any(lam<0):
    print('error: Sigma is not positive semidefinite')
else:
    X = stats.multivariate_normal.rvs(mu,Sigma,size=100)
```

The case of generating correlated standard normals $W_1, \ldots, W_d$, say with correlations $\rho_{ij} = \mathbb{C}or(W_i, W_j) \in [-1, 1]$ $(i \neq j,\ \rho_{ji} = \rho_{ij})$ goes similarly by defining the covariance matrix $\boldsymbol{\Sigma}$ by

$$\boldsymbol{\Sigma}_{ii} = 1; \boldsymbol{\Sigma}_{ij} = \rho_{ij}.$$

Thus, let $\boldsymbol{A}\boldsymbol{A}^T = \boldsymbol{\Sigma}$, and $\boldsymbol{Z} \overset{\mathcal{D}}{\sim} \mathsf{N}(\boldsymbol{0}, \boldsymbol{I})$, then $\boldsymbol{W} = \boldsymbol{A}\boldsymbol{Z}$. In the special case $d = 2$, we get

$$\boldsymbol{\Sigma} = \begin{pmatrix} 1 & \rho \\ \rho & 1, \end{pmatrix} \qquad (\rho \in [-1, 1]),$$

and the Cholesky factorization works out to

$$W_1 = Z_1,\ W_2 = \rho Z_1 + \sqrt{1 - \rho^2}\, Z_2.$$