Automatic random variate generation in Python

Christoph Baumgarten[‡*], Tirth Patel

Abstract—The generation of random variates is an important tool that is required in many applications. Various software programs or packages contain generators for standard distributions like the normal, exponential or Gamma, e.g., the programming language R and the packages SciPy and NumPy in Python. However, it is not uncommon that sampling from new/non-standard distributions is required. Instead of deriving specific generators in such situations, so-called automatic or black-box methods have been developed. These allow the user to generate random variates from fairly large classes of distributions by only specifying some properties of the distributions (e.g. the density and/or cumulative distribution function). In this note, we describe the implementation of such methods from the C library UNU.RAN in the Python package SciPy and provide a brief overview of the functionality.

Index Terms—numerical inversion, generation of random variates

Introduction

The generation of random variates is an important tool that is required in many applications. Various software programs or packages contain generators for standard distributions, e.g., R ([R C21]) and SciPy ([VGO+20]) and NumPy ([HMvdW+20]) in Python. Standard references for these algorithms are the books [Dev86], [Dag88], [Gen03], and [Knu14]. An interested reader will find many references to the vast existing literature in these works. While relying on general methods such as the rejection principle, the algorithms for well-known distributions are often specifically designed for a particular distribution. This is also the case in the module `stats` in SciPy that contains more than 100 distributions and the module `random` in NumPy with more than 30 distributions. However, there are also so-called automatic or black-box methods for sampling from large classes of distributions with a single piece of code. For such algorithms, information about the distribution such as the density, potentially together with its derivative, the cumulative distribution function (CDF), and/or the mode must be provided. See [HLD04] for a comprehensive overview of these methods. Although the development of such methods was originally motivated to generate variates from non-standard distributions, these universal methods have advantages that make their usage attractive even for sampling from standard distributions. We mention some of the important properties (see [LH00], [HLD04], [DHL10]):

- The algorithms can be used to sample from truncated distributions.
- For inversion methods, the structural properties of the underlying uniform random number generator are preserved and the numerical accuracy of the methods can be controlled by a parameter. Therefore, inversion is usually the only method applied for simulations using quasi-Monte Carlo (QMC) methods.
- Depending on the use case, one can choose between a fast setup with slow marginal generation time and vice versa.

The latter point is important depending on the use case: if a large number of samples is required for a given distribution with fixed shape parameters, a slower setup that only has to be run once can be accepted if the marginal generation times are low. If small to moderate samples sizes are required for many different shape parameters, then it is important to have a fast setup. The former situation is referred to as the fixed-parameter case and the latter as the varying parameter case.

Implementations of various methods are available in the C library UNU.RAN ([HL07]) and in the associated R package `Runuran` (https://cran.r-project.org/web/packages/Runuran/index.html, [TL03]). The aim of this note is to introduce the Python implementation in the SciPy package that makes some of the key methods in UNU.RAN available to Python users in SciPy 1.8.0. These general tools can be seen as a complement to the existing specific sampling methods: they might lead to better performance in specific situations compared to the existing generators, e.g., if a very large number of samples are required for a fixed parameter of a distribution or if the implemented sampling method relies on a slow default that is based on numerical inversion of the CDF. For advanced users, they also offer various options that allow to fine-tune the generators (e.g., to control the time needed for the setup step).

Automatic algorithms in SciPy

Many of the automatic algorithms described in [HLD04] and [DHL10] are implemented in the ANSI C library, UNU.RAN (Universal Non-Uniform RANdom variate generators). Our goal was to provide a Python interface to the most important methods from UNU.RAN to generate univariate discrete and continuous non-uniform random variates. The following generators have been implemented in SciPy 1.8.0:

- `TransformedDensityRejection`: Transformed Density Rejection (TDR) ([Hö95], [GW92])
- `NumericalInverseHermite`: Hermite interpolation based INVersion of CDF (HINV) ([HL03])
- `NumericalInversePolynomial`: Polynomial interpolation based INVersion of CDF (PINV) ([DHL10])

* *Corresponding author: christoph.baumgarten@gmail.com*
‡ *Unaffiliated*

- `SimpleRatioUniforms`: Simple Ratio-Of-Uniforms (SROU) ([Ley01], [Ley03])
- `DiscreteGuideTable`: (Discrete) Guide Table method (DGT) ([CA74])
- `DiscreteAliasUrn`: (Discrete) Alias-Urn method (DAU) ([Wal77])

Before describing the implementation in SciPy in Section scipy_impl, we give a short introduction to random variate generation in Section intro_rv_gen.

A very brief introduction to random variate generation

It is well-known that random variates can be generated by inversion of the CDF F of a distribution: if U is a uniform random number on $(0,1)$, $X := F^{-1}(U)$ is distributed according to F. Unfortunately, the inverse CDF can only be expressed in closed form for very few distributions, e.g., the exponential or Cauchy distribution. If this is not the case, one needs to rely on implementations of special functions to compute the inverse CDF for standard distributions like the normal, Gamma or beta distributions or numerical methods for inverting the CDF are required. Such procedures, however, have the disadvantage that they may be slow or inaccurate, and developing fast and robust inversion algorithms such as HINV and PINV is a non-trivial task. HINV relies on Hermite interpolation of the inverse CDF and requires the CDF and PDF as an input. PINV only requires the PDF. The algorithm then computes the CDF via adaptive Gauss-Lobatto integration and an approximation of the inverse CDF using Newton's polynomial interpolation. Note that an approximation of the inverse CDF can be achieved by interpolating the points $(F(x_i), x_i)$ for points x_i in the domain of F, i.e., no evaluation of the inverse CDF is required.

For discrete distributions, F is a step-function. To compute the inverse CDF $F^{-1}(U)$, the simplest idea would be to apply sequential search: if X takes values $0,1,2,\ldots$ with probabilities $p_0, p_1, p_2, \ldots$, start with $j = 0$ and keep incrementing j until $F(j) = p_0 + \cdots + p_j \geq U$. When the search terminates, $X = j = F^{-1}(U)$. Clearly, this approach is generally very slow and more efficient methods have been developed: if X takes L distinct values, DGT realizes very fast inversion using so-called guide tables / hash tables to find the index j. In contrast DAU is not an inversion method but uses the alias method, i.e., tables are precomputed to write X as an equi-probable mixture of L two-point distributions (the alias values).

The rejection method has been suggested in [VN51]. In its simplest form, assume that f is a bounded density on $[a,b]$, i.e., $f(x) \leq M$ for all $x \in [a,b]$. Sample two independent uniform random variates on U on $[0,1]$ and V on $[a,b]$ until $M \cdot U \leq f(V)$. Note that the accepted points (U,V) are uniformly distributed in the region between the x-axis and the graph of the PDF. Hence, $X := V$ has the desired distribution f. This is a special case of the general version: if f, g are two densities on an interval J such that $f(x) \leq c \cdot g(x)$ for all $x \in J$ and a constant $c \geq 1$, sample U uniformly distributed on $[0,1]$ and X distributed according to g until $c \cdot U \cdot g(X) \leq f(X)$. Then X has the desired distribution f. It can be shown that the expected number of iterations before the acceptance condition is met is equal to c. Hence, the main challenge is to find hat functions g for which c is small and from which random variates can be generated efficiently. TDR solves this problem by applying a transformation T to the density such that $x \mapsto T(f(x))$ is concave. A hat function can then be found by computing tangents at suitable design points. Note that by its nature any rejection method requires not always the same number of uniform variates to generate one non-uniform variate; this makes the use of QMC and of some variance reduction methods more difficult or impossible. On the other hand, rejection is often the fastest choice for the varying parameter case.

The Ratio-Of-Uniforms method (ROU, [KM77]) is another general method that relies on rejection. The underlying principle is that if (U,V) is uniformly distributed on the set $A_f := \{(u,v) : 0 < v \leq \sqrt{f(u/v)}, a < u/v < b\}$ where f is a PDF with support (a,b), then $X := U/V$ follows a distribution according to f. In general, it is not possible to sample uniform values on A_f directly. However, if $A_f \subset R := [u_-, u_+] \times [0, v_+]$ for finite constants u_-, u_+, v_+, one can apply the rejection method: generate uniform values (U,V) on the bounding rectangle R until $(U,V) \in A_f$ and return $X = U/V$. Automatic methods relying on the ROU method such as SROU and automatic ROU ([Ley00]) need a setup step to find a suitable region $S \in \mathbb{R}^2$ such that $A_f \subset S$ and such that one can generate (U,V) uniformly on S efficiently.

Description of the SciPy interface

SciPy provides an object-oriented API to UNU.RAN's methods. To initialize a generator, two steps are required:

1) creating a distribution class and object,
2) initializing the generator itself.

In step 1, a distributions object must be created that implements required methods (e.g., `pdf`, `cdf`). This can either be a custom object or a distribution object from the classes `rv_continuous` or `rv_discrete` in SciPy. Once the generator is initialized from the distribution object, it provides a `rvs` method to sample random variates from the given distribution. It also provides a `ppf` method that approximates the inverse CDF if the initialized generator uses an inversion method. The following example illustrates how to initialize the `NumericalInversePolynomial` (PINV) generator for the standard normal distribution:

```
import numpy as np
from scipy.stats import sampling
from math import exp

# create a distribution class with implementation
# of the PDF. Note that the normalization constant
# is not required
class StandardNormal:
    def pdf(self, x):
        return exp(-0.5 * x**2)

# create a distribution object and initialize the
# generator
dist = StandardNormal()
rng = sampling.NumericalInversePolynomial(dist)

# sample 100,000 random variates from the given
# distribution
rvs = rng.rvs(100000)
```

As `NumericalInversePolynomial` generator uses an inversion method, it also provides a `ppf` method that approximates the inverse CDF:

```
# evaluate the approximate PPF at a few points
ppf = rng.ppf([0.1, 0.5, 0.9])
```

It is also easy to sample from a truncated distribution by passing a `domain` argument to the constructor of the generator. For example, to sample from truncated normal distribution:

```
# truncate the distribution by passing a
# `domain` argument
rng = sampling.NumericalInversePolynomial(
   dist, domain=(-1, 1)
)
```

While the default options of the generators should work well in many situations, we point out that there are various parameters that the user can modify, e.g., to provide further information about the distribution (such as `mode` or `center`) or to control the numerical accuracy of the approximated PPF. (`u_resolution`). Details can be found in the SciPy documentation https://docs.scipy.org/doc/scipy/reference/. The above code can easily be generalized to sample from parametrized distributions using instance attributes in the distribution class. For example, to sample from the gamma distribution with shape parameter `alpha`, we can create the distribution class with parameters as instance attributes:

```
class Gamma:
    def __init__(self, alpha):
        self.alpha = alpha

    def pdf(self, x):
        return x**(self.alpha-1) * exp(-x)

    def support(self):
        return 0, np.inf

# initialize a distribution object with varying
# parameters
dist1 = Gamma(2)
dist2 = Gamma(3)

# initialize a generator for each distribution
rng1 = sampling.NumericalInversePolynomial(dist1)
rng2 = sampling.NumericalInversePolynomial(dist2)
```

In the above example, the `support` method is used to set the domain of the distribution. This can alternatively be done by passing a `domain` parameter to the constructor.

In addition to continuous distribution, two UNU.RAN methods have been added in SciPy to sample from discrete distributions. In this case, the distribution can be either be represented using a probability vector (which is passed to the constructor as a Python list or NumPy array) or a Python object with the implementation of the probability mass function. In the latter case, a finite domain must be passed to the constructor or the object should implement the `support` method[1].

```
# Probability vector to represent a discrete
# distribution. Note that the probability vector
# need not be vectorized
pv = [0.1, 9.0, 2.9, 3.4, 0.3]

# PCG64 uniform RNG with seed 123
urng = np.random.default_rng(123)
rng = sampling.DiscreteAliasUrn(
   pv, random_state=urng
)

# sample from the given discrete distribution
rvs = rng.rvs(100000)
```

Underlying uniform pseudo-random number generators

NumPy provides several generators for uniform pseudo-random numbers[2]. It is highly recommended to use NumPy's default random number generator `np.random.PCG64` for better speed and performance, see [O'N14] and https://numpy.org/doc/stable/reference/random/bit_generators/index.html. To change the uniform random number generator, a `random_state` parameter can be passed as shown in the example below:

```
# 64-bit PCG random number generator in NumPy
urng = np.random.Generator(np.random.PCG64())
# The above line can also be replaced by:
# ``urng = np.random.default_rng()``
# as PCG64 is the default generator starting
# from NumPy 1.19.0

# change the uniform random number generator by
# passing the `random_state` argument
rng = sampling.NumericalInversePolynomial(
   dist, random_state=urng
)
```

We also point out that the PPF of inversion methods can be applied to sequences of quasi-random numbers. SciPy provides different sequences in its QMC module (`scipy.stats.qmc`).

`NumericalInverseHermite` provides a `qrvs` method which generates random variates using QMC methods present in SciPy (`scipy.stats.qmc`) as uniform random number generators[3]. The next example illustrates how to use `qrvs` with a generator created directly from a SciPy distribution object.

```
from scipy import stats
from scipy.stats import qmc

# 1D Halton sequence generator.
qrng = qmc.Halton(d=1)

rng = sampling.NumericalInverseHermite(stats.norm())

# generate quasi random numbers using the Halton
# sequence as uniform variates
qrvs = rng.qrvs(size=100, qmc_engine=qrng)
```

Benchmarking

To analyze the performance of the implementation, we tested the methods applied to several standard distributions against the generators in NumPy and the original UNU.RAN C library. In addition, we selected one non-standard distribution to demonstrate that substantial reductions in the runtime can be achieved compared to other implementations. All the benchmarks were carried out using NumPy 1.22.4 and SciPy 1.8.1 running in a single core on Ubuntu 20.04.3 LTS with Intel(R) Core(TM) i7-8750H CPU (2.20GHz clock speed, 16GB RAM). We run the benchmarks with NumPy's MT19937 (Mersenne Twister) and PCG64 random number generators (`np.random.MT19937` and `np.random.PCG64`) in Python and use NumPy's C implementation of MT19937 in the UNU.RAN C benchmarks. As explained above, the use of PCG64 is recommended, and MT19937 is only included to compare the speed of the Python implementation and the C library by relying on the same uniform number generator (i.e., differences in the performance of the uniform number generation are not taken into account). The code for all the benchmarks can be found on https://github.com/tirthasheshpatel/unuran_benchmarks.

The methods used in NumPy to generate normal, gamma, and beta random variates are:

- the ziggurat algorithm ([MT00b]) to sample from the standard normal distribution,

1. Support for discrete distributions with infinite domain hasn't been added yet.

2. By default, NumPy's legacy random number generator, MT19937 (`np.random.RandomState()`) is used as the uniform random number generator for consistency with the stats module in SciPy.

3. In SciPy 1.9.0, `qrvs` will be added to `NumericalInversePolynomial`.

- the rejection algorithms in Chapter XII.2.6 in [Dev86] if $\alpha < 1$ and in [MT00a] if $\alpha > 1$ for the Gamma distribution,
- Johnk's algorithm ([Jöh64], Section IX.3.5 in [Dev86]) if $\max\{\alpha, \beta\} \leq 1$, otherwise a ratio of two Gamma variates with shape parameter α and β (see Section IX.4.1 in [Dev86]) for the beta distribution.

Benchmarking against the normal, gamma, and beta distributions

Table 1 compares the performance for the standard normal, Gamma and beta distributions. We recall that the density of the Gamma distribution with shape parameter $a > 0$ is given by $x \in (0,\infty) \mapsto x^{a-1}e^{-x}$ and the density of the beta distribution with shape parameters $\alpha, \beta > 0$ is given by $x \in (0,1) \mapsto \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)}$ where $\Gamma(\cdot)$ and $B(\cdot,\cdot)$ are the Gamma and beta functions. The results are reported in Table 1.

We summarize our main observations:

1) The setup step in Python is substantially slower than in C due to expensive Python callbacks, especially for PINV and HINV. However, the time taken for the setup is low compared to the sampling time if large samples are drawn. Note that as expected, SROU has a very fast setup such that this method is suitable for the varying parameter case.
2) The sampling time in Python is slightly higher than in C for the MT19937 random number generator. If the recommended PCG64 generator is used, the sampling time in Python is slightly lower. The only exception is SROU: due to Python callbacks, the performance is substantially slower than in C. However, as the main advantage of SROU is the fast setup time, the main use case is the varying parameter case (i.e., the method is not supposed to be used to generate large samples).
3) PINV, HINV, and TDR are at most about 2x slower than the specialized NumPy implementation for the normal distribution. For the Gamma and beta distribution, they even perform better for some of the chosen shape parameters. These results underline the strong performance of these black-box approaches even for standard distributions.
4) While the application of PINV requires bounded densities, no issues are encountered for $\alpha = 0.05$ since the unbounded part is cut off by the algorithm. However, the setup can fail for very small values of α.

Benchmarking against a non-standard distribution

We benchmark the performance of PINV to sample from the generalized normal distribution ([Sub23]) whose density is given by $x \in (-\infty,\infty) \mapsto \frac{pe^{-|x|^p}}{2\Gamma(1/p)}$ against the method proposed in [NP09] and against the implementation in SciPy's `gennorm` distribution. The approach in [NP09] relies on transforming Gamma variates to the generalized normal distribution whereas SciPy relies on computing the inverse of CDF of the Gamma distribution (https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.gammainccinv.html). The results for different values of p are shown in Table 2.

PINV is usually about twice as fast than the specialized method and about 15-150 times faster than SciPy's implementation[4]. We also found an R package `pgnorm` (https://cran.r-project.org/web/packages/pgnorm/) that implements various approaches from [KR13]. In that case, PINV is usually about 70-200 times faster. This clearly shows the benefit of using a black-box algorithm.

Conclusion

The interface to UNU.RAN in SciPy provides easy access to different algorithms for non-uniform variate generation for large classes of univariate continuous and discrete distributions. We have shown that the methods are easy to use and that the algorithms perform very well both for standard and non-standard distributions. A comprehensive documentation suite, a tutorial and many examples are available at https://docs.scipy.org/doc/scipy/reference/stats.sampling.html and https://docs.scipy.org/doc/scipy/tutorial/stats/sampling.html. Various methods have been implemented in SciPy, and if specific use cases require additional functionality from UNU.RAN, the methods can easily be added to SciPy given the flexible framework that has been developed. Another area of further development is to better integrate SciPy's QMC generators for the inversion methods.

Finally, we point out that other sampling methods like Markov Chain Monte Carlo and copula methods are not part of SciPy. Relevant Python packages in that context are PyMC ([PHF10]), PyStan relying on Stan ([Tea21]), Copulas (https://sdv.dev/Copulas/) and PyCopula (https://blent-ai.github.io/pycopula/).

Acknowledgments

The authors wish to thank Wolfgang Hörmann and Josef Leydold for agreeing to publish the library under a BSD license and for helpful feedback on the implementation and this note. In addition, we thank Ralf Gommers, Matt Haberland, Nicholas McKibben, Pamphile Roy, and Kai Striega for their code contributions, reviews, and helpful suggestions. The second author was supported by the Google Summer of Code 2021 program[5].

References

[CA74] Hui-Chuan Chen and Yoshinori Asau. On generating random variates from an empirical distribution. *AIIE Transactions*, 6(2):163–166, 1974. doi:10.1080/05695557408974949.

[Dag88] John Dagpunar. *Principles of random variate generation*. Oxford University Press, USA, 1988.

[Dev86] Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, 1986. doi:10.1007/978-1-4613-8643-8.

[DHL10] Gerhard Derflinger, Wolfgang Hörmann, and Josef Leydold. Random variate generation by numerical inversion when only the density is known. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 20(4):1–25, 2010. doi:10.1145/1842722.1842723.

[Gen03] James E Gentle. *Random number generation and Monte Carlo methods*, volume 381. Springer, 2003. doi:10.1007/b97336.

[GW92] Walter R Gilks and Pascal Wild. Adaptive rejection sampling for Gibbs sampling. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 41(2):337–348, 1992. doi:10.2307/2347565.

[Hö95] Wolfgang Hörmann. A rejection technique for sampling from T-concave distributions. *ACM Trans. Math. Softw.*, 21(2):182–193, 1995. doi:10.1145/203082.203089.

[HL03] Wolfgang Hörmann and Josef Leydold. Continuous random variate generation by fast numerical inversion. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 13(4):347–362, 2003. doi:10.1145/945511.945517.

4. In SciPy 1.9.0, the speed will be improved by implementing the method from [NP09]

5. https://summerofcode.withgoogle.com/projects/#5912428874825728

Distribution	Method	Python			C	
		Setup	Sampling (PCG64)	Sampling (MT19937)	Setup	Sampling (MT19937)
Standard normal	PINV	4.6	29.6	36.5	0.27	32.4
	HINV	2.5	33.7	40.9	0.38	36.8
	TDR	0.2	37.3	47.8	0.02	41.4
	SROU	8.7 µs	2510	2160	0.5 µs	232
	NumPy	-	17.6	22.4	-	-
Gamma(0.05)	PINV	196.0	29.8	37.2	37.9	32.5
	HINV	24.5	36.1	43.8	1.9	40.7
	NumPy	-	55.0	68.1	-	-
Gamma(0.5)	PINV	16.5	31.2	38.6	2.0	34.5
	HINV	4.9	34.2	41.7	0.6	37.9
	NumPy	-	86.4	99.2	-	-
Gamma(3.0)	PINV	5.3	30.8	38.7	0.5	34.6
	HINV	5.3	33	40.6	0.4	36.8
	TDR	0.2	38.8	49.6	0.03	44
	NumPy	-	36.5	47.1	-	-
Beta(0.5, 0.5)	PINV	21.4	33.1	39.9	2.4	37.3
	HINV	2.1	38.4	45.3	0.2	42
	NumPy	-	101	112	-	-
Beta(0.5, 1.0)	HINV	0.2	37	44.3	0.01	41.1
	NumPy	-	125	138	-	-
Beta(1.3, 1.2)	PINV	15.7	30.5	37.2	1.7	34.3
	HINV	4.1	33.4	40.8	0.4	37.1
	TDR	0.2	46.8	57.8	0.03	45
	NumPy	-	74.3	97	-	-
Beta(3.0, 2.0)	PINV	9.7	30.2	38.2	0.9	33.8
	HINV	5.8	33.7	41.2	0.4	37.4
	TDR	0.2	42.8	52.8	0.02	44
	NumPy	-	72.6	92.8	-	-

TABLE 1

Average time taken (reported in milliseconds, unless mentioned otherwise) to sample 1 million random variates from the standard normal distribution. The mean is computed over 7 iterations. Standard deviations are not reported as they were very small (less than 1% of the mean in the large majority of cases). Note that not all methods can always be applied, e.g., TDR cannot be applied to the Gamma distribution if $a < 1$ since the PDF is not log-concave in that case. As NumPy uses rejection algorithms with precomputed constants, no setup time is reported.

p	0.25	0.45	0.75	1	1.5	2	5	8
Nardon and Pianca (2009)	100	101	101	45	148	120	128	122
SciPy's `gennorm` distribution	832	1000	1110	559	5240	6720	6230	5950
Python (PINV Method, PCG64 urng)	50	47	45	41	40	37	38	38

TABLE 2

Comparing SciPy's implementation and a specialized method against PINV to sample 1 million variates from the generalized normal distribution for different values of the parameter p. Time reported in milliseconds. The mean is computer over 7 iterations.

[HL07] Wolfgang Hörmann and Josef Leydold. UNU.RAN - Universal Non-Uniform RANdom number generators, 2007. https://statmath.wu.ac.at/unuran/doc.html.

[HLD04] Wolfgang Hörmann, Josef Leydold, and Gerhard Derflinger. *Automatic nonuniform random variate generation*. Springer, 2004. doi:10.1007/978-3-662-05946-3.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020. doi:10.1038/s41586-020-2649-2.

[Jöh64] MD Jöhnk. Erzeugung von betaverteilten und gammaverteilten Zufallszahlen. *Metrika*, 8(1):5–15, 1964. doi:10.1007/bf02613706.

[KM77] Albert J Kinderman and John F Monahan. Computer generation of random variables using the ratio of uniform deviates. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):257–260, 1977. doi:10.1145/355744.355750.

[Knu14] Donald E Knuth. *The Art of Computer Programming, Volume 2: Seminumerical algorithms*. Addison-Wesley Professional, 2014. doi:10.2307/2317055.

[KR13] Steve Kalke and W-D Richter. Simulation of the p-generalized Gaussian distribution. *Journal of Statistical Computation and Simulation*, 83(4):641–667, 2013. doi:10.1080/00949655.2011.631187.

[Ley00] Josef Leydold. Automatic sampling with the ratio-of-uniforms method. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):78–98, 2000. doi:10.1145/347837.347863.

[Ley01] Josef Leydold. A simple universal generator for continuous and discrete univariate T-concave distributions. *ACM Transactions on Mathematical Software (TOMS)*, 27(1):66–82, 2001. doi:10.1145/382043.382322.

[Ley03] Josef Leydold. Short universal generators via generalized ratio-of-uniforms method. *Mathematics of Computation*, 72(243):1453–1471, 2003. doi:10.1090/s0025-5718-03-01511-4.

[LH00] Josef Leydold and Wolfgang Hörmann. Universal algorithms as an alternative for generating non-uniform continuous random variates. In *Proceedings of the International Conference on Monte Carlo Simulation 2000.*, pages 177–183, 2000.

[MT00a] George Marsaglia and Wai Wan Tsang. A simple method for generating gamma variables. *ACM Transactions on Mathematical Software (TOMS)*, 26(3):363–372, 2000. doi:10.1145/358407.358414.

[MT00b] George Marsaglia and Wai Wan Tsang. The ziggurat method for generating random variables. *Journal of statistical software*, 5(1):1–7, 2000. doi:10.18637/jss.v005.i08.

[NP09] Martina Nardon and Paolo Pianca. Simulation techniques for generalized Gaussian densities. *Journal of Statistical Computation and Simulation*, 79(11):1317–1329, 2009. doi:10.1080/00949650802290912.

[O'N14] Melissa E. O'Neill. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. Technical Report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA, September 2014.

[PHF10] Anand Patil, David Huard, and Christopher J Fonnesbeck. PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35(4):1, 2010. doi:10.18637/jss.v035.i04.

[R C21] R Core Team. R: A language and environment for statistical computing, 2021. https://www.R-project.org/.

[Sub23] M.T. Subbotin. On the law of frequency of error. *Mat. Sbornik*, 31(2):296–301, 1923.

[Tea21] Stan Development Team. Stan modeling language users guide and reference manual, version 2.28., 2021. https://mc-stan.org.

[TL03] Günter Tirler and Josef Leydold. Automatic non-uniform random variate generation in r. In *Proceedings of DSC*, page 2, 2003.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, pages 1–12, 2020. doi:10.1038/s41592-019-0686-2.

[VN51] John Von Neumann. Various techniques used in connection with random digits. *Appl. Math Ser*, 12(36-38):3, 1951.

[Wal77] Alastair J Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):253–256, 1977. doi:10.1145/355744.355749.