More Random Sampling Methods

This version of the document is dated 2020-12-28.

Peter Occil

1 Contents

1.1 Specific Distributions

Requires random real numbers. This section shows algorithms to sample several popular non-uniform distributions. The algorithms are exact unless otherwise noted, and applications should choose algorithms with either no error (including rounding error) or a user-settable error bound. See the **appendix** for more information.

1.1.1 Normal (Gaussian) Distribution

The ***normal distribution*** (also called the Gaussian distribution) takes the following two parameters:

- `mu` (μ) is the mean (average), or where the peak of the distribution's "bell curve" is.
- `sigma` (σ), the standard deviation, affects how wide the "bell curve" appears. The probability that a normally-distributed random number will be within one standard deviation from the mean is about 68.3%; within two standard deviations (2 times `sigma`), about 95.4%; and within three standard deviations, about 99.7%. (Some publications give σ^2, or variance, rather than standard deviation, as the second parameter. In this case, the standard deviation is the variance's square root.)

There are a number of methods for sampling the normal distribution. An application can combine some or all of these.

1. The ratio-of-uniforms method (given as `NormalRatioOfUniforms` below).
2. In the *Box–Muller transformation*, `mu + radius * cos(angle)` and `mu + radius * sin(angle)`, where `angle = RNDRANGEMaxExc(0, 2 * pi)` and `radius = sqrt(Expo(0.5)) * sigma`, are two independent normally-distributed random numbers. The polar method (given as `NormalPolar` below) likewise produces two independent normal random

numbers at a time.
3. Karney's algorithm to sample from the normal distribution, in a manner that minimizes approximation error and without using floating-point numbers (Karney 2014)**(1)**.

For surveys of Gaussian samplers, see (Thomas et al. 2007)**(2)**, and (Malik and Hemani 2016)**(3)**.

```
METHOD NormalRatioOfUniforms(mu, sigma)
    while true
        a=RNDRANGEMinExc(0,1)
        b=RNDRANGE(0,sqrt(2.0/exp(1.0)))
        if b*b <= -a * a * 4 * ln(a)
          return (RNDINT(1) * 2 - 1) *
            (b * sigma / a) + mu
        end
    end
END METHOD

METHOD NormalPolar(mu, sigma)
  while true
    a = RNDRANGEMinExc(0,1)
    b = RNDRANGEMinExc(0,1)
    if RNDINT(1) == 0: a = 0 - a
    if RNDINT(1) == 0: b = 0 - b
    c = a * a + b * b
    if c != 0 and c <= 1
       c = sqrt(-ln(c) * 2 / c)
       return [a * sigma * c + mu, b * sigma * c + mu]
    end
  end
END METHOD
```

Notes:

1. The *standard normal distribution* is implemented as `Normal(0, 1)`.
2. Methods implementing a variant of the normal distribution, the *discrete Gaussian distribution*, generate *integers* that closely follow the normal distribution. Examples include the one in (Karney 2014)**(1)**, an improved version in (Du et al. 2020)**(4)**, as well as so-called "constant-time" methods such as (Micciancio and Walter 2017)**(5)** that are used above all in *lattice-based cryptography*.
3. The following are some approximations to the normal distribution that papers have suggested:
    - The sum of twelve `RNDRANGEMaxExc(0, sigma)` numbers, subtracted by 6 * `sigma`. (Kabal 2000/2019)**(6)** "warps" this sum in the following way (before adding the mean `mu`) to approximate the normal distribution better: `ssq = sum * sum; sum = ((((0.0000001141*ssq - 0.0000005102) * ssq + 0.00007474) * ssq + 0.0039439) * ssq + 0.98746) * sum`. See also **"Irwin–Hall distribution"**, namely the sum of `n` many `RNDRANGE(0, 1)` numbers, on Wikipedia. D. Thomas (2014)**(7)**, describes a more general approximation called CLT_k, which combines `k` uniform random numbers as follows: `RNDRANGE(0, 1) - RNDRANGE(0, 1) + RNDRANGE(0, 1) - ...`.
    - Numerical **inversions** of the normal distribution's cumulative distribution function (CDF), including those by Wichura, by Acklam, and by Luu (Luu 2016)**(8)**. See also **"A literate program to compute**

the inverse of the normal CDF". Notice that the normal distribution's inverse CDF has no closed form.

1.1.2 Gamma Distribution

The following method generates a random number that follows a *gamma distribution* and is based on Marsaglia and Tsang's method from 2000[(9)] and (Liu et al. 2015)[(10)]. Usually, the number expresses either—

- the lifetime (in days, hours, or other fixed units) of a random component with an average lifetime of `meanLifetime`, or
- a random amount of time (in days, hours, or other fixed units) that passes until as many events as `meanLifetime` happen.

Here, `meanLifetime` must be an integer or noninteger greater than 0, and `scale` is a scaling parameter that is greater than 0, but usually 1 (the random gamma number is multiplied by `scale`).

```
METHOD GammaDist(meanLifetime, scale)
    // Needs to be greater than 0
    if meanLifetime <= 0 or scale <= 0: return error
    // Exponential distribution special case if
    // `meanLifetime` is 1 (see also (Devroye 1986), p. 405)
    if meanLifetime == 1: return Expo(1.0 / scale)
    if meanLifetime < 0.3 // Liu, Martin, Syring 2015
       lamda = (1.0/meanLifetime) - 1
       w = meanLifetime / (1-meanLifetime) * exp(1)
       r = 1.0/(1+w)
       while true
            z = 0
            x = RNDRANGE(0, 1)
            if x <= r: z = -ln(x/r)
            else: z = -Expo(lamda)
            ret = exp(-z/meanLifetime)
            eta = 0
            if z>=0: eta=exp(-z)
            else: eta=w*lamda*exp(lamda*z)
            if RNDRANGE(0, eta) < exp(-ret-z): return ret * scale
       end
    end
    d = meanLifetime
    v = 0
    if meanLifetime < 1: d = d + 1
    d = d - (1.0 / 3) // NOTE: 1.0 / 3 must be a fractional number
    c = 1.0 / sqrt(9 * d)
    while true
        x = 0
        while true
           x = Normal(0, 1)
           v = c * x + 1;
           v = v * v * v
           if v > 0: break
        end
        u = RNDRANGEMinExc(0,1)
        x2 = x * x
        if u < 1 - (0.0331 * x2 * x2): break
        if ln(u) < (0.5 * x2) + (d * (1 - v + ln(v))): break
    end
    ret = d * v
    if meanLifetime < 1
```

```
        ret = ret * pow(RNDRANGE(0, 1), 1.0 / meanLifetime)
    end
    return ret * scale
END METHOD
```

> **Note:** The following is a useful identity for the gamma distribution: `GammaDist(a) = BetaDist(a, b - a) * GammaDist(b)` (Stuart 1962)**[11]**.

1.1.3 Beta Distribution

The beta distribution is a bounded-domain probability distribution; its two parameters, `a` and `b`, are both greater than 0 and describe the distribution's shape. Depending on `a` and `b`, the shape can be a smooth peak or a smooth valley.

The following method generates a random number that follows a *beta distribution*, in the interval [0, 1).

```
METHOD BetaDist(a, b)
  if b==1 and a==1: return RNDRANGE(0, 1)
  // Min-of-uniform
  if a==1: return 1.0-pow(RNDRANGE(0, 1),1.0/b)
  // Max-of-uniform.  Use only if a is small to
  // avoid accuracy problems, as pointed out
  // by Devroye 1986, p. 675.
  if b==1 and a < 10: return pow(RNDRANGE(0, 1),1.0/a)
  x=GammaDist(a,1)
  return x/(x+GammaDist(b,1))
END METHOD
```

I give an **error-bounded sampler** for the beta distribution (when `a` and `b` are both 1 or greater) in a separate page.

1.1.4 von Mises Distribution

The *von Mises distribution* describes a distribution of circular angles and uses two parameters: `mean` is the mean angle and `kappa` is a shape parameter. The distribution is uniform at `kappa = 0` and approaches a normal distribution with increasing `kappa`.

The algorithm below generates a random number from the von Mises distribution, and is based on the Best–Fisher algorithm from 1979 (as described in (Devroye 1986)**[12]** with errata incorporated).

```
METHOD VonMises(mean, kappa)
    if kappa < 0: return error
    if kappa == 0
        return RNDRANGEMinMaxExc(mean-pi, mean+pi)
    end
    r = 1.0 + sqrt(4 * kappa * kappa + 1)
    rho = (r - sqrt(2 * r)) / (kappa * 2)
    s = (1 + rho * rho) / (2 * rho)
    while true
        u = RNDRANGEMaxExc(-pi, pi)
        v = RNDRANGEMinMaxExc(0, 1)
        z = cos(u)
        w = (1 + s*z) / (s + z)
        y = kappa * (s - w)
        if y*(2 - y) - v >=0 or ln(y / v) + 1 - y >= 0
           if angle<-1: angle=-1
           if angle>1: angle=1
```

```
            // NOTE: Inverse cosine replaced here
            // with `atan2` equivalent
            angle = atan2(sqrt(1-w*w),w)
            if u < 0: angle = -angle
            return mean + angle
         end
    end
END METHOD
```

1.1.5 Stable Distribution

As more and more independent random numbers, generated the same way, are added together, their distribution tends to a ***stable distribution***, which resembles a curve with a single peak, but with generally "fatter" tails than the normal distribution. (Here, the stable distribution means the "alpha-stable distribution".) The pseudocode below uses the Chambers–Mallows–Stuck algorithm. The `Stable` method, implemented below, takes two parameters:

- `alpha` is a stability index in the interval (0, 2].
- `beta` is an asymmetry parameter in the interval [-1, 1]; if `beta` is 0, the curve is symmetric.

```
METHOD Stable(alpha, beta)
    if alpha <=0 or alpha > 2: return error
    if beta < -1 or beta > 1: return error
    halfpi = pi * 0.5
    unif=RNDRANGEMinMaxExc(-halfpi, halfpi)
    c=cos(unif)
    if alpha == 1
       s=sin(unif)
       if beta == 0: return s/c
       expo=Expo(1)
       return 2.0*((unif*beta+halfpi)*s/c -
         beta * ln(halfpi*expo*c/(unif*beta+halfpi)))/pi
    else
       z=-tan(alpha*halfpi)*beta
       ug=unif+atan2(-z, 1)/alpha
       cpow=pow(c, -1.0 / alpha)
       return pow(1.0+z*z, 1.0 / (2*alpha))*
          (sin(alpha*ug)*cpow)*
          pow(cos(unif-alpha*ug)/expo, (1.0 - alpha) / alpha)
    end
END METHOD
```

Methods implementing the strictly geometric stable and general geometric stable distributions are shown below (Kozubowski 2000)**[13]**. Here, `alpha` is in (0, 2], `lamda` is greater than 0, and `tau`'s absolute value is not more than min(1, 2/`alpha` − 1). The result of `GeometricStable` is a symmetric Linnik distribution if `tau = 0`, or a Mittag-Leffler distribution if `tau = 1` and `alpha < 1`.

```
METHOD GeometricStable(alpha, lamda, tau)
   rho = alpha*(1-tau)/2
   sign = -1
   if tau==1 or RNDINT(1)==0 or RNDRANGE(0, 1) < tau
       rho = alpha*(1+tau)/2
       sign = 1
   end
   w = 1
```

```
   if rho != 1
      rho = rho * pi
      cotparam = RNDRANGE(0, rho)
      w = sin(rho)*cos(cotparam)/sin(cotparam)-cos(rho)
   end
   return Expo(1) * sign * pow(lamda*w, 1.0/alpha)
END METHOD

METHOD GeneralGeoStable(alpha, beta, mu, sigma)
   z = Expo(1)
   if alpha == 1: return mu*z+Stable(alpha, beta)*sigma*z+
          sigma*z*beta*2*pi*ln(sigma*z)
   else: return mu*z+
          Stable(alpha, beta)*sigma*pow(z, 1.0/alpha)
END METHOD
```

1.1.6 Multivariate Normal (Multinormal) Distribution

The following pseudocode calculates a random vector (list of numbers) that follows a ***multivariate normal (multinormal) distribution***. The method `MultivariateNormal` takes the following parameters:

- A list, `mu` (μ), which indicates the means to add to the random vector's components. `mu` can be `nothing`, in which case each component will have a mean of zero.
- A list of lists `cov`, that specifies a *covariance matrix* (Σ, a symmetric positive definite N×N matrix, where N is the number of components of the random vector).

```
METHOD Decompose(matrix)
  numrows = size(matrix)
  if size(matrix[0])!=numrows: return error
  // Does a Cholesky decomposition of a matrix
  // assuming it's positive definite and invertible
  ret=NewList()
  for i in 0...numrows
    submat = NewList()
    for j in 0...numrows: AddItem(submat, 0)
    AddItem(ret, submat)
  end
  s1 = sqrt(matrix[0][0])
  if s1==0: return ret // For robustness
  for i in 0...numrows
    ret[0][i]=matrix[0][i]*1.0/s1
  end
  for i in 0...numrows
    msum=0.0
    for j in 0...i: msum = msum + ret[j][i]*ret[j][i]
    sq=matrix[i][i]-msum
    if sq<0: sq=0 // For robustness
    ret[i][i]=math.sqrt(sq)
  end
  for j in 0...numrows
    for i in (j + 1)...numrows
      // For robustness
      if ret[j][j]==0: ret[j][i]=0
      if ret[j][j]!=0
        msum=0
        for k in 0...j: msum = msum + ret[k][i]*ret[k][j]
        ret[j][i]=(matrix[j][i]-msum)*1.0/ret[j][j]
      end
```

```
    end
  end
  return ret
END METHOD

METHOD MultivariateNormal(mu, cov)
  mulen=size(cov)
  if mu != nothing
    mulen = size(mu)
    if mulen!=size(cov): return error
    if mulen!=size(cov[0]): return error
  end
  // NOTE: If multiple random points will
  // be generated using the same covariance
  // matrix, an implementation can consider
  // precalculating the decomposed matrix
  // in advance rather than calculating it here.
  cho=Decompose(cov)
  i=0
  ret=NewList()
  vars=NewList()
  for j in 0...mulen: AddItem(vars, Normal(0, 1))
  while i<mulen
    nv=Normal(0,1)
    msum = 0
    if mu == nothing: msum=mu[i]
    for j in 0...mulen: msum=msum+vars[j]*cho[j][i]
    AddItem(ret, msum)
    i=i+1
  end
  return ret
end
```

Note: The **Python sample code** contains a variant of this method for generating multiple random vectors in one call.

Examples:

1. A **binormal distribution** (two-variable multinormal distribution) can be sampled using the following idiom: `MultivariateNormal([mu1, mu2], [[s1*s1, s1*s2*rho], [rho*s1*s2, s2*s2]])`, where `mu1` and `mu2` are the means of the two normal random numbers, `s1` and `s2` are their standard deviations, and `rho` is a *correlation coefficient* greater than -1 and less than 1 (0 means no correlation).
2. **Log-multinormal distribution**: Generate a multinormal random vector, then apply `exp(n)` to each component `n`.
3. A **Beckmann distribution**: Generate a random binormal vector `vec`, then apply `Norm(vec)` to that vector.
4. A **Rice (Rician) distribution** is a Beckmann distribution in which the binormal random pair is generated with `m1 = m2 = a / sqrt(2)`, `rho = 0`, and `s1 = s2 = b`, where `a` and `b` are the parameters to the Rice distribution.
5. **Rice–Norton distribution**: Generate `vec = MultivariateNormal([v,v,v], [[w,0,0],[0,w,0],[0,0,w]])` (where `v = a/sqrt(m*2)`, `w = b*b/m`, and `a`, `b`, and `m` are the parameters to the Rice–Norton distribution), then apply `Norm(vec)` to that vector.
6. A **standard** **complex normal distribution** is a binormal distribution in which the binormal random pair is generated with `s1 = s2 = sqrt(0.5)` and `mu1 = mu2 = 0` and treated as the real and imaginary parts of a complex number.

7. **Multivariate Linnik distribution**: Generate a multinormal random vector, then multiply each component by `GeometricStable(alpha/2.0, 1, 1)`, where `alpha` is a parameter in (0, 2] (Kozubowski 2000)**(13)**.

1.1.7 Gaussian and Other Copulas

A *copula* is a way to describe the dependence between random numbers.

One example is a *Gaussian copula*; this copula is sampled by sampling from a **multinormal distribution**, then converting the resulting numbers to *dependent* uniform random numbers. In the following pseudocode, which implements a Gaussian copula:

- The parameter `covar` is the covariance matrix for the multinormal distribution.
- `erf(v)` is the **error function** of the number `v` (see the appendix).

```
METHOD GaussianCopula(covar)
   mvn=MultivariateNormal(nothing, covar)
   for i in 0...size(covar)
      // Apply the normal distribution's CDF
      // to get uniform random numbers
      mvn[i] = (erf(mvn[i]/(sqrt(2)*sqrt(covar[i][i])))+1)*0.5
   end
   return mvn
END METHOD
```

Each of the resulting uniform random numbers will be in the interval [0, 1], and each one can be further transformed to any other probability distribution (which is called a *marginal distribution* here) by taking the quantile of that uniform number for that distribution (see "**Inverse Transform Sampling**", and see also (Cario and Nelson 1997) **(14)**.)

Examples:

1. To generate two correlated uniform random numbers with a Gaussian copula, generate `GaussianCopula([[1, rho], [rho, 1]])`, where `rho` is the Pearson correlation coefficient, in the interval [-1, 1]. (Other correlation coefficients besides `rho` exist. For example, for a two-variable Gaussian copula, the **Spearman correlation coefficient** `srho` can be converted to `rho` by `rho = sin(srho * pi / 6) * 2`. Other correlation coefficients, and other measures of dependence between random numbers, are not further discussed in this document.)

2. The following example generates two random numbers that follow a Gaussian copula with exponential marginals (`rho` is the Pearson correlation coefficient, and `rate1` and `rate2` are the rates of the two exponential marginals).

```
METHOD CorrelatedExpo(rho, rate1, rate2)
   copula = GaussianCopula([[1, rho], [rho, 1]])
   // Transform to exponentials using that
   // distribution's quantile function
   return [-log1p(-copula[0]) / rate1,
     -log1p(-copula[1]) / rate2]
END METHOD
```

Note: The Gaussian copula is also known as the *normal-to-anything* method.

Other kinds of copulas describe different kinds of dependence between random numbers. Examples of other copulas are—

- the **Fréchet–Hoeffding upper bound copula** *[x, x, ..., x]* (e.g., `[x, x]`), where `x = RNDRANGE(0, 1)`,
- the **Fréchet–Hoeffding lower bound copula** `[x, 1.0 - x]` where `x = RNDRANGE(0, 1)`,
- the **product copula**, where each number is a separately generated `RNDRANGE(0, 1)` (indicating no dependence between the numbers), and
- the **Archimedean copulas**, described by M. Hofert and M. Mächler (2011)**[15]**.

2 Notes

- [1] Karney, C.F.F., "**Sampling exactly from the normal distribution**", arXiv:1303.6257v2 [physics.comp-ph], 2014.
- [2] Thomas, D., et al., "Gaussian Random Number Generators", *ACM Computing Surveys* 39(4), 2007.
- [3] Malik, J.S., Hemani, A., "Gaussian random number generation: A survey on hardware architectures", *ACM Computing Surveys* 49(3), 2016.
- [4] Yusong Du, Baoying Fan, and Baodian Wei, "**An Improved Exact Sampling Algorithm for the Standard Normal Distribution**", arXiv:2008.03855 [cs.DS], 2020.
- [5] Micciancio, D. and Walter, M., "Gaussian sampling over the integers: Efficient, generic, constant-time", in Annual International Cryptology Conference, August 2017 (pp. 455-485).
- [6] Kabal, P., "Generating Gaussian Pseudo-Random Variates", McGill University, 2000/2019.
- [7] Thomas, D.B., 2014, May. FPGA Gaussian random number generators with guaranteed statistical accuracy. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines* (pp. 149-156).
- [8] Luu, T., "Fast and Accurate Parallel Computation of Quantile Functions for Random Number Generation", Dissertation, University College London, 2016.
- [9] Marsaglia, G., Tsang, W.W., "A simple method for generating gamma variables", *ACM Transactions on Mathematical Software* 26(3), 2000.
- [10] Liu, C., Martin, R., Syring, N., "**Simulating from a gamma distribution with small shape parameter**", arXiv:1302.1884v3 [stat.CO], 2015.
- [11] A. Stuart, "Gamma-distributed products of independent random variables", *Biometrika* 49, 1962.
- [12] Devroye, L., ***Non-Uniform Random Variate Generation***, 1986.
- [13] Tomasz J. Kozubowski, "**Computer simulation of geometric stable distributions**", *Journal of Computational and Applied Mathematics* 116(2), pp. 221-229, 2000. **https://doi.org/10.1016/S0377-0427(99)00318-0**
- [14] Cario, M. C., B. L. Nelson, "Modeling and generating random vectors with arbitrary marginal distributions and correlation matrix", 1997.
- [15] Hofert, M., and Maechler, M. "Nested Archimedean Copulas Meet R: The nacopula Package". *Journal of Statistical Software* 39(9), 2011, pp. 1-20.
- [16] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- [17] Oberhoff, Sebastian, "**Exact Sampling and Prefix Distributions**", *Theses and Dissertations*, University of Wisconsin Milwaukee, 2018.

3 Appendix

3.1 Implementation of `erf`

The pseudocode below shows an approximate implementation of the **error function** `erf`, in case the programming language used doesn't include a built-in version of `erf` (such as

JavaScript at the time of this writing). In the pseudocode, `EPSILON` is a very small number to end the iterative calculation.

```
METHOD erf(v)
    if v==0: return 0
    if v<0: return -erf(-v)
    if v==infinity: return 1
    // NOTE: For Java `double`, the following
    // line can be added:
    // if v>=6: return 1
    i=1
    ret=0
    zp=-(v*v)
    zval=1.0
    den=1.0
    while i < 100
        r=v*zval/den
        den=den+2
        ret=ret+r
        // NOTE: EPSILON can be pow(10,14),
        // for example.
        if abs(r)<EPSILON: break
        if i==1: zval=zp
        else: zval = zval*zp/i
        i = i + 1
    end
    return ret*2/sqrt(pi)
END METHOD
```

3.2 Exact, Error-Bounded, and Approximate Algorithms

There are three kinds of randomization algorithms:

1. An *exact algorithm* is an algorithm that samples from the exact distribution requested, assuming that computers—

    - can store and operate on real numbers (which have unlimited precision), and
    - can generate independent uniform random real numbers

    (Devroye 1986, p. 1-2)**(12)**. However, an exact algorithm implemented on real-life computers can incur error due to the use of fixed precision, such as rounding and cancellations, especially when floating-point numbers are involved. An exact algorithm can achieve a guaranteed bound on accuracy (and thus be an *error-bounded algorithm*) using either arbitrary-precision or interval arithmetic (see also Devroye 1986, p. 2)**(12)**. All methods given on this page are exact unless otherwise noted. Note that the `RNDRANGE` method is exact in theory, but has no required implementation.

2. An *error-bounded algorithm* is a sampling algorithm with the following requirements:

    - If the ideal distribution is discrete (takes on a countable number of values), the algorithm samples exactly from that distribution.
    - If the ideal distribution is continuous, the algorithm samples from a distribution that is close to the ideal within a user-specified error tolerance (see below for details). The algorithm can instead sample a random number only partially, as long as the fully sampled number can be made close to the ideal within any

error tolerance desired.
  - In sampling from a distribution, the algorithm incurs no approximation error not already present in the inputs (except errors needed to round the final result to the user-specified error tolerance).

   Many error-bounded algorithms use random bits as their only source of random numbers. An application should use error-bounded algorithms whenever possible.

3. An *inexact*, *approximate*, or *biased algorithm* is neither exact nor error-bounded; it uses "a mathematical approximation of sorts" to generate a random number that is close to the desired distribution (Devroye 1986, p. 2)**(12)**. An application should use this kind of algorithm only if it's willing to trade accuracy for speed.

Most algorithms on this page, though, are not *error-bounded*, but even so, they may still be useful to an application willing to trade accuracy for speed.

There are many ways to describe closeness between two distributions. One suggestion by Devroye and Gravel (2020)**(16)** is Wasserstein distance (or "earth-mover distance"). Here, an algorithm has accuracy ε (the user-specified error tolerance) if it samples random numbers whose distribution is close to the ideal distribution by a Wasserstein distance of not more than ε.

> **Examples:**
>
> 1. Generating an exponential random number via `-ln(RNDRANGE(0, 1))` is an *exact algorithm* (in theory), but not an *error-bounded* one for common floating-point number formats. The same is true of the Box–Muller transformation.
> 2. Generating an exponential random number using the `ExpoExact` method from the section "**Exponential Distribution**" is an *error-bounded algorithm*. Karney's algorithm for the normal distribution (Karney 2014)**(1)** is also error-bounded because it returns a result that can be made to come close to the normal distribution within any error tolerance desired simply by appending more random digits to the end. See also (Oberhoff 2018)**(17)**.
> 3. Examples of *approximate algorithms* include generating a Gaussian random number via a sum of `RNDRANGE(0, 1)`, or most cases of generating a random integer via modulo reduction (see notes in the section "**RNDINT**").

4 License