More Random Sampling Methods

This version of the document is dated 2020-11-22.

Peter Occil

1 Contents

- **Contents**
  - **Specific Distributions**
    - **Normal (Gaussian) Distribution**
    - **Gamma Distribution**
    - **Beta Distribution**
    - **von Mises Distribution**
    - **Stable Distribution**
    - **Multivariate Normal (Multinormal) Distribution**
    - **Gaussian and Other Copulas**
- **Notes**
- **Appendix**
  - **Implementation of `erf`**
  - **A Note on Integer Generation Algorithms**
  - **A Note on Weighted Choice Algorithms**
  - **Exact, Error-Bounded, and Approximate Algorithms**
- **License**

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–Müller 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=RNDU01ZeroExc()
        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 = RNDU01ZeroExc()
    b = RNDU01ZeroExc()
    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 `RNDU01()` numbers, on Wikipedia. D. Thomas (2014)**(7)**, describes a more general approximation called CLT_k, which combines `k` uniform random numbers as follows: `RNDU01() - RNDU01() + RNDU01() - ...`.
   - 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 = RNDU01()
            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 = RNDU01ZeroExc()
        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(RNDU01(), 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 RNDU01()
  // Min-of-uniform
  if a==1: return 1.0-pow(RNDU01(),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(RNDU01(),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 = RNDU01ZeroOneExc()
        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 RNDU01() < 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 = RNDU01()`,
- the **Fréchet–Hoeffding lower bound copula** `[x, 1.0 - x]` where `x = RNDU01()`,
- the **product copula**, where each number is a separately generated `RNDU01()` (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) Knuth, Donald E. and Andrew Chi-Chih Yao. "The complexity of nonuniform random number generation", in *Algorithms and Complexity: New Directions and Recent Results*, 1976.
- (17) This is because the binary entropy of `p = 1/n` is `p * log2(1/p) = log2(n) / n`, and the sum of `n` binary entropies (for `n` outcomes with probability `1/n` each) is `log2(n)`. Any optimal integer generation algorithm will come within 2 bits of this lower bound on average.
- (18) D. Lemire, "A fast alternative to the modulo reduction", Daniel Lemire's blog, 2016.
- (19) Lemire, D., "**Fast Random Integer Generation in an Interval**", arXiv:1805.10941v4 [cs.DS], 2018.
- (20) Lumbroso, J., "**Optimal Discrete Uniform Generation from Coin Flips, and Applications**", arXiv:1304.1916 [cs.DS]
- (21) "**Probability and Random Numbers**", Feb. 29, 2004.

- [22] Mennucci, A.C.G. "**Bit Recycling for Scaling Random Number Generators**", arXiv:1012.4290 [cs.IT], 2018.
- [23] Devroye, L., Gravel, C., "**Random variate generation using only finitely many unbiased, independently and identically distributed random bits**", arXiv:1502.02539v6 [cs.IT], 2020.
- [24] Saad, F.A., Freer C.E., et al., "**The Fast Loaded Dice Roller: A Near-Optimal Exact Sampler for Discrete Probability Distributions**", arXiv:2003.03830v2 [stat.CO], also in *AISTATS 2020: Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics, Proceedings of Machine Learning Research* 108, Palermo, Sicily, Italy, 2020.
- [25] Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka, "**Optimal Approximate Sampling From Discrete Probability Distributions**", arXiv:2001.04555v1 [cs.DS], also in Proc. ACM Program. Lang. 4, POPL, Article 36 (January 2020), 33 pages.
- [26] Shaddin Dughmi, Jason D. Hartline, Robert Kleinberg, and Rad Niazadeh. 2017. Bernoulli Factories and Black-Box Reductions in Mechanism Design. In *Proceedings of 49th Annual ACM SIGACT Symposium on the Theory of Computing*, Montreal, Canada, June 2017 (STOC'17).
- [27] Morina, G., Łatuszyński, K., et al., "**From the Bernoulli Factory to a Dice Enterprise via Perfect Sampling of Markov Chains**", arXiv:1912.09229v1 [math.PR], 2019.
- [28] K. Bringmann and K. Panagiotou, "Efficient Sampling Methods for Discrete Distributions." In: Proc. 39th International Colloquium on Automata, Languages, and Programming (ICALP'12), 2012.
- [29] A.J. Walker, "An efficient method for generating discrete random variables with general distributions", *ACM Transactions on Mathematical Software* 3, 1977.
- [30] Vose, Michael D. "A linear algorithm for generating random numbers with a given distribution." IEEE Transactions on software engineering 17, no. 9 (1991): 972-975.
- [31] K. Bringmann and K. G. Larsen, "Succinct Sampling from Discrete Distributions", In: Proc. 45th Annual ACM Symposium on Theory of Computing (STOC'13), 2013.
- [32] L. Hübschle-Schneider and P. Sanders, "**Parallel Weighted Random Sampling**", arXiv:1903.00227v2 [cs.DS], 2019.
- [33] Y. Tang, "An Empirical Study of Random Sampling Methods for Changing Discrete Distributions", Master's thesis, University of Alberta, 2019.
- [34] Roy, Sujoy Sinha, Frederik Vercauteren and Ingrid Verbauwhede. "**High Precision Discrete Gaussian Sampling on FPGAs**." *Selected Areas in Cryptography* (2013).
- [35] T. S. Han and M. Hoshi, "Interval algorithm for random number generation", *IEEE Transactions on Information Theory* 43(2), March 1997.
- [36] 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 A Note on Integer Generation Algorithms

There are many algorithms for the `RNDINT(maxInclusive)` method, which generates uniform random integers in [0, maxInclusive]. This section deals with "optimal" `RNDINT` algorithms in terms of the number of random bits they use on average (assuming we have a source of "truly" random bits).

Knuth and Yao (1976)**[16]** showed that any algorithm that uses only unbiased random bits to generate random integers with separate probabilities can be described as a *binary tree* (also known as a *DDG tree* or *discrete distribution generating tree*). Random unbiased bits trace a path in this tree, and each leaf (terminal node) in the tree represents an outcome. In the case of `RNDINT`, there are `n = maxInclusive + 1` outcomes that each occur with probability `1/n`. Knuth and Yao showed that any *optimal* algorithm for `RNDINT`, once described as a DDG tree, needs at least `log2(n)` and at most `log2(n) + 2` bits on average (where `log2(x) = ln(x)/ln(2)`).**[17]**

As also shown by Knuth and Yao, however, any integer generating algorithm that is both optimal *and unbiased (exact)* will also run forever in the worst case, even if it uses few random bits on average. This is because in most cases, `n` will not be a power of 2, so that `n` will have an infinite binary expansion, so that the resulting DDG tree will have to either be infinitely deep, or include "rejection leaves" at the end of the tree. (If `n` is a power of 2, the binary expansion will be finite, so that the DDG tree will have a finite depth and no rejection leaves.)

Because of this, there is no general way to "fix" the worst case of running forever, while still having an unbiased (exact) algorithm. For instance, modulo reductions can be represented by a DDG tree in which rejection leaves are replaced with labeled outcomes, but the bias occurs because only some outcomes can replace rejection leaves this way. Even with rejection sampling, stopping the rejection after a fixed number of iterations will likewise lead to bias, for the same reasons. However, which outcomes are biased this way depends on the algorithm.

The following are some ways to implement `RNDINT`. (The column "Unbiased?" means whether the algorithm generates random integers without bias, even if `n` is not a power of 2.)

Algorithm	Optimal?	Unbiased?	Time Complexity
Rejection sampling: Sample in a bigger range until a sampled number fits the smaller range.	Not always	Yes	Runs forever in worst case

Multiply-and-shift reduction: Generate `bignumber`, a `k`-bit random integer with many more bits than `n` has, then find `(bignumber * n) >> k` (see (Lemire 2016)[18], (Lemire 2018)[19], and the "Integer Multiplication" algorithm surveyed by M. O'Neill).	No	No	Constant
Modulo reduction: Generate `bignumber` as above, then find `rem(bignumber, n)`	No	No	Constant
Fast Dice Roller (Lumbroso 2013)[20]	Yes	Yes	Runs forever in worst case
Math Forum (2004)[21] or (Mennucci 2018)[22] (batching/recycling random bits)	Yes	Yes	Runs forever in worst case
"FP Multiply" surveyed by **M. O'Neill**	No	No	Constant
Algorithm in "Conclusion" section by O'Neill	No	Yes	Runs forever in worst case
"Debiased" and "Bitmask with Rejection" surveyed by M. O'Neill	No	Yes	Runs forever in worst case

There are various techniques that can reduce the number of bits "wasted" by an integer-generating algorithm, and bring that algorithm closer to the theoretical lower bound of `log2(n)` bits per random integer, even if the algorithm isn't "optimal". These techniques, which include batching, bit recycling, and randomness extraction, are discussed, for example, in the Math Forum page and the Lumbroso and Mennucci papers referenced above, and in (Devroye and Gravel 2020, section 2.3)[23].

> **Note:** A similar question is how to generate a random integer given rolls of a fair die; more specifically, how to roll a *k*-sided die given a *p*-sided die. This can't be done without "wasting" randomness, unless "every prime number dividing *k* also divides *p*" (see "**Simulating a dice with a dice**" by B. Kloeckner, 2008). However, since *randomness extraction* (see my **Note on Randomness Extraction**) can turn die rolls into unbiased bits, so that the discussion above applies, this question is interesting only when someone wants to build instructions to choose a number at random by rolling real dice or flipping real coins.
>
> **Example**: As an example of batching, to generate three digits from 0 through 9, we can call `RNDINT(999)` to generate an integer in [0, 999], then break the number it returns into three digits.

3.3 A Note on Weighted Choice Algorithms

Just like integer generation algorithms (see the previous section), weighted choice algorithms (implementations of `WeightedChoice` that sample with replacement)—

- involve generating random integers with separate probabilities, and
- can all be described as a binary DDG tree.

In this case, though, the number of random bits an algorithm uses on average is bounded from below by the sum of binary entropies of all the probabilities involved. For example, say we give the four integers 1, 2, 3, 4 the following weights: 3, 15, 1, 2. The binary entropies of these weights are 0.4010... + 0.3467... + 0.2091... + 0.3230... = 1.2800... (because the sum of the weights is 21 and the binary entropy of 3/21 is `(3/21) * log2(21/3) = 0.4010...`, and so on for the other weights). Thus, any such algorithm will require at least 1.2800... bits on average to generate a random number with these weights.[17] Another difference from integer generation algorithms is that usually a special data structure has to be built for the sampling to work, and often there is a need to make updates to the structure as items are sampled.

The following are some ways to implement `WeightedChoice`. (Some algorithms come close, on average, to the optimal number of random bits, and the notes may show this.) For these samplers to be *error-bounded*:

- Weights passed to these algorithms should first be converted to integers (see `IntegerWeightsListFP` or `NormalizeRatios` in "**Sampling for Discrete Distributions**" for conversion methods), or rational numbers when indicated.
- Floating-point arithmetic and floating-point random number generation (such as `RNDRANGE()`) should be avoided.

Algorithm	Notes
Linear search with cumulative weights	The `WeightedChoice` pseudocode calculates a list of cumulative weights (also known as a *cumulative distribution table* or *CDT*), then generates a random number less than the sum of (original) weights, then does a linear scan of the new list to find the first item whose cumulative weight exceeds the random number.
Fast Loaded Dice Roller (Saad et al., 2020)[24].	Uses integer weights only, and samples using random bits ("fair coins"). This sampler comes within 6 bits, on average, of the optimal number of bits.
Samplers described in (Saad et al., 2020)[25]	Uses integer weights only, and samples using random bits. The samplers come within 2 bits, on average, of the optimal number of bits as long as the sum of the weights is of the form 2^k or $2^k - 2^m$.
Rejection sampling	Given a list (`weights`) of `n` weights: (1) find the highest weight and call it *max*; (2) set *i* to `RNDINT(n - 1)`; (3) With probability `weights[i]/max` (e.g., if `ZeroOrOne(weights[i], max)==1` for integer weights), return *i*, and go to step 2 otherwise. (See, e.g., sec. 4 of the Fast Loaded Dice Roller paper, or the Tang or Klundert papers. `weights[i]` can also be a function that calculates the weight for `i` "on the fly"; in that case, `max` is the maximum value of `weights[i]` for all `i`.) If the weights are instead "coins", each with a separate but unknown bias, the algorithm is also called *Bernoulli race* (Dughmi et al. 2017)[26]; see also (Morina et al., 2019)[27]: (1) set *i* to `RNDINT(n - 1)`; (2) flip coin *i* (the first coin is 0, the second is 1, etc.), then return *i* if it returns 1 or heads, or go to step 1 otherwise.
(Bringmann and Panagiotou 2012)[28].	Shows a sampler designed to work on a sorted list of weights.
Alias method (Walker	Michael Vose's version of the alias method (Vose 1991)[30] is described in "**Darts, Dice, and Coins: Sampling from a Discrete Distribution**". Weights should be rational numbers.

1977)[29]	
(Klundert 2019)[29]	Various data structures, with emphasis on how they can support changes in weights.
The Bringmann–Larsen succinct data structure (Bringmann and Larsen 2013)[31]	Uses rejection sampling if the sum of weights is large, and a compressed structure otherwise.
(Hübschle-Schneider and Sanders 2019)[32].	Parallel weighted random samplers.
(Tang 2019)[33].	Presents various algorithms, including two- and multi-level search, as well as linear search (with original weights), binary search (with cumulative weights), and a new "flat" method.
"Loaded Die from Biased Coins"	Given a list of probabilities `probs` that must sum to 1 and should be rational numbers: (1) Set `cumu` to 1 and `i` to 0; (2) with probability `probs[i]/cumu`, return `i`; (3) subtract `probs[i]` from `cumu`, then add 1 to `i`, then go to step 2. For a correctness proof, see "Darts, Dice, and Coins". If each probability in `probs` is calculated "on the fly", this is also called sequential search; see chapter 10 of (Devroye 1986)[12] (but this generally won't be exact unless all the probabilities involved are rational numbers).
Knuth and Yao (1976)[16]	Generates a DDG tree from the binary expansions of the probabilities. Comes within 2 bits, on average, of the optimal number of bits. This is suggested in exercise 3.4.2 of chapter 15 of (Devroye 1986, p. 1-2)[12], implemented in *randomgen.py* as the `discretegen` method, and also described in (Roy et al. 2013)[34] and (Devroye and Gravel 2020)[23]. `discretegen` can work with probabilities that are irrational numbers (which have infinite binary expansions) as long as there is a way to calculate the binary expansion "on the fly".
(Han and Hoshi 1997)[35]	Uses cumulative probabilities as input. An error-bounded version is described in (Devroye and Gravel 2020)[23] and comes within 3 bits, on average, of the optimal number of bits.

> **Note:** If the source of randomness is a "biased coin" which returns heads with *unknown* probability of heads, and tails otherwise, it can be turned into a "fair" coin (and so output unbiased bits) via *randomness extraction* (see my **Note on Randomness Extraction**), so that the algorithms above can be used.

3.4 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 of any precision, and
- can generate independent uniform random real numbers of any precision

(Devroye 1986, p. 1-2)[(12)]. However, an exact algorithm implemented on real-life computers can incur rounding and other errors, especially errors involving floating-point arithmetic or irrational numbers. 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 `RNDU01` or `RNDRANGE` are exact in theory, but have 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)[(23)] 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(RNDU01())` 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–Müller 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 (an example when the return value has 53 bits after the point is as follows: `for i in 54..100: ret = ret + RNDINT(1) * pow(2,-i)`). See also (Oberhoff 2018)[(36)].
3. Examples of *approximate algorithms* include generating a Gaussian

random number via a sum of `RNDU01()`, or most cases of generating a random integer via modulo reduction (see "**A Note on Integer Generation Algorithms**").

4 License