Random: A C++ Class for Generating Random Number Distributions

Richard Saucier

February 2017

Contents

List of Figures

List of Tables

List of Listings

1 Summary

This report describes **Random**, a C++ class for generating random number distributions, suitable for performing Monte Carlo simulations. This class gives the programmer the ability to generate random number distributions as if they were native types in the C++ language.

There are two broad aspects to this class:

- **Random Number Generator**
  The class provides a number of generators to choose from. These are the engines for generating the pseudorandom numbers. Each engine will deliver 32-bit and 64-bit integers as well as floating point numbers between 0 and 1. Many of the generators also have jump capabilities.

- **Random Number Distribution**
  Independently of whichever generator is selected, the class provides many different distributions to choose from. The class currently contains 27 continuous distributions, 9 discrete distributions, distributions based on empirical data, and bivariate distributions, as well as distributions based on number theory. Moreover, it allows the user–programmer to specify an arbitrary function or procedure to use for generating distributions that are not already in the collection. It is also shown that it is easy to extend the collection to include new distributions.

To generate 1000 normally distributed random numbers (with default mean 0 and standard deviation 1) using the **jkiss** generator, we would write the following code:

```cpp
#include "Random.h"

int main() {

   // create a new generator using jkiss as the engine
   Generator<uint32_t> *rng = new JKISS::jkiss;

   // create a random number distribution object and initialize it to use this generator
   rnd::Random<uint32_t> rnd( rng );

   // output 1000 normally distributed pseudorandom numbers
   for ( int i = 0; i < 10; i++ )
      std::cout << rnd.normal() << std::endl;

   delete rng;

   return 0;
}
```

No libraries are required and there is nothing to build; one merely needs to include the header file **Random.h** in order to make use of the class.

2 Introduction

This report deals with random number distributions, the foundation for performing Monte Carlo simulations. Although Lord Kelvin may have been the first to use Monte Carlo methods in his 1901 study of the Boltzmann equation in statistical mechanics, their widespread use dates back to the development of the atomic bomb in 1944. Monte Carlo methods have been used extensively in the field of nuclear physics for the study of neutron transport and radiation shielding. They remain useful whenever the underlying physical law is either unknown or it is known but one cannot obtain enough detailed information in order to apply it directly in a deterministic manner. In particular, the field of operations research has a long history of employing Monte Carlo simulations. There are several reasons for using simulations, but they basically fall into three categories.

- **To Supplement Theory**
  While the underlying process or physical law may be understood, an analytical solution—or even a solution by numerical methods—may not be available. In addition, even in the cases where we possess a deterministic solution, we may be unable to obtain the initial conditions or other information necessary to apply it.

- **To Supplement Experiment**
  Experiments can be very costly or we may be unable to perform the measurements required for a particular mathematical model.

- **Computing Power has Increased while Cost has Decreased**
  In 1965, when writing an article for *Electronics* magazine, Gordon Moore formulated what has since been named Moore's Law: the number of components that could be squeezed onto a silicon chip would double every year. Moore updated this prediction in 1975 from doubling every year to doubling every two years. These observations proved remarkably accurate; the processing technology of 1996, for example, was some eight million times more powerful than that of 1966 [Helicon Publishing 1999].

In short, computer simulations are viable alternatives to both theory and experiment—and we have every reason to believe they will continue to be so in the future. A reliable source of random numbers, and a means of transforming them into prescribed distributions, is essential for the success of the simulation approach. This report describes various ways to obtain distributions, how to estimate the distribution parameters, descriptions of the distributions, choosing a good uniform random number generator, and some illustrations of how the distributions may be used.

3 Methods for Generating Random Number Distributions

We wish to generate random numbers,[*] x, that belong to some domain, $x \in [x_{\min}, x_{\max}]$, in such a way that the frequency of occurrence, or probability density, will depend upon the value of x in a prescribed functional form $f(x)$. Here, we review several techniques for doing this. We should point out that all of these methods presume that we have a supply of uniformly distributed random numbers in the half-closed unit inteval $[0, 1)$. These methods are only concerned with transforming the uniform random variate on the unit interval into another functional form. The subject of how to generate the underlying uniform random variates is discussed in Appendix A.

We begin with the inverse transformation technique, as it is probably the easiest to understand and is also the method most commonly used. A word on notation: $f(x)$ is used to denote the probability density and $F(x)$ is used to denote the cumulative distribution function.

3.1 Inverse Transformation

If we can invert the cumulative distribution function $F(x)$, then it is a simple matter to generate the probability density function $f(x)$. The algorithm for this technique is as follows:

[*] Of course, all such numbers generated according to precise and specific algorithms on a computer are not truly random at all but only exhibit the appearance of randomness and are therefore best described as "pseudo-random." However, throughout this report, we use the term "random number" as merely a shorthand to signify the more correct term of "pseudo-random number."

1. Generate $U \sim \mathrm{U}(0,1)$.
2. Return $X = F^{-1}(U)$.

It is not difficult to see how this method workswith the aid of Fig. 1.

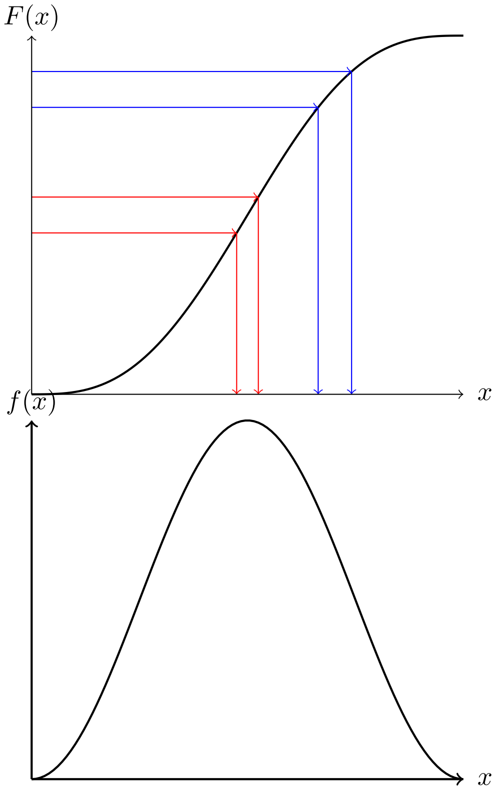

Figure 1. Inverse transform method

We take uniformly distributed samples along the y axis between 0 and 1. Referring to Fig. 1, we see that where the distribution function $F(x)$ is relatively steep (red lines), there will result a high density of points along the x axis (giving a larger value of $f(x)$), and, on the other hand, where $F(x)$ has a relatively shallow slope (blue lines), there will result in a corresponding lower density of points along the x axis (giving a smaller value of $f(x)$). More formally, if

$$x = F^{-1}(y), \tag{1}$$

where $F(x)$ is the indefinite integral $F(x) = \int_{-\infty}^{x} f(t)dt$ of the desired density function $f(x)$, then $y = F(x)$ and

$$\frac{dy}{dx} = f(x). \tag{2}$$

This technique can be illustrated with the Weibull distribution. In this case, we have $F(x) = 1 - e^{-(x/b)^c}$. So, if $U \sim \mathrm{U}(0,1)$ and $U = F(X)$, then we find[*] $X = b[-\ln(1-U)]^{1/c}$.

The inverse transform method is a simple, efficient technique for obtaining the probability density, but it requires that we be able to invert the distribution function. As this is not always feasible, we need to consider other techniques as well.

[*] Since $1-U$ has precisely the same distribution as U, in practice, we use $X = b(-\ln U)^{1/c}$, which saves a subtraction and is therefore slightly more efficient.

3.2 Composition

Composition is a simple extension of the inverse transformation technique. It applies to a situation where the probability density function can be written as a linear combination of simpler composition functions and where each of the composition functions has an indefinite integral that is invertible.[*] Thus, we consider cases where the density function $f(x)$ can be expressed as

$$f(x) = \sum_{i=1}^{n} p_i f_i(x), \tag{3}$$

where

$$\sum_{i=1}^{n} p_i = 1 \tag{4}$$

and each of the f_i has an indefinite integral, $F_i(x)$ with a known inverse. The algorithm is as follows:

1. Select index i with probability p_i.
2. Independently generate $U \sim \mathrm{U}(0,1)$.
3. Return $X = F_i^{-1}(U)$.

For example, consider the density function for the Laplace distribution (also called the double exponential distribution):

$$f(x) = \frac{1}{2b} \exp\left(-\frac{|x-a|}{b}\right). \tag{5}$$

This can also be written as

$$f(x) = \frac{1}{2} f_1(x) + \frac{1}{2} f_2(x), \tag{6}$$

where

$$f_1(x) \equiv \begin{cases} \frac{1}{b}\exp\left(\frac{x-a}{b}\right) & x < a \\ 0 & x \geq a \end{cases} \quad \text{and} \quad f_2(x) \equiv \begin{cases} 0 & x < a \\ \frac{1}{b}\exp\left(-\frac{x-a}{b}\right) & x \geq a \end{cases}. \tag{7}$$

Now each of these has an indefinite integral, namely

$$F_1(x) \equiv \begin{cases} \exp\left(\frac{x-a}{b}\right) & x < a \\ 0 & x \geq a \end{cases} \quad \text{and} \quad F_2(x) \equiv \begin{cases} 0 & x < a \\ 1 - \exp\left(-\frac{x-a}{b}\right) & x \geq a \end{cases} \tag{8}$$

that is invertible. Since $p_1 = p_2 = 1/2$, we can select $U_1 \sim \mathrm{U}(0,1)$ and set

$$i = \begin{cases} 1 & \text{if } U_1 \geq 1/2 \\ 2 & \text{if } U_1 < 1/2 \end{cases} \tag{9}$$

Independently, we select $U_2 \sim \mathrm{U}(0,1)$ and then, using the inversion technique of section 3.1,

$$X = \begin{cases} a + b \ln U_2 & \text{if } i = 1 \\ a - b \ln U_2 & \text{if } i = 2 \end{cases} \tag{10}$$

[*] The composition functions f_i must be defined on disjoint intervals, so that if $f_i(x) > 0$, then $f_j(x) = 0$ for all x whenever $j \neq i$. That is, there is no overlap between the composition functions.

3.3 Convolution

If X and Y are independent random variables from known density functions $f_X(x)$ and $f_Y(y)$, then we can generate new distributions by forming various algebraic combinations of X and Y. Here, we show how this can be done via summation, multiplication, and division. We only treat the case when the distributions are independent—in which case, the joint probability density function is simply $f(x,y) = f_X(x)f_Y(y)$. First consider summation. The cumulative distribution is given by

$$F_{X+Y}(u) = \iint\limits_{x+y \le u} f(x,y)\,\mathrm{d}x\,\mathrm{d}y = \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{u-x} f(x,y)\,\mathrm{d}y \right) \mathrm{d}x. \tag{11}$$

The density is obtained by differentiating with respect to u, and this gives us the convolution formula for the sum

$$f_{X+Y}(u) = \frac{\mathrm{d}}{\mathrm{d}u} F_{X+Y}(u) = \int_{-\infty}^{\infty} f(x, u-x)\,\mathrm{d}x, \tag{12}$$

where we used Leibniz's rule to carry out the differentiation (first on x and then on y). Notice that, if the random variables are nonnegative, then the lower limit of integration can be replaced with zero, since $f_X(x) = 0$ for all $x < 0$, and the upper limit can be replaced with u, since $f_Y(u-x) = 0$ for $x > u$.

Let us apply this formula to the sum of two uniform random variables on $[0,1]$. We have

$$f_{X+Y}(u) = \int_{-\infty}^{\infty} f(x)f(u-x)\,\mathrm{d}x. \tag{13}$$

Since $f(x) = 1$ when $0 < x < 1$, and is zero otherwise, we have

$$f_{X+Y}(u) = \int_0^1 f(u-x)\,\mathrm{d}x = \int_{u-1}^{u} f(t)\,\mathrm{d}t = \begin{cases} u & u \le 1 \\ 2-u & 1 < u \le 2 \end{cases} \tag{14}$$

and we recognize this as a triangular distribution (see section 5.1.24). As another example, consider the sum of two independent exponential random variables with location $a = 0$ and scale b. The density function for the sum is

$$f_{X+Y}(z) = \int_0^z f_X(x)f_Y(z-x)\,\mathrm{d}x = \int_0^z \frac{1}{b}e^{-x/b}\frac{1}{b}e^{-(z-x)/b}\,\mathrm{d}x = \frac{1}{b^2}ze^{-z/b}. \tag{15}$$

Using mathematical induction, it is straightforward to generalize to the case of n independent exponential random variates:

$$f_{X_1+\cdots+X_n}(x) = \frac{x^{n-1}e^{-x/b}}{(n-1)!b^n} = \text{gamma}(0, b, n), \tag{16}$$

where we recognized this density as the gamma density for location parameter $a = 0$, scale parameter b, and shape parameter $c = n$ (see section 5.1.11).

Thus, the convolution technique for summation applies to a situation where the probability distribution may be written as a sum of other random variates, each of which can be generated directly. The algorithm is as follows:

1. Generate $X_i \sim F_i^{-1}(U)$ for $i = 1, 2, \ldots, n$.
2. Set $X = X_1 + X_2 + \cdots + X_n$.

To pursue this a bit further, we can derive a result that will be useful later. Consider, then, the Erlang distribution; it is a special case of the gamma distribution when the shape parameter c is an integer. From the aforementioned discussion, we see that this is the sum of c independent exponential random variables (see section 5.1.8), so that

$$X = -b\ln X_1 - \cdots - b\ln X_c \;=\; -\,b\ln(X_1 \cdots X_c). \tag{17}$$

This shows that if we have c IID exponential variates, then the Erlang distribution can be generated via

$$X = -b \ln \prod_{i=1}^{c} X_i. \tag{18}$$

Random variates may be combined in ways other than summation. Consider the product of X and Y. The cumulative distribution is

$$F_{XY}(u) = \iint\limits_{xy \le u} f(x,y)\,\mathrm{d}x\,\mathrm{d}y = \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{u/x} f(x,y)\,\mathrm{d}y \right) \mathrm{d}x. \tag{19}$$

Once again, the density is obtained by differentiating with respect to u:

$$f_{XY}(u) = \int_{-\infty}^{\infty} f(x, u/x) \frac{1}{x}\,\mathrm{d}x. \tag{20}$$

Let us apply this to the product of two uniform densities. We have

$$f_{XY}(u) = \int_{-\infty}^{\infty} f(x) f(u/x) \frac{1}{x}\,\mathrm{d}x. \tag{21}$$

On the unit interval, $f(x)$ is zero when $x > 1$ and $f(u/x)$ is zero when $x < u$. Therefore,

$$f_{XY}(u) = \int_{u}^{1} \frac{1}{x}\,\mathrm{d}x = -\ln u. \tag{22}$$

This shows that the log distribution can be generated as the product of two IID uniform variates (see section 5.1.13).

Finally, let's consider the ratio of two variates:

$$F_{Y/X}(u) = \iint\limits_{y/x \le u} f(x,y)\,\mathrm{d}x\,\mathrm{d}y = \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{ux} f(x,y)\,\mathrm{d}y \right) \mathrm{d}x. \tag{23}$$

Differentiating this to get the density,

$$f_{Y/X}(u) = \int_{-\infty}^{\infty} f(x, ux)|x|\,\mathrm{d}x. \tag{24}$$

As an example, let us apply this to the ratio of two normal variates with mean 0 and variance 1. We have

$$f_{Y/X}(u) = \int_{-\infty}^{\infty} f(x) f(ux)|x|\,\mathrm{d}x = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{-x^2/2} e^{-u^2x^2/2} |x|\,\mathrm{d}x, \tag{25}$$

and we find that

$$f_{Y/X}(u) \;=\; \frac{1}{\pi} \int_{0}^{\infty} e^{-(1+u^2)x^2/2} x\,\mathrm{d}x = \frac{1}{\pi(1+u^2)}. \tag{26}$$

This is recognized as a Cauchy distribution (see section 5.1.3).

3.4 Acceptance–Rejection

Whereas the previous techniques are direct methods, this is an indirect technique for generating the desired distribution. It is a more general method, which can be used when more direct methods fail; however, it is generally not as efficient as direct methods. Its basic virtue is that it will always work—even for cases where there is no explicit formula for the density function (as long as there is some way of evaluating the density at any point in its domain). The technique is best understood geometrically. Consider an arbitrary probability density function, $f(x)$, shown in Fig. 2. The motivation behind this method is the simple observation that, if we have some way of generating uniformly distributed points in two dimensions under the curve of $f(x)$, then the frequency of occurrence of the x values will have the desired distribution.

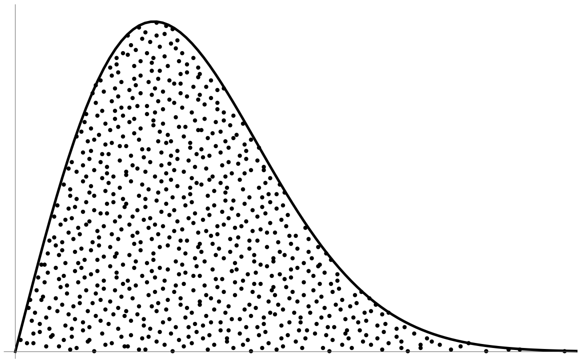

Figure 2. Probability density generated from uniform areal density

A simple way to do this is as follows.

1. Select $X \sim \mathrm{U}(x_{\min}, x_{\max})$.
2. Independently select $Y \sim \mathrm{U}(0, y_{\max})$.
3. Accept X if and only if $Y \leq f(X)$.

This illustrates the idea, and it will work, but it is inefficient due to the fact that there may be many points that are enclosed by the bounding rectangle that lie above the function. So this can be made more efficient by first finding a function $\hat{f}$ that majorizes $f(x)$, in the sense that $\hat{f}(x) \geq f(x)$ for all x in the domain, and, at the same time, the integral of $\hat{f}$ is invertible. Thus, let

$$\hat{F}(x) = \int_{x_{\min}}^{x} \hat{f}(x)\,\mathrm{d}x \quad \text{and define} \quad A_{\max} \equiv \int_{x_{\min}}^{x_{\max}} \hat{f}(x)\,\mathrm{d}x. \tag{27}$$

Then the more efficient algorithm is as follows:

1. Select $A \sim \mathrm{U}(0, A_{\max})$.
2. Compute $X = \hat{F}^{-1}(A)$.
3. Independently select $Y \sim \mathrm{U}(0, \hat{f}(X))$.
4. Accept X if and only if $Y \leq f(X)$.

The acceptance-rejection technique can be illustrated with the following example. Let $f(x) = 10,296x^5(1-x)^7$. It would be very difficult to use the inverse transform method upon this function, since it would involve finding the roots of a 13th degree polynomial. From calculus, we find that $f(x)$ has a maximum value of 2.97188 at $x = 5/12$. Therefore, the function $\hat{f}(x) = 2.97188$ majorizes $f(x)$. So, with $A_{\max} = 2.97188$, $F(x) = 2.97188x$, and $y_{\max} = 2.97188$, the algorithm is as follows:

1. Select $A \sim \mathrm{U}(0, 2.97188)$.
2. Compute $X = A/2.97188$.
3. Independently select $Y \sim \mathrm{U}(0, 2.9718)$.
4. Accept X if and only if $Y \leq f(X)$.

3.5 Sampling and Data–Driven Techniques

One very simple technique for generating distributions is to sample from a given set of data. The simplest technique is to sample with replacement, which effectively treats the data points as independent. The generated distribution is a synthetic data set in which some fraction of the original data is duplicated. The bootstrap method (Diaconis and Efron 1983) uses this technique to generate bounds on statistical measures for which analytical formulas are not known. As such, it can be considered as a Monte Carlo simulation (see section 3.7) We can also sample without replacement, which effectively treats the data as dependent. A simple way of doing this is to first perform a random shuffle of the data and then to return the data in sequential order. Both of these sampling techniques are discussed in section 5.3.3.

Sampling empirical data works well as far as it goes. It is simple and fast, but it is unable to go beyond the data points to generate new points. A classic example that illustrates its limitation is the distribution of darts thrown at a dart board. If a bull's eye is not contained in the data, it will never be generated with sampling. The standard way to handle this is to first fit a known density function to the data and then draw samples from it. The question arises as to whether it is possible to make use of the data directly without having to fit a distribution beforehand, and yet return new values. Fortunately, there is a technique for doing this. It goes by the name of "data-based simulation" or, the name preferred here,"stochastic interpolation." This is a more sophisticated technique that will generate new data points, which have the same statistical properties as the original data at a local level, but without having to pay the price of fitting a distribution beforehand. The underlying theory is discussed in (Taylor and Thompson 1986; Thompson 1989; Bodt and Taylor 1982) and is presented in section 5.3.4.

3.6 Techniques Based on Number Theory

Number theory has been used to generate random bits of 0 and 1 in a very efficient manner and also to produce quasi-random sequences. The latter are sequences of points that take on the appearance of randomness while, at the same time, possessing other desirable properties. Two techniques are included in this report.

1. *Primitive Polynomials Modulo Two*
   These are useful for generating random bits of 1's and 0's that cycle through all possible combinations (excluding all zeros) before repeating. This is discussed in section 5.5.1.

2. *Prime Number Theory*
   This has been exploited to produce sequences of quasi-random numbers that are self-avoiding. This is discussed in section 5.5.2.

3.7 Monte Carlo Simulation

Monte Carlo simulation is a very powerful technique that can be used when the underlying probability density is unknown, or does not come from a known function, but we have a model or method that can be used to simulate the desired distribution. Unlike the other techniques discussed so far, there is not a direct implementation of this method in section 5, due to its generality. Instead, we use this opportunity to illustrate this technique. For this purpose, we use an example that occurs in fragment penetration of plate targets.

Consider a cube of side length a, material density ρ, and mass $m = \rho a^3$. Its geometry is such that one, two, or, at most, three sides will be visible from any direction. Imagine the cube situated at the origin of a cartesian coordinate system with its face surface normals oriented along each of the coordinate axes. Then the presented area of the cube can be parametrized by the polar angle θ and the azimuthal angle ϕ. Defining a dimensionless shape factor γ by

$$A_p = \gamma(m/\rho)^{3/2}, \tag{28}$$

where A_p is the presented area, we find that the dimensionless shape factor is

$$\gamma(\theta, \phi) = \sin\theta\cos\phi + \sin\theta\sin\phi + \cos\theta. \tag{29}$$

It is sufficient to let $\theta \in [0, \pi/2)$ and $\phi \in [0, \pi/2)$ in order for γ to take on all possible values. Once we have this parametrization, it is a simple matter to directly simulate the shape factor according to the following algorithm:

1. Generate $(\theta, \phi) \sim \text{uniformSpherical}(0, \pi/2, 0, \pi/2)$.
2. Return $\gamma = \sin\theta\cos\phi + \sin\theta\sin\phi + \cos\theta$.

Figure 3 shows a typical simulation of the probability density $f(\gamma)$.

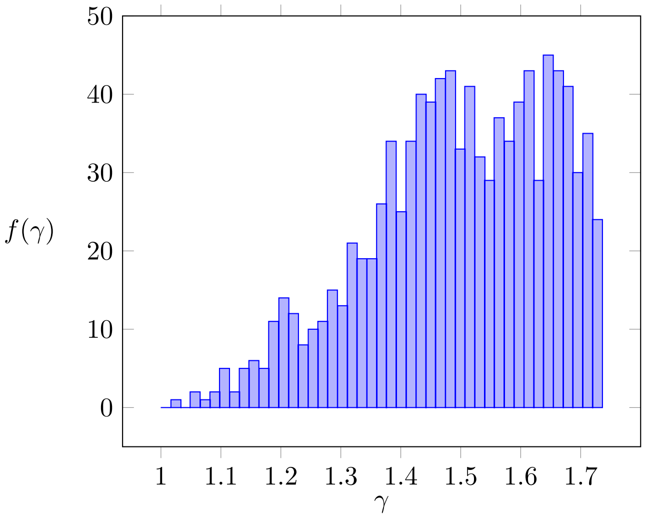

Figure 3. Histogram of a randomly oriented cube via Monte Carlo simulation

3.8 Correlated Bivariate Distributions

If we need to generate bivariate distributions, and the variates are independent, then we simply generate the distribution for each dimension separately. However, there may be known correlations between the variates. Here we show how to generate correlated bivariate distributions.

To generate correlated random variates in two dimensions, the basic idea is that we first generate independent variates and then perform a rotation of the coordinate system to bring about the desired correlation, as shown in Figure 4.

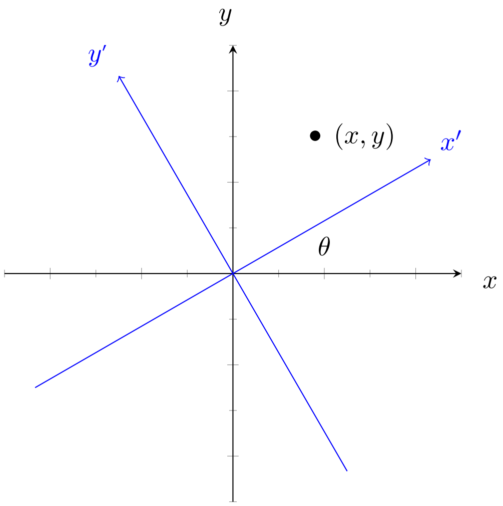

Figure 4. Coordinate rotation to induce correlations

The transformation between the two coordinate systems is given by

$$x' = x\cos\theta + y\sin\theta \quad \text{and} \quad y' = -x\sin\theta + y\cos\theta. \tag{30}$$

Setting the correlation coefficient $\rho = \cos\theta$ so that

$$x' = \rho x + \sqrt{1-\rho^2}\, y \tag{31}$$

induces the desired correlation. To check this,

$$\operatorname{corr}(x,x') = \rho \operatorname{corr}(x,x) + \sqrt{1-\rho^2}\operatorname{corr}(x,y) = \rho(1) + \sqrt{1-\rho^2}\,(0) = \rho, \tag{32}$$

since $\operatorname{corr}(x,x) = 1$ and $\operatorname{corr}(x,y) = 0$.

Here are some special cases:

$$\begin{cases} \theta = 0 & \rho = 1 & x' = x \\ \theta = \pi/2 & \rho = 0 & x' \text{ is independent of } x \\ \theta = \pi & \rho = -1 & x' = -x \end{cases} \tag{33}$$

Thus, the algorithm for generating correlated random variables (x, x'), with correlation coefficient ρ, is as follows.

1. Independently generate X and Y (from the same distribution).
2. Set $X' = \rho X + \sqrt{1-\rho^2}\,Y$.
3. Return the correlated pair (X, X').

3.9 Truncated Distributions

Consider a probability density function $f(x)$ defined on some interval (finite or infinite) and suppose that we want to truncate the distribution to the subinterval $[a, b]$. This can be accomplished by defining a truncated density:

$$\tilde{f}(x) \equiv \begin{cases} \dfrac{f(x)}{F(b) - F(a)} & a \le x \le b \\ 0 & \text{otherwise} \end{cases}, \tag{34}$$

which has corresponding truncated distribution

$$\tilde{F}(x) \equiv \begin{cases} 0 & x < a \\ \dfrac{F(x) - F(a)}{F(b) - F(a)} & a \le x \le b \\ 1 & x > b \end{cases}. \tag{35}$$

An algorithm for generating random variates having distribution function $\tilde{F}$ is as follows:

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Set $Y = F(a) + [F(b) - F(a)]U$.
3. Return $X = F^{-1}(Y)$.

This method works well with the inverse-transform method. However, if an explicit formula for the function F is not available for forming the truncated distribution given in Equation 35, or if we do not have an explicit formula for F^{-1}, then a less efficient but nevertheless correct method of producing the truncated distribution is the following algorithm.

1. Generate a candidate X from the distribution F.
2. If $a \le X \le b$, then accept X; otherwise, go back to step 1.

This algorithm essentially throws away variates that lie outside the domain of interest.

4 Parameter Estimation

The distributions presented in section 5 have parameters that are either known or have to be estimated from data. In the case of continuous distributions, these may include the location parameter, a; the scale parameter, b; and/or the shape parameter, c. In some cases, we need to specify the range of the random variate, $x_{\min}$ and $x_{\max}$. In the case of the discrete distributions, we may need to specify the probability of occurrence, p, and the number of trials, n. Here, we show how these parameters may be estimated from data and present two techniques for doing this.

4.1 Linear Regression (Least–Squares Estimate)

Sometimes, it is possible to linearize the cumulative distribution function by transformation and then to perform a multiple regression to determine the values of the parameters. It can best be explained with an example. Consider the Weibull distribution with location $a = 0$:

$$F(x) = 1 - \exp[-(x/b)^c]. \tag{36}$$

We first sort the data x_i in ascending order:

$$x_1 \le x_2 \le x_3 \le \cdots \le x_N. \tag{37}$$

The corresponding cumulative probability is $F(x_i) = F_i = i/N$. Rearranging Eq. 36 so that the parameters appear linearly, we have

$$\ln[-\ln(1 - F_i)] = c \ln x_i - c \ln b. \tag{38}$$

This shows that if we regress the left-hand side of this equation against the logarithms of the data, then we should get a straight line.[*] The least-squares fit will give the parameter c as the slope of the line and the quantity $-c \ln b$ as the intercept, from which we easily determine b and c.

4.2 Maximum Likelihood Estimation

In this method, we assume that the given data came from some underlying distribution that contains a parameter β whose value is unknown. The probability of getting the observed data with the given distribution is the product of the individual densities:

$$L(\beta) = f_\beta(X_1) f_\beta(X_2) \cdots f_\beta(X_N). \tag{39}$$

The value of β that maximizes $L(\beta)$ is the best estimate in the sense of maximizing the probability. In practice, it is easier to deal with the logarithm of the likelihood function (which has the same location as the likelihood function itself).

As an example, consider the lognormal distribution. The density function is

$$f_{\mu,\sigma^2}(x) = \begin{cases} \dfrac{1}{\sqrt{2\pi}\,\sigma x} \exp\left[-\dfrac{(\ln x - \mu)^2}{2\sigma^2}\right] & x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{40}$$

The log-likelihood function is

$$\ln L(\mu, \sigma^2) = \ln \prod_{i=1}^{N} f_{\mu,\sigma^2}(x_i) = \sum_{i=1}^{N} \ln f_{\mu,\sigma^2}(x_i) \tag{41}$$

[*] We should note that linearizing the cumulative distribution will also transform the error term. Normally distributed errors will be transformed into something other than a normal distribution. However, the error distribution is rarely known, and assuming it is Gaussian to begin with is usually no more than an act of faith. See the chapter “Modeling of Data” in Press et al. (1992) for a discussion of this point.

and, in this case,

$$\ln L(\mu, \sigma^2) = \sum_{i=1}^{N} \left[\ln(\sqrt{2\pi\sigma^2}\, x_i) + \frac{(\ln x_i - \mu)^2}{2\sigma^2} \right]. \tag{42}$$

This is a maximum when both

$$\frac{\partial \ln L(\mu, \sigma^2)}{\partial \mu} = 0 \quad \text{and} \quad \frac{\partial \ln L(\mu, \sigma^2)}{\partial \sigma^2} = 0 \tag{43}$$

and we find

$$\mu = \frac{1}{N} \sum_{i=1}^{N} \ln x_i \quad \text{and} \quad \sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (\ln x_i - \mu)^2. \tag{44}$$

Thus, maximum likelihood parameter estimation leads to a very simple procedure in this case: First, take the logarithms of all the data points; then, μ is the sample mean, and σ^2 is the sample variance.

5 Probability Distribution Functions

In this section, we present the random number distributions in a form intended to be most useful to the actual practitioner of Monte Carlo simulations. The distributions are divided into five subsections as follows:

- **Continuous Distributions**
  There are 27 continuous distributions. For the most part, they make use of three parameters: a location parameter, a; a scale parameter, b; and a shape parameter, c. There are a few exceptions to this notation. In the case of the normal distribution, for instance, it is customary to use μ for the location parameter and σ for the scale parameter. In the case of the beta distribution, there are two shape parameters and these are denoted by v and w. Also, in some cases, it is more convenient for the user to select the interval via x_{min} and x_{max} than the location and scale. The location parameter merely shifts the position of the distribution on the x-axis without affecting the shape, and the scale parameter merely compresses or expands the distribution, also without affecting the shape. The shape parameter may have a small effect on the overall appearance, such as in the Weibull distribution, or it may have a profound effect, as in the beta distribution.

- **Discrete Distributions**
  There are nine discrete distributions. For the most part, they make use of the probability of an event, p, and the number of trials, n.

- **Empirical and Data-Driven Distributions**
  There are four empirical distributions.

- **Bivariate Distributions**
  There are five bivariate distributions.

- **Distributions Generated from Number Theory**
  There are two number-theoretic distributions.

5.1 Continuous Distributions

To aid in selecting an appropriate distribution, we have summarized some characteristics of the continuous distributions in Table 1. The subsections that follow describe each distribution in more detail.

Table 1. Properties for Selecting the Appropriate Continuous Distribution

Distribution Name	Parameters	Symmetric about the Mode?
Arcsine	$x_{\min}$ and $x_{\max}$	yes
Beta	$x_{\min}$, $x_{\max}$ and shape v and w	only when v and w are equal
Cauchy	location a and scale b	yes
Chi–Square	shape v (degrees of freedom)	no
Cosine	$x_{\min}$ and $x_{\max}$	yes
Double Log	$x_{\min}$ and $x_{\max}$	yes
Erlang	scale b and shape c	no
Exponential	location a and scale b	no
Extreme Value	location a and scale b	no
F Ratio	shape v and w (degrees of freedom)	no
Gamma	location a, scale b, and shape c	no
Laplace	location a and scale b	yes
Logarithmic	$x_{\min}$ and $x_{\max}$	no
Logistic	location a and scale b	yes
Lognormal	location a, scale μ, and shape σ	no
Normal (Gaussian)	location μ and scale σ	yes
Parabolic	$x_{\min}$ and $x_{\max}$	yes
Pareto	shape c	no
Pearson's Type 5	scale b and shape c	no
Pearson's Type 6	scale b and shape v and w	no
Power	shape c	no
Rayleigh	location a and scale b	no
Student's t	shape ν (degrees of freedom)	yes
Triangular	$x_{\min}$, $x_{\max}$, and shape c	only when $c = (x_{\min} + x_{\max})/2$
Uniform	$x_{\min}$ and $x_{\max}$	yes
User–Specified	$x_{\min}$, $x_{\max}$ and $y_{\min}$, $y_{\max}$	depends upon the function
Weibull	location a, scale b, and shape c	no

5.1.1 Arcsine

Density Function

$$f(x) = \begin{cases} \dfrac{1}{\pi\sqrt{x(1-x)}} & 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} 0 & x < 0 \\ \dfrac{2}{\pi}\sin^{-1}(\sqrt{x}) & 0 \le x \le 1 \\ 1 & x > 1 \end{cases}$$

Input x_{min}, minimum value of random variable; x_{max}, maximum value of random variable

Output $x \in [x_{\text{min}}, x_{\text{max}})$

Mode x_{min} and x_{max}

Median $(x_{\text{min}} + x_{\text{max}})/2$

Mean $(x_{\text{min}} + x_{\text{max}})/2$

Variance $(x_{\text{max}} - x_{\text{min}})^2/8$

Regression Equation $\sin^2(F_i\pi/2) = x_i/(x_{\text{max}} - x_{\text{min}}) - x_{\text{min}}/(x_{\text{max}} - x_{\text{min}})$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Algorithm 1. Generate $U \sim \text{U}(0,1)$.
2. Return $X = x_{\text{min}} + (x_{\text{max}} - x_{\text{min}})\sin^2(U\pi/2)$.

Source Code

```
double arcsine( double xMin, double xMax ) {

   assert( xMin < xMax );

   double q = sin( M_PI_2 * uniform( 0, 1 ) );
   return xMin + ( xMax - xMin ) * q * q;
}
```

Notes This is a special case of the beta distribution (when $v = w = 1/2$).

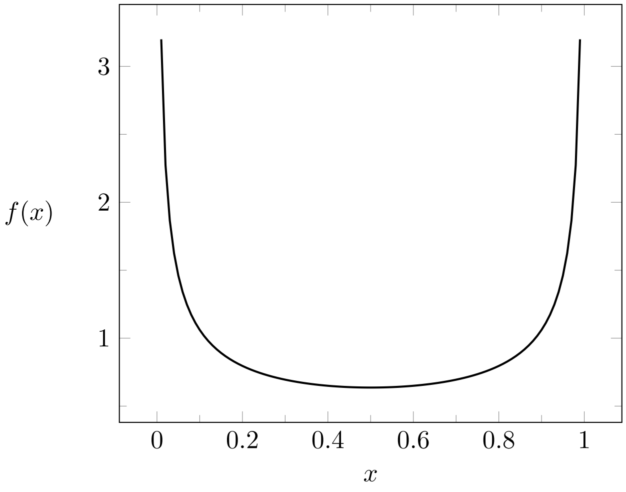

Figure 5. Plot of arcsine PDF

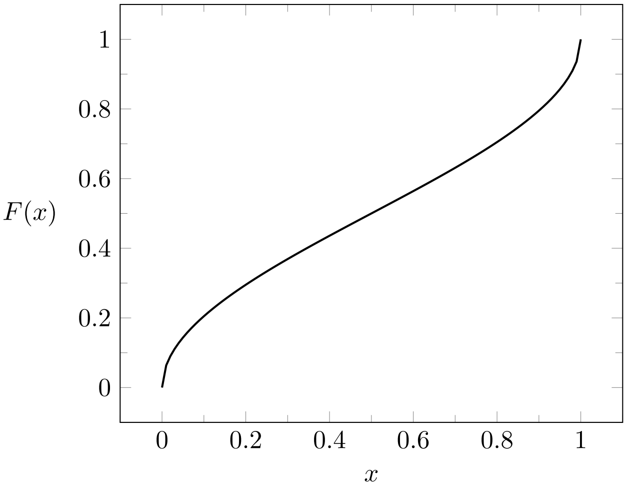

Figure 6. Plot of arcsine CDF

5.1.2 Beta

Density Function

$$f(x) = \begin{cases} \dfrac{x^{v-1}(1-x)^{w-1}}{\mathrm{B}(v,w)} & 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

where $\mathrm{B}(v,w)$ is the *beta function*, defined by $\mathrm{B}(v,w) \equiv \int_0^1 t^{v-1}(1-t)^{w-1}dt$

Distribution Function

$$F(x) = \begin{cases} \mathrm{B}_x(v,w)/\mathrm{B}(v,w) & 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

where the *incomplete beta function* is defined by $\mathrm{B}_x(v,w) \equiv \int_0^x t^{v-1}(1-t)^{w-1}dt$

Input

$x_{\min}$, minimum value of random variable;
$x_{\max}$, maximum value of random variable;
v and w, positive shape parameters

Output $x \in [x_{\min}, x_{\max})$

Mode $(v-1)/(v+w-2)$ for $v > 1$ and $w > 1$ on the interval $[0,1]$

Mean $v/(v+w)$ on the interval $[0,1]$

Variance $vw/[(v+w)^2(1+v+w)]$ on the interval $[0,1]$

Algorithm

1. Generate two IID gamma variates, $Y_1 \sim \text{gamma}(1, v)$ and $Y_2 \sim \text{gamma}(1, w)$.
2. Return $X = \begin{cases} x_{\min} + (x_{\max} - x_{\min})Y_1/(Y_1+Y_2) & \text{if } v \ge w \\ x_{\min} - (x_{\max} - x_{\min})Y_2/(Y_1+Y_2) & \text{if } v < w. \end{cases}$

Source Code

```
double Random::beta( double v, double w, double xMin, double xMax ) {

   if ( v < w ) return xMax - ( xMax - xMin ) * beta( w, v );
   double y1 = gamma( 0., 1., v );
   double y2 = gamma( 0., 1., w );
   return xMin + ( xMax - xMin ) * y1 / ( y1 + y2 );
}
```

Notes

1. $X \sim \mathrm{B}(v,w)$ if and ony if $1 - X \sim \mathrm{B}(w,v)$.
2. When $v = w = 1/2$, this reduces to the arcsine distribution.
3. When $v = w = 1$, this reduces to the uniform distribution.

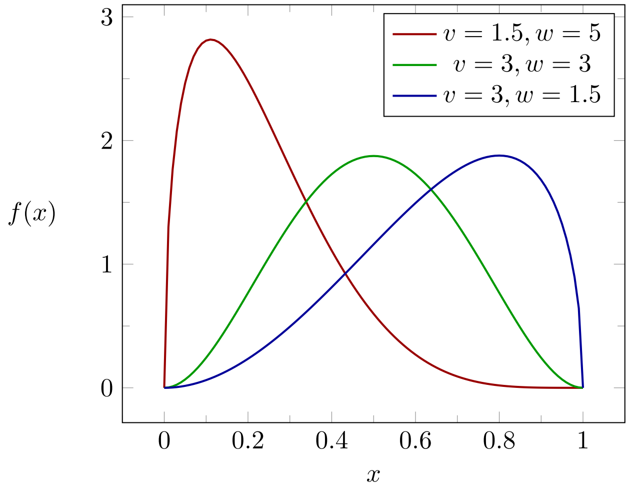

Figure 7. Plot of beta PDF

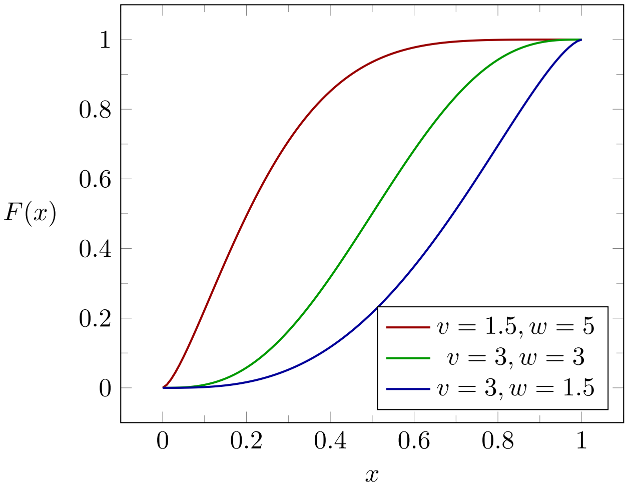

Figure 8. Plot of beta CDF

5.1.3 Cauchy (Lorentz)

Density Function	$f(x) = \dfrac{1}{\pi b}\left[1 + \left(\dfrac{x-a}{b}\right)^2\right]^{-1} \qquad -\infty < x < \infty$
Distribution Function	$F(x) = \dfrac{1}{2} + \dfrac{1}{\pi}\tan^{-1}\left(\dfrac{x-a}{b}\right) \qquad -\infty < x < \infty$
Input	a, location parameter; b, scale parameter is the half-width at half-maximum
Output	$x \in (-\infty, \infty)$
Mode	a
Median	a
Mean	a
Variance	Does not exist
Regression Equation	$\tan[\pi(F_i - 1/2)] = x_i/b - a/b$
Algorithm	1. Generate $U \sim \mathrm{U}(-1/2, 1/2)$. 2. Return $X = a + b\tan(\pi U)$.

Source Code

```
double cauchy( double a, double b ) {

   assert( b > 0 );
   return a + b * tan( M_PI * uniform( -0.5, 0.5 ) );
}
```

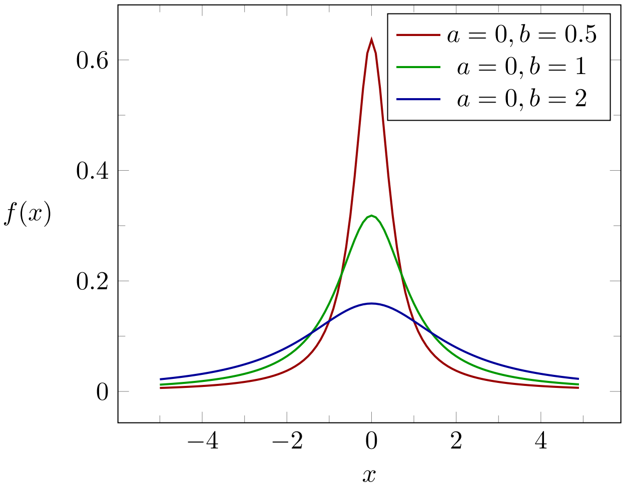

Figure 9. Plot of Cauchy PDF

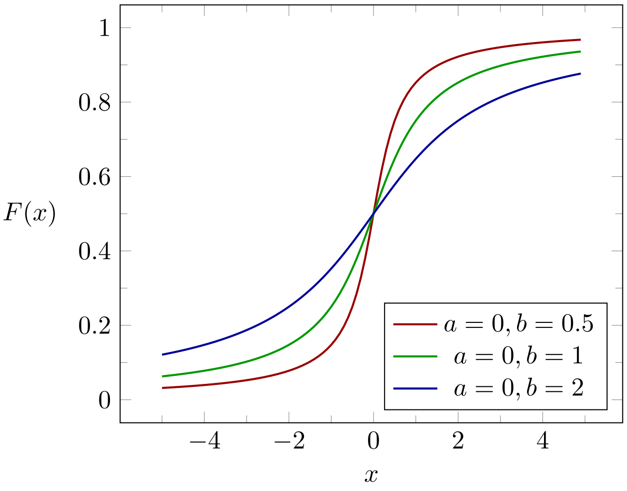

Figure 10. Plot of Cauchy CDF

5.1.4 Chi-Square

Density Function
$$f(x) = \begin{cases} \dfrac{x^{\nu/2-1}e^{-x/2}}{2^{\nu/2}\Gamma(\nu/2)} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\Gamma(z)$ is the *gamma function*, defined by $\Gamma(z) \equiv \int_0^\infty t^{z-1}e^{-t}dt$

Distribution Function
$$F(x) = \begin{cases} \dfrac{1}{2^{\nu/2}\Gamma(\nu/2)} \displaystyle\int_0^x t^{\nu/2-1}e^{-t/2}dt & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Input — Shape parameter $\nu \geq 1$ is the number of degrees of freedom

Output — $x \in (0, \infty)$

Mode — $\nu - 2$ for $\nu \geq 2$

Mean — ν

Variance — 2ν

Algorithm — Return $X \sim \text{gamma}(0, 2, \nu/2)$.

Source Code

```
double Random::chiSquare( int df ) {

   assert( df >= 1 );
   return gamma( 0, 2, 0.5 * double( df ) );
}
```

Notes

1. The chi-square distribution with ν degrees of freedom is equal to the gamma distribution with a scale parameter of 2 and a shape parameter of $\nu/2$.

2. Let $X_i \sim \text{N}(0, 1)$ be IID normal variates for $i = 1, \ldots, \nu$, then $X^2 = \sum_{i=1}^{\nu} X_I^2$ is a χ^2 distribution with ν degrees of freedom.

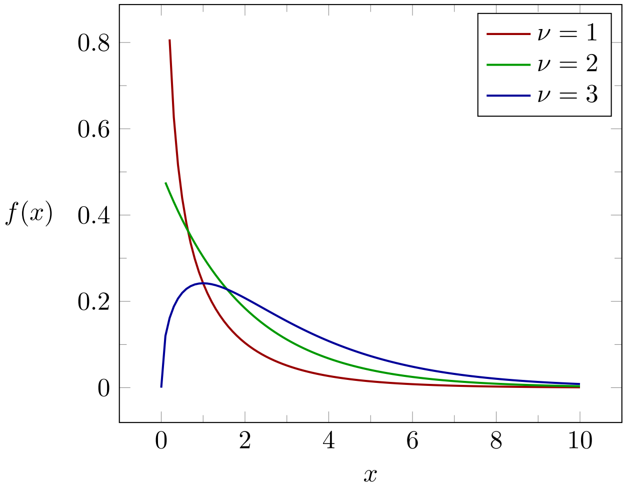

Figure 11. Plot of chi-square PDF

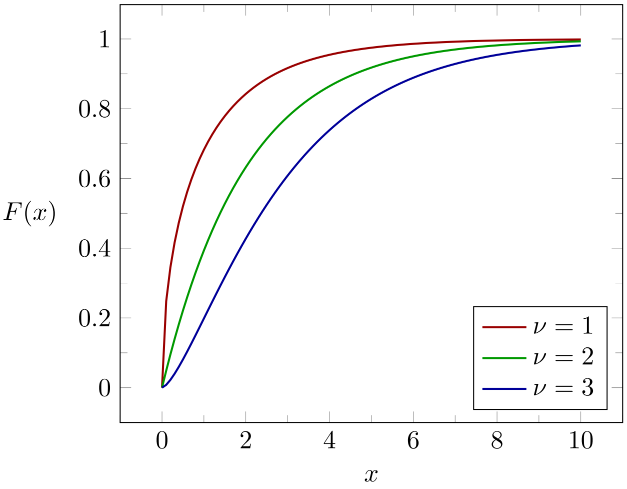

Figure 12. Plot of chi-square CDF

5.1.5 Cosine

Density Function

$$f(x) = \begin{cases} \dfrac{1}{2b}\cos\left(\dfrac{x-a}{b}\right) & x_{\min} \le x \le x_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} 0 & x < x_{\min} \\ \dfrac{1}{2}\left[1 + \sin\left(\dfrac{x-a}{b}\right)\right] & x_{\min} \le x \le x_{\max} \\ 1 & x > x_{\max} \end{cases}$$

Input

$x_{\min}$, minimum value of random variable; $x_{\max}$, maximum value of random variable; location parameter $a = (x_{\min} + x_{\max})/2$; scale parameter $b = (x_{\max} - x_{\min})/\pi$

Output

$x \in [x_{\min}, x_{\max})$

Mode

a

Median

a

Mean

a

Variance

$b^2(\pi^2 - 8)/4$

Regression Equation

$\sin^{-1}(2F_i - 1) = x_i/b - a/b$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Algorithm

1. Generate $U \sim \mathrm{U}(-1, 1)$.
2. Return $X = a + b \sin^{-1} U$.

Source Code

```
double Random::cosine( double xMin, double xMax ) {

   assert( xMin < xMax );
   double a = 0.5 * ( xMin + xMax );    // location parameter
   double b = ( xMax - xMin ) / M_PI;   // scale parameter
   return a + b * asin( uniform( -1, 1 ) );
}
```

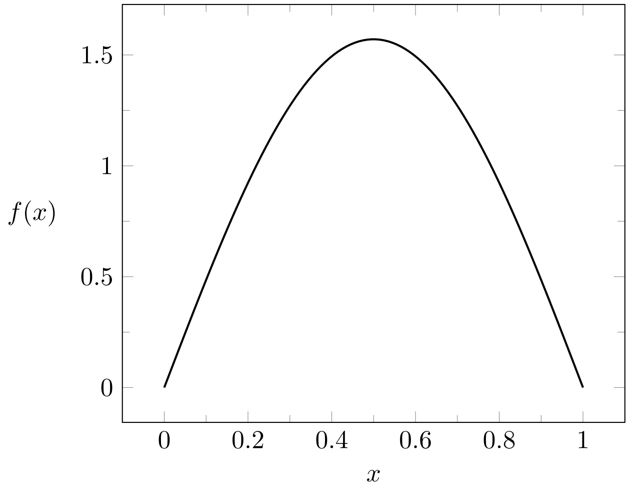

Figure 13. Plot of cosine PDF

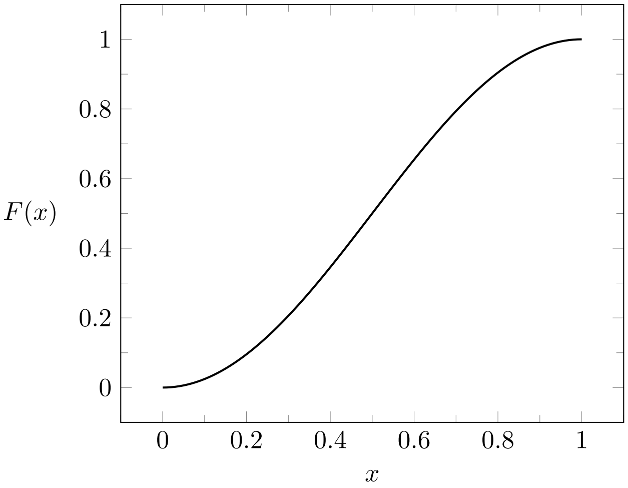

Figure 14. Plot of cosine CDF

5.1.6 Double Log

Density Function

$$f(x)=\begin{cases} -\dfrac{1}{2b}\ln\left(\dfrac{|x-a|}{b}\right) & x_{\min}\le x\le x_{\max} \\ \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x)=\begin{cases} \dfrac{1}{2}-\left(\dfrac{|x-a|}{2b}\right)\left[1-\ln\left(\dfrac{|x-a|}{b}\right)\right] & x_{\min}\le x\le a \\ \\ \dfrac{1}{2}+\left(\dfrac{|x-a|}{2b}\right)\left[1-\ln\left(\dfrac{|x-a|}{b}\right)\right] & a\le x\le x_{\max} \end{cases}$$

Input: $x_{\min}$, minimum value of random variable; $x_{\max}$, maximum value of random variable; location parameter $a=(x_{\min}+x_{\max})/2$; scale parameter $b=(x_{\max}-x_{\min})/\pi$.

Output: $x\in[x_{\min},x_{\max})$

Mode: a (Note that, strictly speaking, $f(a)$ does not exist since $\lim_{x\to a} f(x)=\infty$.)

Median: a

Mean: a

Variance: $(x_{\min}-x_{\max})^2/36$

Algorithm: Based on composition and convolution for the product of two uniform densities:

1. Generate two IID uniform variates, $U_i\sim \mathrm{U}(0,1), i=1,2$.
2. Generate a Bernoulli variate, $U\sim \mathrm{Bernoulli}(0.5)$.
3. If $U=1$, return $X=a+bU_1U_2$; else if $U=0$, return $X=a-bU_1U_2$.

Source Code

```
double Random::doubleLog( double xMin, double xMax ) {

   assert( xMin < xMax );
   double a = 0.5 * ( xMin + xMax );    // location parameter
   double b = 0.5 * ( xMax - xMin );    // scale parameter

   if ( bernoulli( 0.5 ) ) return a + b * uniform() * uniform();
   else                    return a - b * uniform() * uniform();
}
```

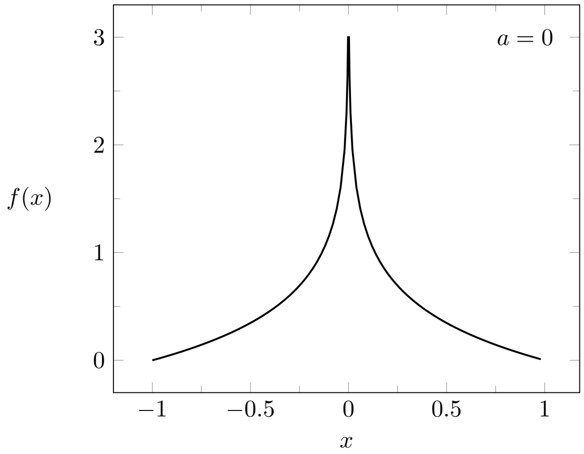

Figure 15. Plot of double log PDF

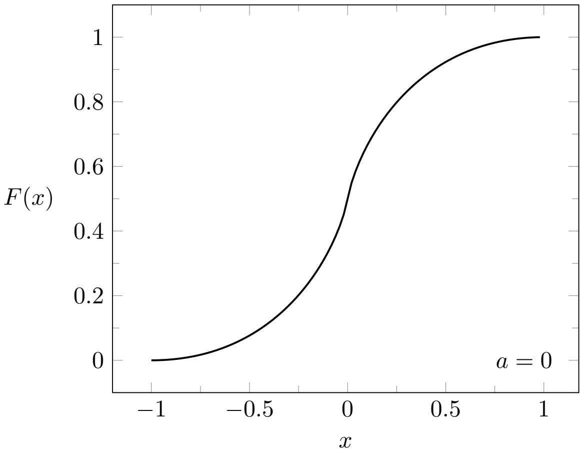

Figure 16. Plot of double log CDF

5.1.7 Erlang

Density Function

$$f(x) = \begin{cases} \dfrac{(x/b)^{c-1} e^{-x/b}}{b(c-1)!} & x \ge 0 \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} 1 - e^{-x/b} \displaystyle\sum_{i=0}^{c-1} \dfrac{(x/b)^i}{i!} & x \ge 0 \\ 0 & \text{otherwise} \end{cases}$$

Input

Scale parameter $b > 0$; shape parameter c, a positive integer.

Output

$x \in [0, \infty)$

Mode

$b(c-1)$

Mean

bc

Variance

$b^2 c$

Algorithm

This algorithm is based on the convolution formula.
1. Generate c IID uniform variates, $U \sim \mathrm{U}(0,1), i = 1, \ldots, c$.
2. Return $X = -b \sum_{i=1}^{c} \ln U_i = -b \ln \prod_{i=1}^{c} U_i$.

Source Code

```
double Random::erlang( double b, int c ) {

   assert( b > 0. && c >= 1 );

   double prod = 1;
   for ( int i = 0; i < c; i++ ) prod *= uniform( 0, 1 );
   return -b * log( prod );
}
```

Notes

The Erlang random variate is the sum of c exponentially-distributed random variates, each with mean b. It reduces to the exponential distribution (when $c = 1$).

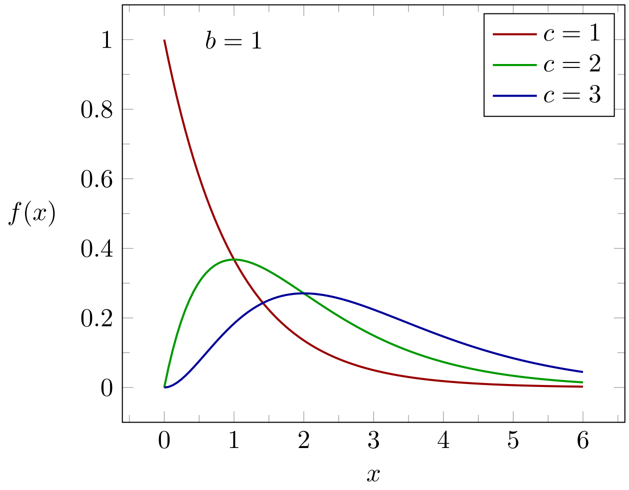

Figure 17. Plot of Erlang PDF

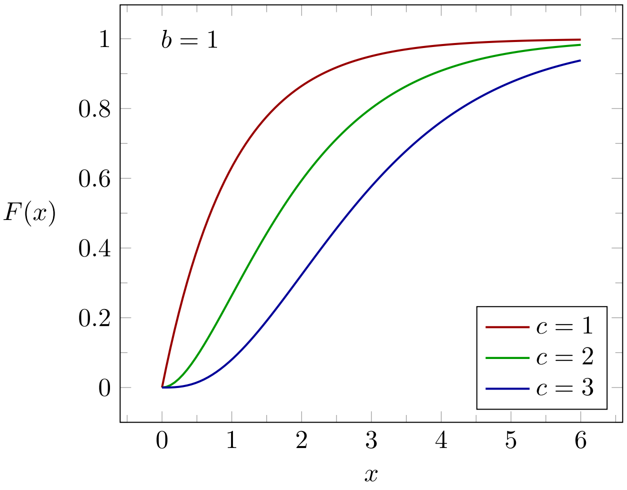

Figure 18. Plot of Erlang CDF

5.1.8 Exponential

Density Function
$$f(x) = \begin{cases} \dfrac{1}{b} e^{-(x-a)/b} & x \geq a \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function
$$F(x) = \begin{cases} 1 - e^{-(x-a)/b} & x \geq a \\ 0 & \text{otherwise} \end{cases}$$

Input: Location parameter a, any real number; scale parameter $b > 0$.

Output: $x \in [a, \infty)$

Mode: a

Median: $a + b \ln 2$

Mean: $a + b$

Variance: b^2

Regression Equation: $-\ln(1 - F_i) = x_i/b - a/b$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Maximum Likelihood: $b = \bar{X}$, the mean value of the random variates

Algorithm:
1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = a - b \ln U$.

Source Code

```
double exponential( double a, double b ) {

   assert( b > 0 );
   return a - b * log( uniform( 0, 1 ) );
}
```

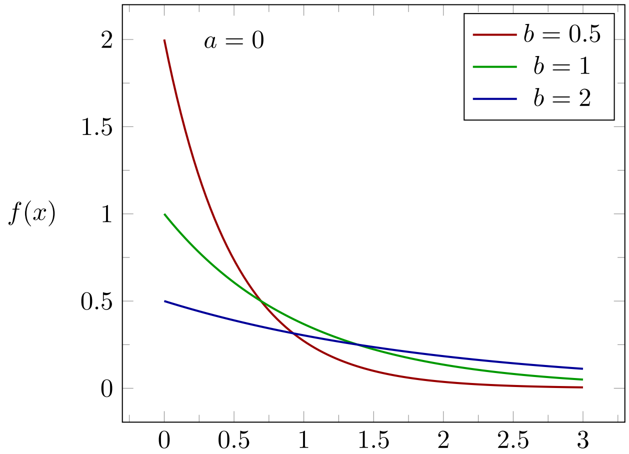

Figure 19. Plot of Exponential PDF

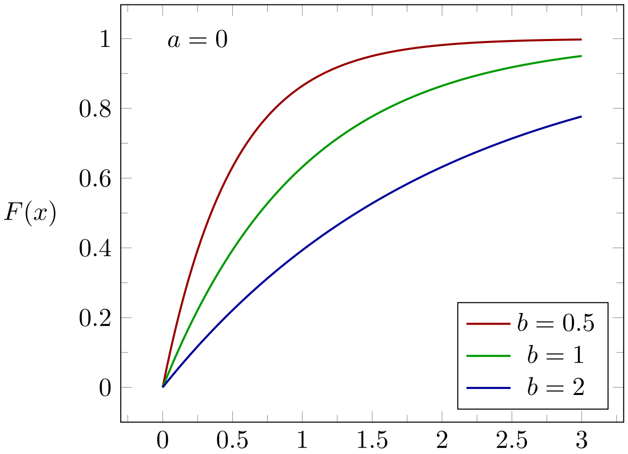

Figure 20. Plot of exponential CDF

5.1.9 Extreme Value

Density Function $f(x) = \frac{1}{b} e^{(x-a)/b} \exp[-e^{(x-a)/b}] \quad -\infty < x < \infty$

Distribution Function $F(x) = 1 - \exp[-e^{(x-a)/b}] \quad -\infty < x < \infty$

Input Location parameter a, any real number; scale parameter $b > 0$.
Output $x \in (-\infty, \infty)$
Mode a
Median $a + b \ln \ln 2$
Mean $a - b\gamma$, where $\gamma \approx 0.57721$ is Euler's constant
Variance $b^2\pi^2/6$
Regression Equation $\ln[-\ln(1 - F_i)] = x_i/b - a/b$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.
Algorithm 1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = a + b \ln(-\ln U)$.

Source Code

```
double extremeValue( double a, double b ) {

   assert( b > 0 );

   return a + b * log( -log( uniform( 0, 1 ) ) );
}
```

Notes This is the distribution of the *smallest* extreme. The distribution of the *largest* extreme may be obtained from this distribution by reversing the sign of X relative to the location parameter a, i.e., $X \to -(X - a)$.

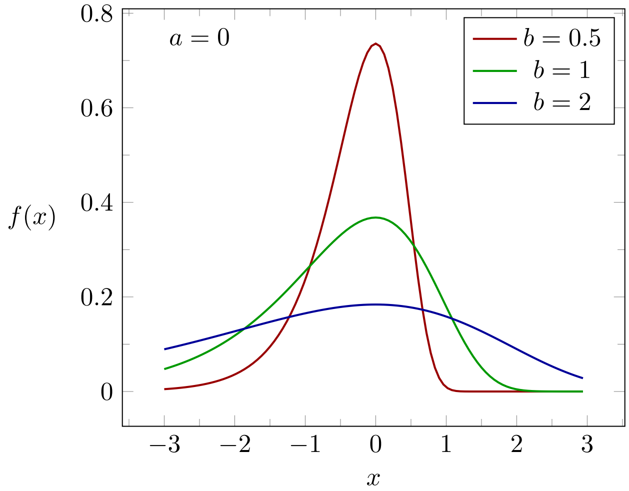

Figure 21. Plot of extreme value PDF

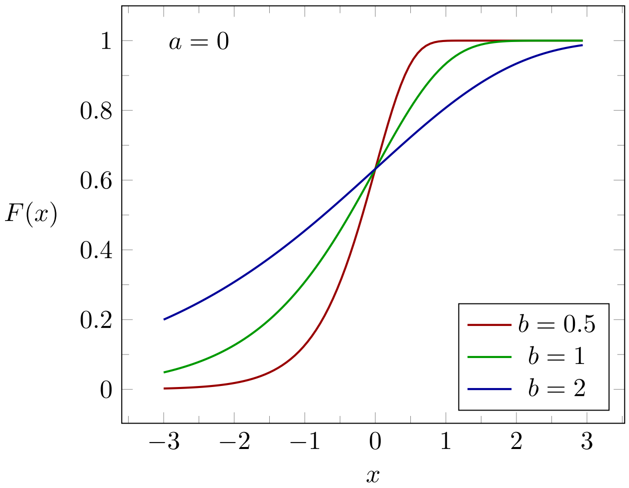

Figure 22. Plots of extreme value CDF

5.1.10 F Ratio

Density Function $$f(x) = \begin{cases} \dfrac{\Gamma[(v+w)/2]}{\Gamma(v/2)\Gamma(w/2)} \dfrac{(v/w)^{v/2} x^{(v-2)/2}}{(1 + xv/w)^{(v+w)/2}} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\Gamma(z)$ is the *gamma function*, defined by $\Gamma(z) \equiv \int_0^\infty t^{z-1} e^{-t} dt$

Distribution Function No closed form, in general.

Input Shape parameters v and w are positive integers (degrees of freedom).

Output $x \in [0, \infty)$

Mode $\dfrac{w(v-2)}{v(w+2)}$ for $v > 2$

Mean $\dfrac{w}{w-2}$ for $w > 2$

Variance $\dfrac{2w^2(v+w-2)}{v(w-2)^2(w-4)}$ for $w > 4$

Algorithm 1. Generate $V \sim \chi^2(v)$ and $W \sim \chi^2(w)$.

2. Return $X = \dfrac{V/v}{W/w}$.

Source Code

```
double fRatio( int v, int w ) {

   assert( v >= 1 && w >= 1 );

   return ( chiSquare( v ) / v ) / ( chisquare( w ) / w );
}
```

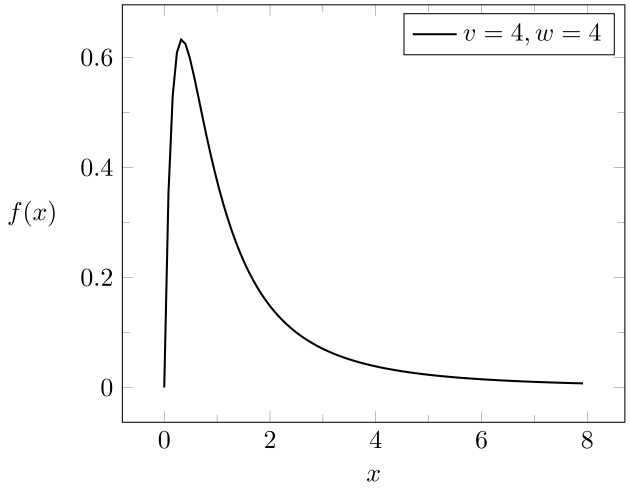

Figure 23. Plot of F Ratio DF

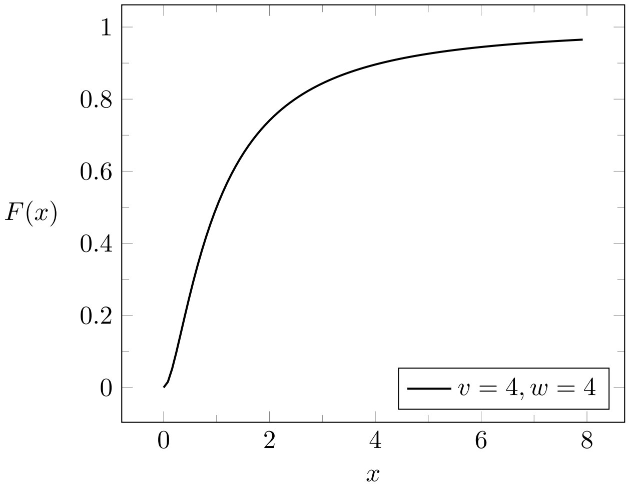

Figure 24. Plot of F Ratio CDF

5.1.11 Gamma

Density Function

$$f(x) = \begin{cases} \dfrac{1}{\Gamma(c)} b^{-c}(x-a)^{c-1}e^{-(x-a)/b} & x > a \\ 0 & \text{otherwise} \end{cases}$$

where $\Gamma(z)$ is the *gamma function*, defined by $\Gamma(z) \equiv \int_0^\infty t^{z-1}e^{-t}dt$

Distribution Function

No closed form, in general. However, if c is a positive integer, then

$$F(x) = \begin{cases} 1 - e^{-(x-a)/b} \displaystyle\sum_{k=0}^{c-1} \frac{1}{k!}\left(\frac{x-a}{b}\right)^k & x > a \\ 0 & \text{otherwise} \end{cases}$$

Input

Location parameter a; scale parameter $b > 0$; shape parameter $c > 0$.

Output

$x \in [a, \infty)$

Mode

$$\begin{cases} a + b(c-1) & c \geq 1 \\ a & c < 1 \end{cases}$$

Mean

$a + bc$

Variance

b^2c

Algorithm

There are three algorithms (Law and Kelton, 1991), depending upon the value of the shape parameter c:

Case 1: $\mathbf{c < 1}$
Let $\beta = 1 + c/e$.
1. Generate $U_1 \sim \mathrm{U}(0,1)$ and set $P = \beta U_1$.
If $P > 1$, go to step 3; otherwise, go to step 2.
2. Set $Y = P^{1/c}$ and generate $U_2 \sim \mathrm{U}(0,1)$.
If $U_2 \leq e^{-Y}$, return $X = Y$; otherwise, go back to step 1.
3. Set $Y = -\ln[(\beta - P)/c]$ and generate $U_2 \sim \mathrm{U}(0,1)$.
If $U_2 \leq Y^{c-1}$, return $X = Y$; otherwise, go back to step 1.

Case 2: $\mathbf{c = 1}$
Return $X \sim$ exponential(a, b).

Case 3: $\mathbf{c > 1}$
Let $\alpha = 1/\sqrt{2c-1}$, $\beta = c - \ln 4$, $q = c + 1/\alpha$, $\theta = 4.5$, and $d = 1 + \ln\theta$.
1. Generate two IID uniform variates, $U_1 \sim \mathrm{U}(0,1)$ and $U_2 \sim \mathrm{U}(0,1)$.
2. Set $V = \alpha\ln[U_1/(1-U_1)]$, $Y = ce^V$, $Z = U_1^2U_2$, and $W = \beta + qV - Y$.
3. If $W + d - \theta Z \geq 0$, return $X = Y$; otherwise, proceed to step 4.
4. If $W \geq \ln Z$, return $X = Y$; otherwise, go back to step 1.

Source Code

```
double gamma( double a, double b, double c ) {

   assert( b > 0. && c > 0. );

   static const double A = 1. / sqrt( 2. * c - 1. );
   static const double B = c - log( 4. );
   static const double Q = c + 1. / A;
   static const double T = 4.5;
   static const double D = 1. + log( T );
   static const double C = 1. + c / M_E;

   if ( c < 1. ) {
      while ( true ) {
         double p = C * _u();
         if ( p > 1. ) {
            double y = -log( ( C - p ) / c );
            if ( _u() <= pow( y, c - 1. ) ) return a + b * y;
         }
         else {
            double y = pow( p, 1. / c );
            if ( _u() <= exp( -y ) ) return a + b * y;
         }
      }
   }
   else if ( c == 1.0 ) return exponential( a, b );
   else {
      while ( true ) {
         double p1 = _u();
         double p2 = _u();
         double v = A * log( p1 / ( 1. - p1 ) );
         double y = c * exp( v );
         double z = p1 * p1 * p2;
         double w = B + Q * v - y;
         if ( w + D - T * z >= 0. || w >= log( z ) ) return a + b * y;
      }
   }
}
```

Notes

1. When $c = 1$, the gamma distribution becomes the exponential distribution.
2. When c is an integer, the gamma distribution becomes the erlang distribution.
3. When $c = v/2$ and $b = 2$, the gamma distribution becomes the chi-square distribution with v degrees of freedom.

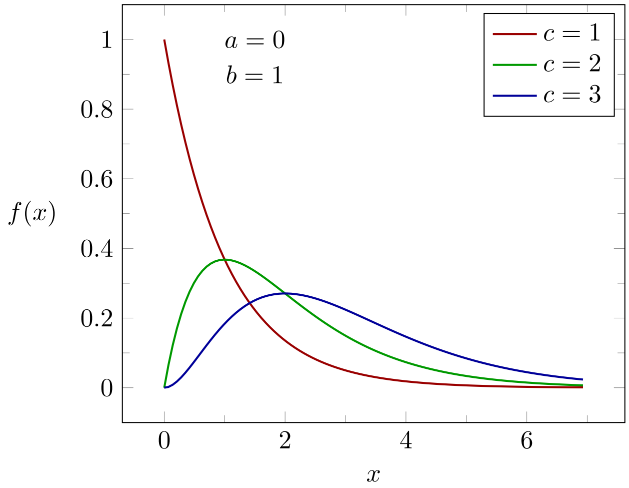

Figure 25. Plot of gamma PDF

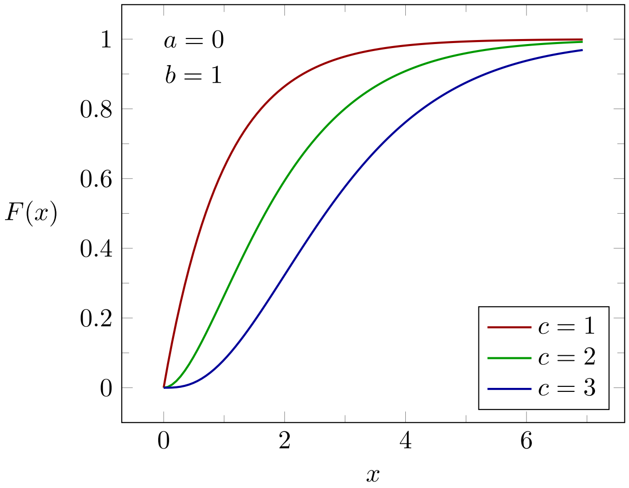

Figure 26. Plot of gamma CDF

5.1.12 Laplace (Double Exponential)

Density Function $$f(x) = \frac{1}{2b} \exp\left(-\frac{|x-a|}{b}\right) \qquad -\infty < x < \infty$$

Distribution Function $$F(x) = \begin{cases} \frac{1}{2} e^{(x-a)/b} & x \le a \\ 1 - \frac{1}{2} e^{-(x-a)/b} & x \ge a \end{cases}$$

Input: Location parameter a, any real number; scale parameter $b > 0$.

Output: $x \in (-\infty, \infty)$

Mode: a

Median: a

Mean: a

Variance: $2b^2$

Regression Equation $$\begin{cases} \ln(2F_i) = x_i/b - a/b & 0 \le F_i \le 1/2 \\ -\ln[2(1-F_i)] = x_i/b - a/b & 1/2 \le F_i \le 1 \end{cases}$$

where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Algorithm

1. Generate two IID random variates, $U_1 \sim \text{U}(0,1)$ and $U_2 \sim \text{U}(0,1)$.
2. Return $X = \begin{cases} a + b \ln U_2 & \text{if } U_1 \ge 1/2 \\ a - b \ln U_2 & \text{if } U_1 < 1/2 \end{cases}$

Source Code

```
double laplace( double a, double b ) {

   assert( b > 0 );

   // composition method
   if ( bernoulli( 0.5 ) ) return a + b * log( uniform( 0, 1 ) );
   else                    return a - b * log( uniform( 0, 1 ) );
}
```

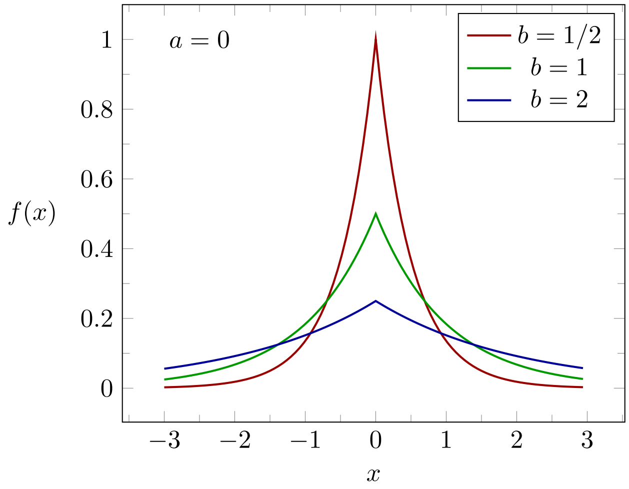

Figure 27. Plot of Laplace PDF

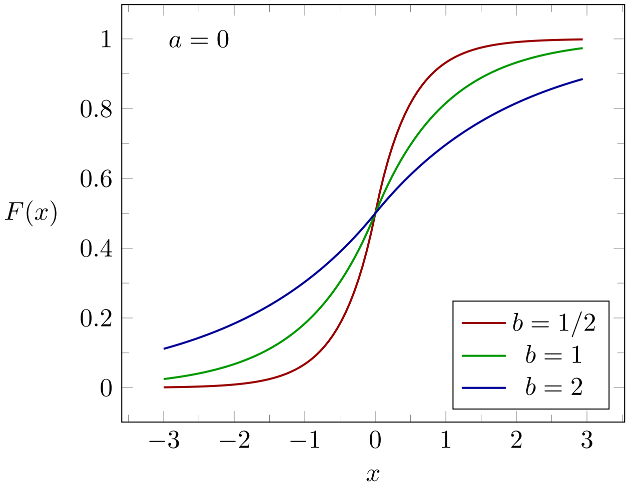

Figure 28. Plot of Laplace CDF

5.1.13 Logarithmic

Density Function

$$f(x) = \begin{cases} -\dfrac{1}{b} \ln\left(\dfrac{x-a}{b}\right) & x_{\min} \le x \le x_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} 0 & x < x_{\min} \\ \left(\dfrac{x-a}{b}\right)\left[1 - \ln\left(\dfrac{x-a}{b}\right)\right] & x_{\min} \le x \le x_{\max} \\ 1 & x > 1 \end{cases}$$

Input: $x_{\min}$, minimum value of random variable; $x_{\max}$, maximum value of random variable; location parameter $a = x_{\min}$; scale parameter $b = x_{\max} - x_{\min}$

Output: $x \in [x_{\min}, x_{\max})$

Mode: $x_{\min}$

Mean: $x_{\min} + (x_{\max} - x_{\min})/4$

Variance: $\dfrac{7}{144}(x_{\max} - x_{\min})^2$

Algorithm: Based on the convolution formula for the product of two uniform densities.
1. Generate two IID uniform variates, $U_1 \sim \mathrm{U}(0,1)$ and $U_2 \sim \mathrm{U}(0,1)$.
2. Return $X = a + bU_1U_2$.

Source Code

```
double logarithmic( double xMin, double xMax ) {

   assert( xMin < xMax );

   double a = xMin;          // location parameter
   double b = xMax - xMin;   // scale parameter
   return a + b * uniform( 0, 1 ) * uniform( 0, 1 );
}
```

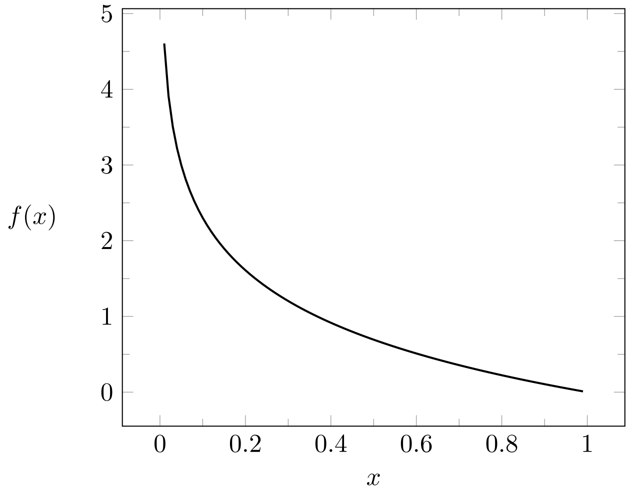

Figure 29. Plot of logarithmic PDF

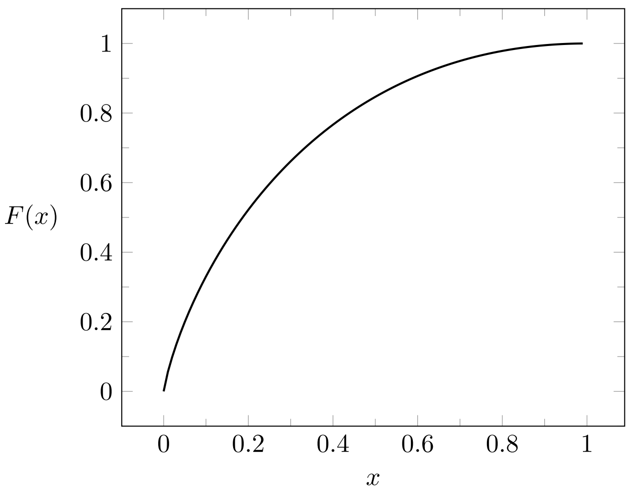

Figure 30. Plot of logarithmic CDF

5.1.14 Logistic

Density Function	$f(x) = \dfrac{1}{b} \dfrac{e^{(x-a)/b}}{[1+e^{(x-a)/b}]^2} \qquad -\infty < x < \infty$
Distribution Function	$F(x) = \dfrac{1}{1+e^{-(x-a)/b}} \qquad -\infty < x < \infty$
Input	Location parameter a, any real number; scale parameter $b > 0$
Output	$x \in (-\infty, \infty)$
Mode	a
Median	a
Mean	a
Variance	$\pi^2 b^2/3$
Regression Equation	$-\ln(F_i^{-1} - 1) = x_i/b - a/b$, where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.
Algorithm	1. Generate $U \sim \mathrm{U}(0,1)$. 2. Return $X = a - b\ln(U^{-1} - 1)$.

Source Code

```
double logistic( double a, double b ) {

   assert( b > 0 );

   return a - b * log( 1 / uniform( 0, 1 ) - 1 );
}
```

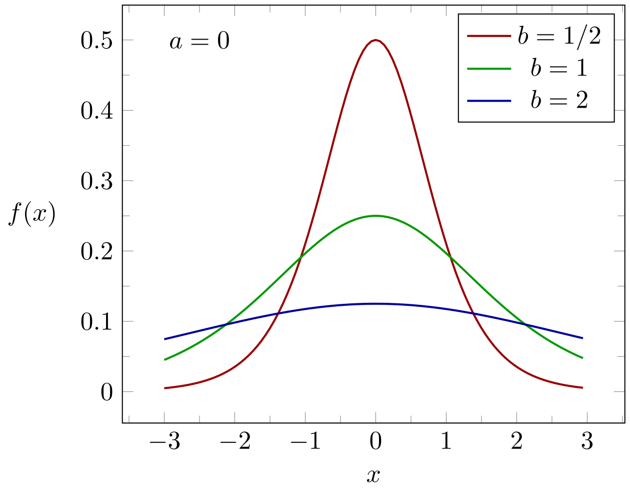

Figure 31. Plot of logistic PDF

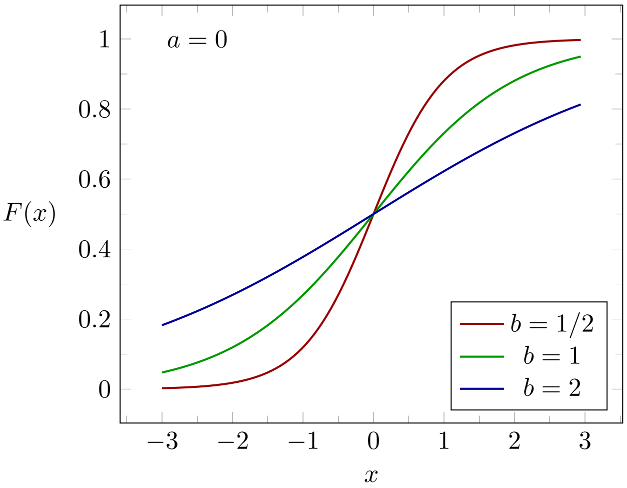

Figure 32. Plot of logistic CDF

5.1.15 Lognormal

Density Function

$$f(x) = \begin{cases} \dfrac{1}{\sqrt{2\pi}\,\sigma(x-a)} \exp\left[-\dfrac{[\ln(x-a)-\mu]^2}{2\sigma^2}\right] & x > a \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} \dfrac{1}{2}\left\{1 + \operatorname{erf}\left[\dfrac{\ln(x-a)-\mu}{\sqrt{2}\,\sigma}\right]\right\} & x > a \\ 0 & \text{otherwise} \end{cases}$$

Input: Location parameter a, any real number, merely shifts the origin; shape parameter $\sigma > 0$; scale parameter μ, any real number.

Output: $x \in [a, \infty)$

Mode: $a + e^{\mu-\sigma^2}$

Median: $a + e^{\mu}$

Mean: $a + e^{\mu+\sigma^2/2}$

Variance: $e^{2\mu+\sigma^2}(e^{\sigma^2} - 1)$

Regression Equation

$$\operatorname{erf}^{-1}(2F_i - 1) = \frac{1}{\sqrt{2}\,\sigma}\ln(x_i - a) - \frac{\mu}{\sqrt{2}\,\sigma},$$

where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \dots, N$.

Maximum Likelihood

$$\mu = \frac{1}{N}\sum_{i=1}^{N} \ln x_i \text{ and } \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(\ln x_i - \mu)^2$$

Algorithm

1. Generate $V \sim \mathrm{N}(\mu, \sigma^2)$.
2. Return $X = a + e^V$.

Source Code

```
double lognormal( double a, double mu, double sigma ) {

   return a + exp( normal( mu, sigma ) );
}
```

Note: $X \sim \mathrm{LN}(\mu, \sigma)$ if and only if $\ln X \sim \mathrm{N}(\mu, \sigma^2)$, where N is the normal distribution.

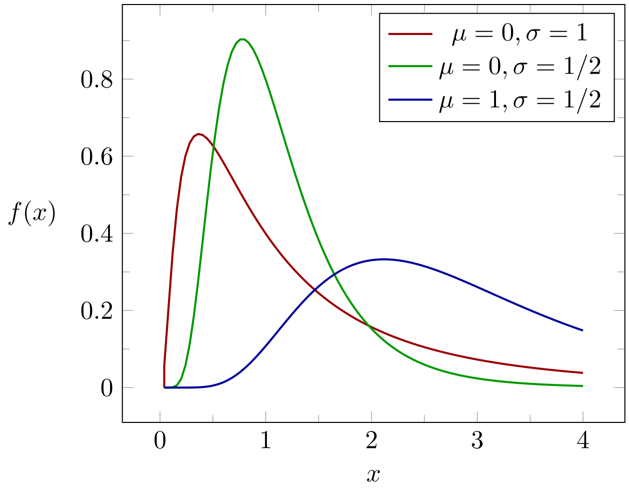

Figure 33. Plot of lognormal PDF

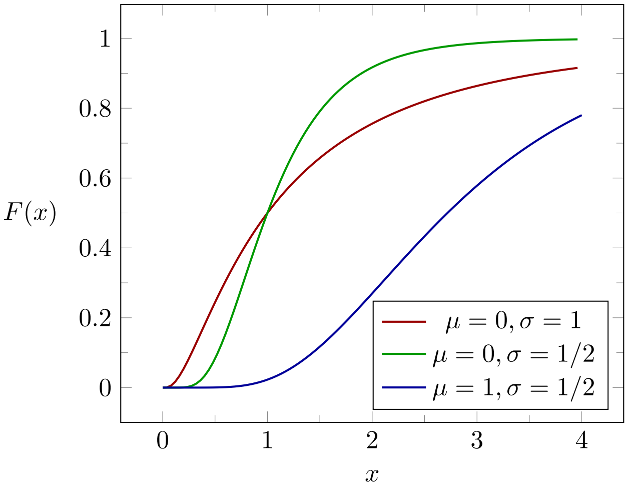

Figure 34. Plot of lognormal CDF

5.1.16 Normal (Gaussian)

Density Function $$f(x) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2} \right] \qquad -\infty < x < \infty$$

Distribution Function $$F(x) = \frac{1}{2}\left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sqrt{2}\,\sigma} \right) \right] \qquad -\infty < x < \infty$$

Input Location parameter μ, any real number; scale parameter $\sigma > 0$.

Output $x \in (-\infty, \infty)$

Mode μ

Median μ

Mean μ

Variance σ^2

Regression Equation $\operatorname{erf}^{-1}(2F_i - 1) = x_i/\sqrt{2}\,\sigma - \mu/\sqrt{2}\,\sigma$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Maximum Likelihood $$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \text{ and } \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

Algorithm
1. Independently generate $U_1 \sim \mathrm{U}(0,1)$ and $U_2 \sim \mathrm{U}(0,1)$.
2. Set $U = U_1^2 + U_2^2$ (note that the square root is not necessary here).
3. If $U < 1$, return $X = \mu + \sigma U_1 \sqrt{-2\ln U/U}$; otherwise go back to step 1.

Source Code

```
double normal( double mu, double sigma ) {
    assert( sigma > 0 );
    static bool f = true;
    static double p2, q;
    double p1, p;
    if ( f ) {
        do { p1 = uniform( -1, 1 ); p2 = uniform( -1, 1 ); p = p1 * p1 + p2 * p2; } while ( p >= 1 );
        f = false;
        q = sqrt( -2 * log( p ) / p );
        return mu + sigma * p1 * q;
    }
    f = true;
    return mu + sigma * p2 * q;
}
```

Note If $X \sim \mathrm{N}(\mu, \sigma)$, then $\exp(X) \sim \Lambda(\mu, \sigma^2)$, the lognormal distribution.

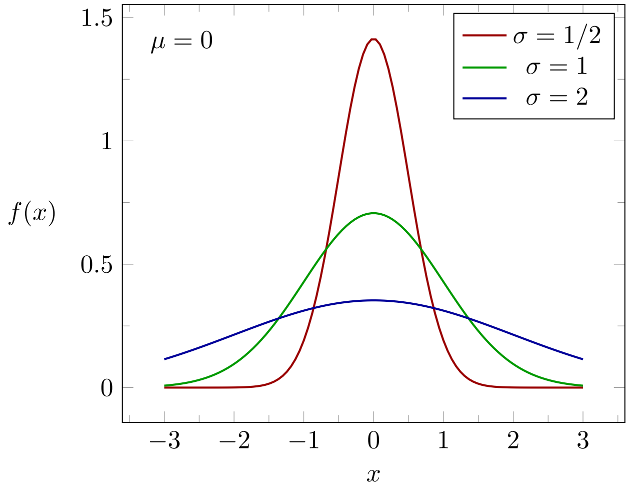

Figure 35. Plot of normal PDF

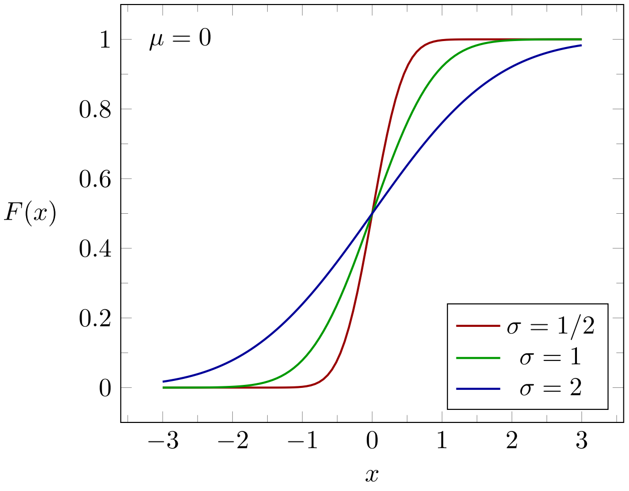

Figure 36. Plot of normal CDF

5.1.17 Parabolic

Density Function $$f(x) = \frac{3}{4b}\left[1 - \left(\frac{x-a}{b}\right)^2\right] \qquad x_{\text{min}} \le x \le x_{\text{max}}$$

Distribution Function $$F(x) = \frac{(a+2b-x)(x-a+b)^2}{4b^3} \qquad x_{\text{min}} \le x \le x_{\text{max}}$$

Input x_{min}, minimum value of random variable; x_{max}, maximum value of random variable; location parameter $a = (x_{\text{min}} + x_{\text{max}})/2$; scale parameter $b = (x_{\text{max}} - x_{\text{min}})/2$;

Output $x \in [x_{\text{min}}, x_{\text{max}})$

Mode $(x_{\text{min}} + x_{\text{max}})/2$

Median $(x_{\text{min}} + x_{\text{max}})/2$

Mean $(x_{\text{min}} + x_{\text{max}})/2$

Variance $(x_{\text{min}} - x_{\text{max}})^2/20$

Algorithm Uses the *acceptance-rejection method* on the above density function, $f(x)$.

Source Code

```
static double parabola( double x, double xMin, double xMax )  // parabola density function

   if ( x < xMin || x > xMax ) return 0;

   double a    = 0.5 * ( xMin + xMax );   // location parameter
   double b    = 0.5 * ( xMax - xMin );   // scale parameter
   double yMax = 0.75 / b;

   return yMax * ( 1. - ( x - a ) * ( x - a ) / ( b * b ) );
}

double parabolic( double xMin, double xMax ) {   // Parabolic distribution

   assert( xMin < xMax );

   double a    = 0.5 * ( xMin + xMax );        // location parameter
   double yMax = parabola( a, xMin, xMax );   // maximum function range

   return userSpecified( parabola, xMin, xMax, 0, yMax );
}
```

Notes

1. This algorithm makes use of the the user-specified distribution.
2. Parabolic is a special case of the beta distribution (when $v = w = 1/2$).

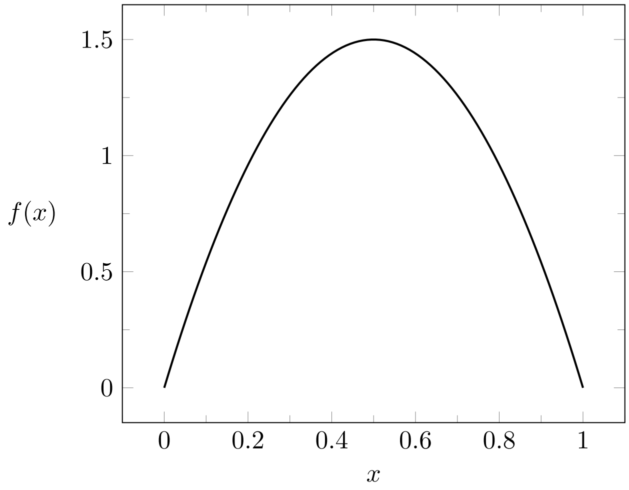

Figure 37. Plot of parabolic PDF

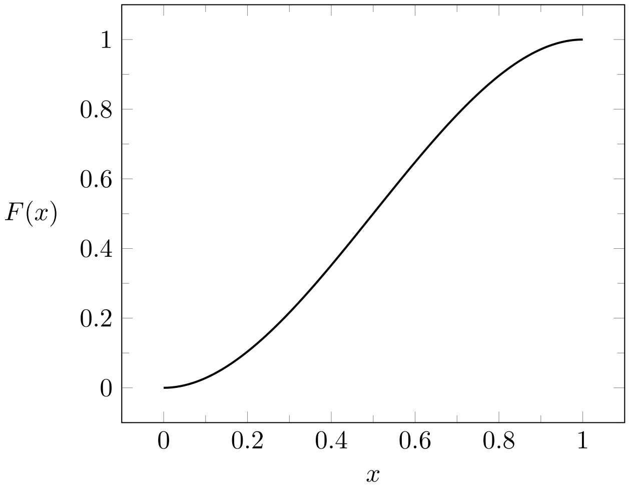

Figure 38. Plot of parabolic CDF

5.1.18 Pareto

Density Function

$$f(x) = \begin{cases} cx^{-c-1} & x \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} 1 - x^{-c} & x \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Input Shape parameter $c > 0$

Output $x \in [1, \infty)$

Mode 1

Median $2^{1/c}$

Mean $c/(c-1)$ for $c > 1$

Variance $[c/(c-2)] - [c/(c-1)]^2$ for $c > 2$

Regression Equation $-\ln(1 - F_i) = c \ln x_i$
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Maximum Likelihood

$$c = \left(\frac{1}{N} \sum_{i=1}^{N} \ln x_i \right)^{-1} = \left(\frac{1}{N} \ln \prod_{i=1}^{N} x_i \right)^{-1}$$

Algorithm

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = U^{-1/c}$.

Source Code

```
double pareto( double c ) {

   assert( c > 0 );

   return pow( uniform( 0, 1 ), -1 / c );
}
```

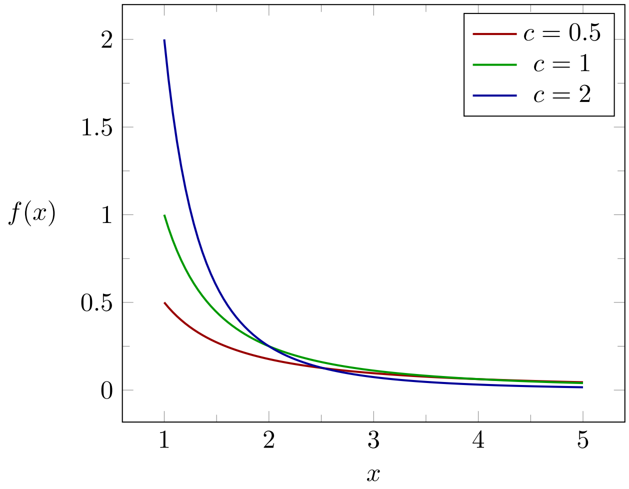

Figure 39. Plot of Pareto PDF

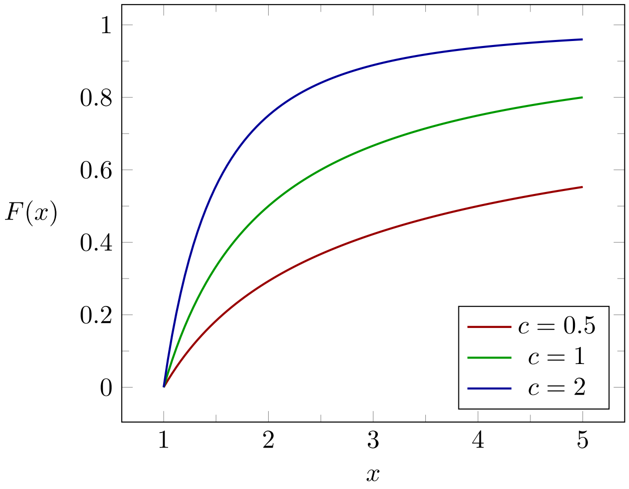

Figure 40. Plot of Pareto CDF

5.1.19 Pearson's Type 5 (Inverted Gamma)

Density Function

$$f(x) = \begin{cases} \dfrac{x^{-(c+1)} e^{-b/x}}{b^{-c}\Gamma(c)} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\Gamma(z)$ is the *gamma function*, defined by $\Gamma(z) \equiv \displaystyle\int_0^\infty t^{z-1} e^{-t} dt$

Distribution Function

$$F(x) = \begin{cases} \dfrac{\Gamma(c, b/x)}{\Gamma(c)} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

where the *incomplete gamma function* is defined by $\Gamma(a, z) \equiv \displaystyle\int_z^\infty t^{a-1} e^{-t} dt$

Input: Scale parameter, $b > 0$; shapew parameter, $c > 0$

Output: $x \in [0, \infty)$

Mode: $b/(c + 1)$

Mean: $b/(c - 1)$ for $c > 1$

Variance: $b^2/[(c - 1)^2(c - 2)]$ for $c > 2$

Algorithm: 1. Generate $Y \sim \text{gamma}(0, 1/b, c)$.
2. Return $X = 1/Y$.

Source Code

```
double pearson5( double c, double b ) {

   assert( c > 0 && b > 0 );

   return 1 / gamma( 0, c, 1 / b );
}
```

Notes: $X \sim \text{PearsonType5}(c, b)$ if and only if $1/X \sim \text{gamma}(0, 1/b, c)$. Thus, the Pearson Type 5 distribution is sometimes called the *inverted gamma distribution.*

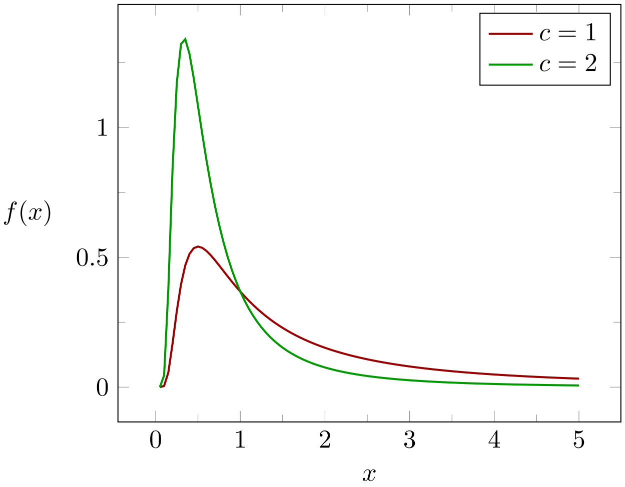

Figure 41. Plot of Pearson's type 5 PDF

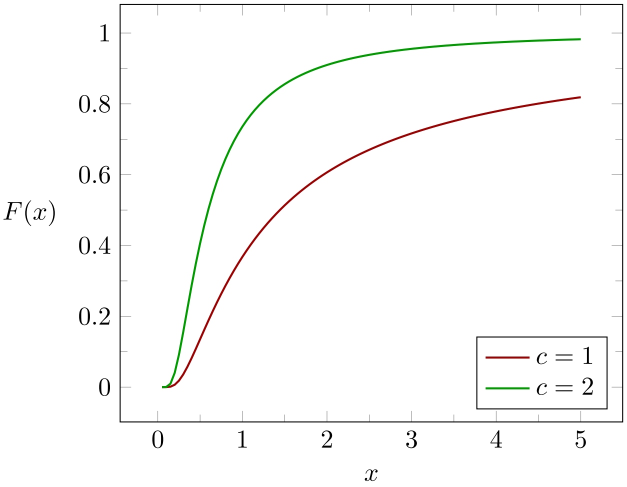

Figure 42. Plot of Pearson's type 5 CDF

5.1.20 Pearson's Type 6

Density Function

$$f(x) = \begin{cases} \dfrac{(x/b)^{v-1}}{b\mathrm{B}(v,w)[1+(x/b)]^{v+w}} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\mathrm{B}(v,w)$ is the *Beta function*, defined by $\mathrm{B}(v,w) \equiv \int_0^1 t^{v-1}(1-t)^{w-1}dt$

Distribution Function

$$F(x) = \begin{cases} F_{\mathrm{B}}\left(\dfrac{x}{x+b}\right) & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $F_{\mathrm{B}}(x)$ is the distribution function of a $\mathrm{B}(v,w)$ random variable.

Input

Shape parameters $v > 0$ and $w > 0$ and scale parameter $b > 0$

Output

$x \in [0, \infty)$

Mode

$$\begin{cases} \dfrac{b(v-1)}{(w+1)} & \text{if } v \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Mean

$\dfrac{bv}{w-1}$ for $w > 1$

Variance

$\dfrac{b^2 v(v+w-1)}{(w-1)^2(w-2)}$ for $w > 2$

Algorithm

1. Generate $Y \sim \text{gamma}(0, v, b)$ and $Z \sim \text{gamma}(0, w, b)$.
2. Return $X = Y/Z$.

Source Code

```
double pearson6( double a1, double a2, double b ) {

   assert( v > 0 && w > 0 && b > 0 );

   return gamma( 0, v, b ) / gamma( 0, w, b );
}
```

Notes

$X \sim \text{PearsonType6}(v, w, 1)$ if and only if $X/(1+X) \sim \text{beta}(v, w)$.

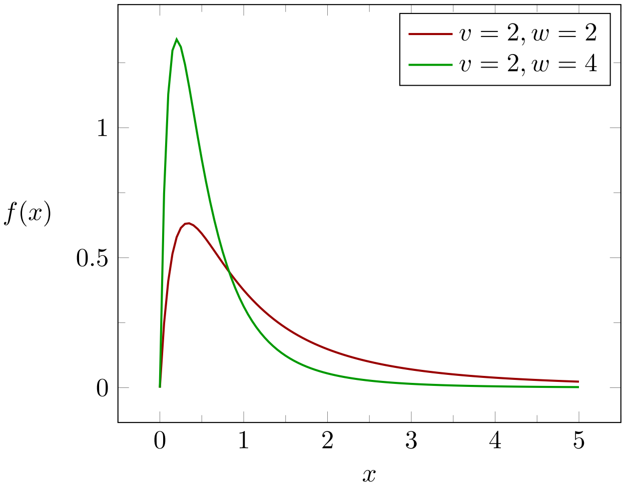

Figure 43. Plot of Pearson's type 6 PDF

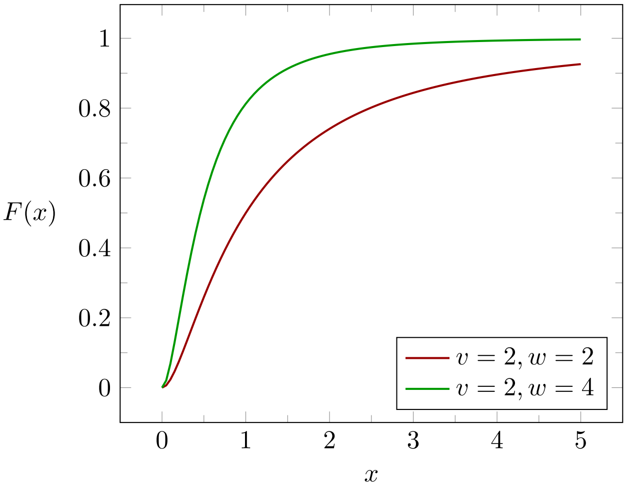

Figure 44. Plot of Pearson's type 6 CDF

5.1.21 Power

Density Function $f(x) = cx^{c-1} \quad 0 \le x \le 1$

Distribution Function $F(x) = x^c \quad 0 \le x \le 1$

Input Shape parameter $c > 0$

Output $x \in [0, 1)$

Mode $\begin{cases} 0 & \text{if } c < 1 \\ 1 & \text{if } c > 1 \end{cases}$

Median $2^{-1/c}$

Mean $\dfrac{c}{c+1}$

Variance $\dfrac{c}{(c+1)^2(c+2)}$

Regression Equation $\ln F_i = c \ln x_i$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Algorithm
1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = U^{1/c}$.

Source Code

```
double power( double c ) {

   assert( c > 0 );

   return pow( uniform( 0, 1 ), 1 / c );
}
```

Notes This reduces to the uniform distribution when $c = 1$.

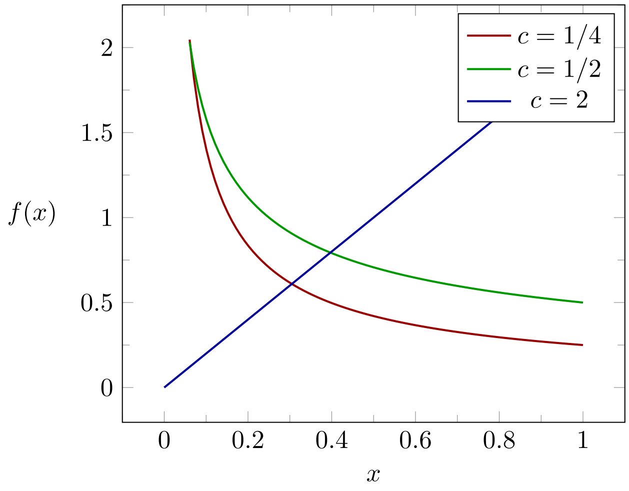

Figure 45. Plot of Power PDF

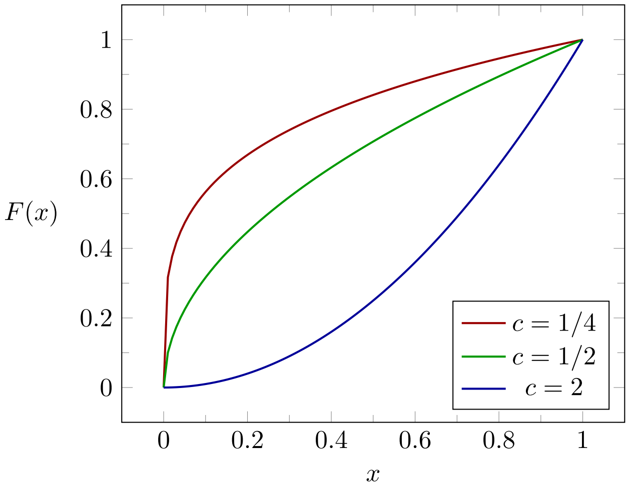

Figure 46. Plot of Power CDF

5.1.22 Raab-Green

Density Function $f(x) = \dfrac{1}{2\pi b}\left[1 + \cos\left(\dfrac{x-a}{b}\right)\right] \quad -x_{\min} \le x \le x_{\max}$

Distribution Function $F(x) = \dfrac{1}{2} + \dfrac{1}{2\pi}\left[\left(\dfrac{x-a}{b}\right) + \sin\left(\dfrac{x-a}{b}\right)\right] \quad -x_{\min} \le x \le x_{\max}$

Input $x_{\min}$, minimum value of random variable; $x_{\max}$, maximum value of random variable; location parameter $a = (x_{\min} + x_{\max})/2$; scale parameter $b = (x_{\max} - x_{\min})/2\pi$.

Output $x \in [-x_{\min}, x_{\max})$

Mode a

Median a

Mean a

Variance $b^2(\pi^2 - 6)/3$

Algorithm This makes use of acceptance-rejection and the alternating series method as developed by Devroye.

Source Code

```
double raab_green( void ) {

   const double x = uniform( -M_PI, M_PI ) );
   const double y = uniform( 0, 2 ) );
   double w = 0, v = 1;
   int n = 0;

   while ( true ) {

      n++;
      v *= x * x / ( ( 2 * n ) * ( 2 * n - 1 ) );
      w += v;
      if ( y >= w ) return x;

      n++;
      v *= x * x / ( ( 2 * n ) * ( 2 * n - 1 ) );
      w -= v;
      if ( y <= w ) return M_PI * sgn( x ) - x;
}
double raab_green( double xMin, double xMax ) {

   assert( xMin < xMax );

   double a = ( xMin + x Max ) /  2.;            // location parameter
   double b = ( xMax - xMin ) / ( 2. * M_PI );   // scale parameter
   return a + b * raab_green();
}
```

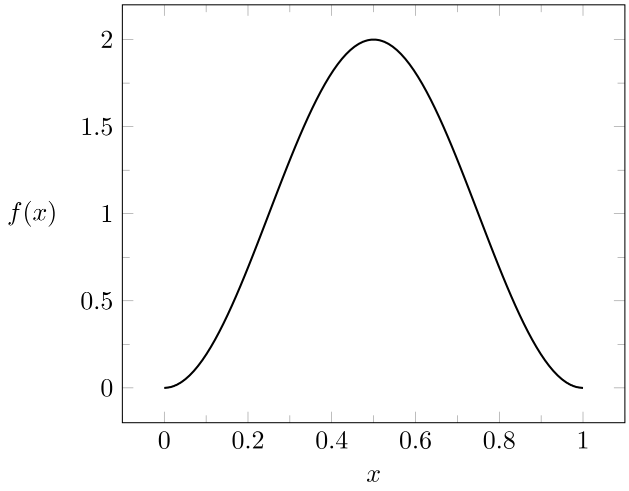

Figure 47. Plot of Raab-Green PDF

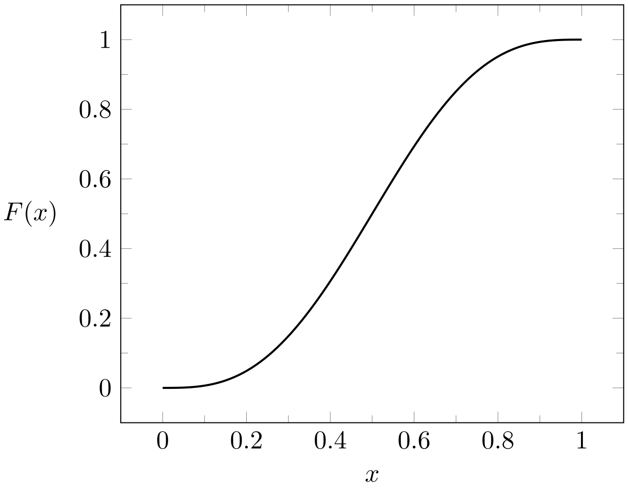

Figure 48. Plot of Raab-Green CDF

5.1.23 Rayleigh

Density Function
$$f(x) = \begin{cases} \dfrac{2}{x-a}\left(\dfrac{x-a}{b}\right)^2 \exp\left[-\left(\dfrac{x-a}{b}\right)^2\right] & x > a \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function
$$F(x) = \begin{cases} 1 - \exp\left[-\left(\dfrac{x-a}{b}\right)^2\right] & x > a \\ 0 & \text{otherwise} \end{cases}$$

Input: Location a, any real number; scale $b > 0$.

Output: $x \in [a, \infty)$

Mode: $a + b/\sqrt{2}$

Median: $a + b\sqrt{\ln 2}$

Mean: $a + b\sqrt{\pi}/2$

Variance: $b^2(1 - \pi/4)$

Regression Equation: $\sqrt{-\ln(1 - F_i)} = x_i/b - a/b$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \dots, N$.

Maximum Likelihood
$$b = \left(\frac{1}{N}\sum_{i=1}^{N} x_i^2\right)^{1/2}$$

Algorithm
1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = a + b\sqrt{-\ln U}$.

Source Code

```
double rayleigh( double a, double b ) {

   assert( b > 0 );

   return a + b * sqrt( -log( uniform( 0, 1 ) ) );
}
```

Notes: Rayleigh is a special case of the Weibull when the shape parameter $c = 2$.

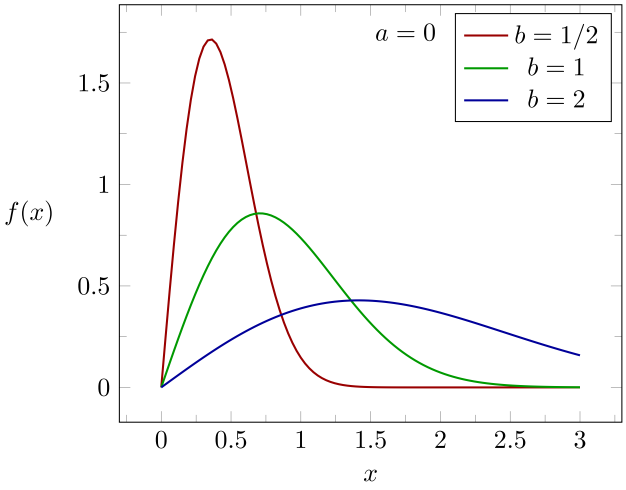

Figure 49. Plot of Rayleigh PDF

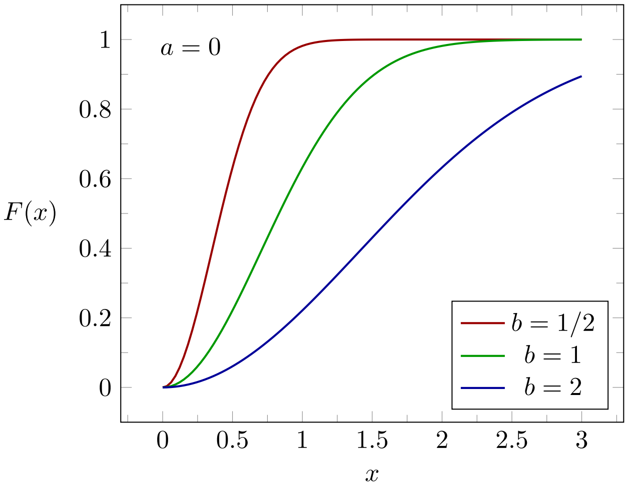

Figure 50. Plot of Rayleigh CDF

5.1.24 Student's t

Density Function	$f(x) = \dfrac{\Gamma[(\nu+1)/2]}{\sqrt{\pi\nu}\,\Gamma(\nu/2)} \left(1 + \dfrac{x^2}{\nu}\right)^{-(\nu+1)/2} \qquad -\infty < x < \infty$ where $\Gamma(z)$ is the *gamma function*, defined by $\Gamma(z) \equiv \int_0^\infty t^{z-1} e^{-t} dt$
Distribution Function	No closed form, in general.
Input	Shape parameter ν, a positive integer (number of degrees of freedom).
Output	$x \in (-\infty, \infty)$
Mode	0
Median	0
Mean	0
Variance	$\nu/(\nu-2)$ for $\nu > 2$
Algorithm	1. Generate $Y \sim \mathrm{N}(0,1)$ and $Z \sim \chi^2(\nu)$. 2. Return $X = Y/\sqrt{Z/\nu}$.

Source Code

```
double studentT( int df ) {

   assert( df >= 1 );

   return normal( 0, 1 ) / sqrt( chiSquare( df ) / df );
}
```

Notes: For $\nu \geq 30$, this distribution can be approximated with the unit normal distribution.

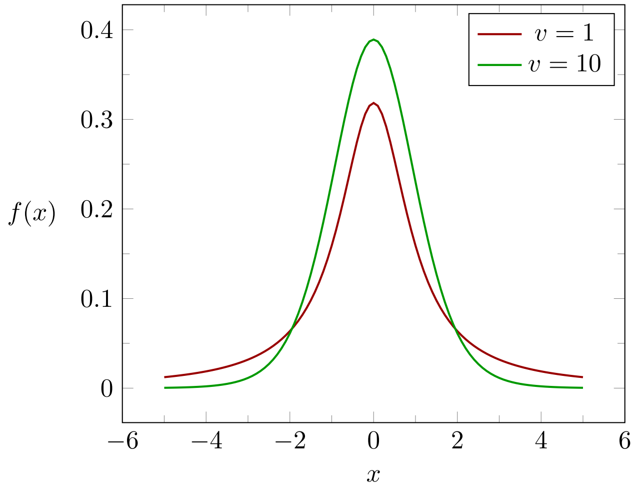

Figure 51. Plot of Student's t PDF

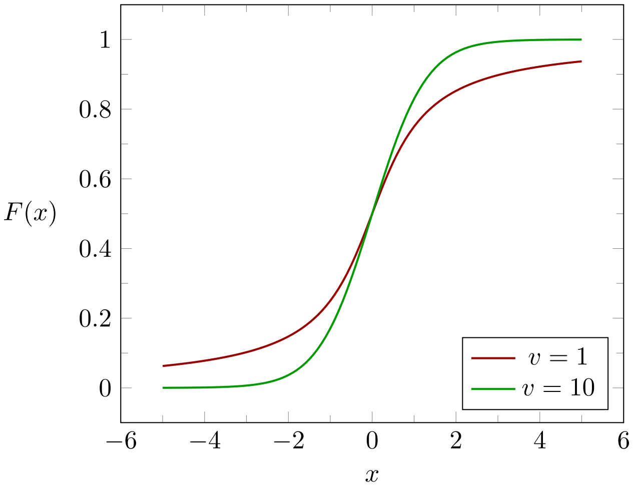

Figure 52. Plot of Student's t CDF

5.1.25 Triangular

Density Function

$$f(x) = \begin{cases} \dfrac{2(x - x_{\text{min}})}{(x_{\text{max}} - x_{\text{min}})(c - x_{\text{min}})} & x_{\text{min}} \le x \le c \\ \dfrac{2(x_{\text{max}} - x)}{(x_{\text{max}} - x_{\text{min}})(x_{\text{max}} - c)} & c \le x \le x_{\text{max}} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} \dfrac{(x - x_{\text{min}})^2}{(x_{\text{max}} - x_{\text{min}})(c - x_{\text{min}})} & x_{\text{min}} \le x \le c \\ 1 - \dfrac{(x_{\text{max}} - x)^2}{(x_{\text{max}} - x_{\text{min}})(x_{\text{max}} - c)} & c \le x \le x_{\text{max}} \end{cases}$$

Input: x_{min}, minimum value of random variable; x_{max}, maximum value of random variable;
c, location of mode

Output: $x \in [x_{\text{min}}, x_{\text{max}})$

Mode: c

Median:

$$\begin{cases} x_{\text{min}} + \sqrt{(x_{\text{max}} - x_{\text{min}})(c - x_{\text{min}})/2} & \text{if } c \ge (x_{\text{min}} + x_{\text{max}})/2 \\ x_{\text{max}} - \sqrt{(x_{\text{max}} - x_{\text{min}})(x_{\text{max}} - c)/2} & \text{if } c \le (x_{\text{min}} + x_{\text{max}})/2 \end{cases}$$

Mean: $(x_{\text{min}} + x_{\text{max}} + c)/3$

Variance: $[3(x_{\text{max}} - x_{\text{min}})^2 + (x_{\text{min}} + x_{\text{max}} - 2c)^2]/72$

Algorithm: 1. Generate $U \sim \text{U}(0, 1)$.

2. Return $X = \begin{cases} x_{\text{min}} + \sqrt{(x_{\text{max}} - x_{\text{min}})(c - x_{\text{min}})U} & U \le (c - x_{\text{min}})/(x_{\text{max}} - x_{\text{min}}) \\ x_{\text{max}} - \sqrt{(x_{\text{max}} - x_{\text{min}})(x_{\text{max}} - c)(1 - U)} & U > (c - x_{\text{min}})/(x_{\text{max}} - x_{\text{min}}) \end{cases}$

Source Code

```
double triangular( double xMin, double xMax, double c ) {

   assert( xMin < xMax && xMin <= c && c <= xMax );

   double p = uniform( 0, 1 ), q = 1 - p;
   if ( p <= ( c - xMin ) / ( xMax - xMin ) )
      return xMin + sqrt( ( xMax - xMin ) * ( c - xMin ) * p );
   else
      return xMax - sqrt( ( xMax - xMin ) * ( xMax - c ) * q );
}
```

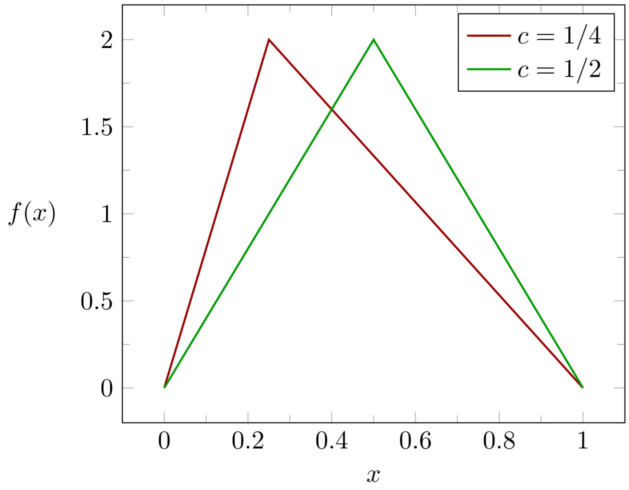

Figure 53. Plot of triangular PDF

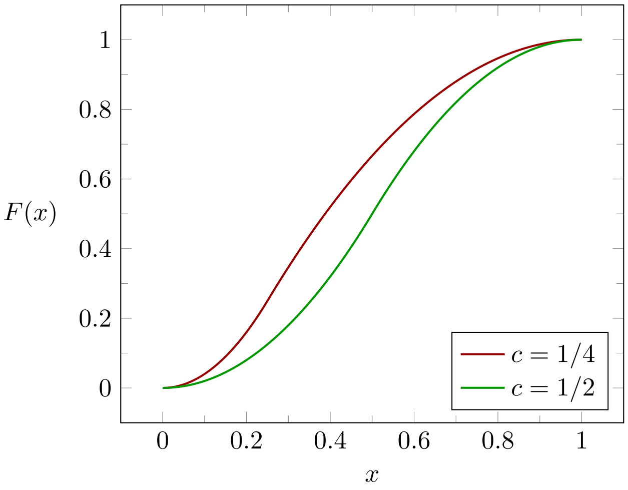

Figure 54. Plot of triangular CDF

5.1.26 Uniform

Density Function

$$f(x) = \begin{cases} \dfrac{1}{x_{\max} - x_{\min}} & x_{\min} \le x \le x_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} 0 & x < x_{\min} \\ \dfrac{x - x_{\min}}{x_{\max} - x_{\min}} & x_{\min} \le x \le x_{\max} \\ 1 & x > x_{\max} \end{cases}$$

Input

$x_{\min}$, minimum value of random variable; $x_{\max}$, maximum value of random variable

Output

$x \in [x_{\min}, x_{\max})$

Mode

Does not uniquely exist.

Median

$(x_{\min} + x_{\max})/2$

Mean

$(x_{\min} + x_{\max})/2$

Variance

$(x_{\min} - x_{\max})^2/12$

Algorithm

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = x_{\min} + (x_{\max} - x_{\min})U$.

Source Code

```
double uniform( double xMin, double xMax ) {

   assert( xMin < xMax );

   return xMin + ( xMax - xMin ) * _u01();
}
```

Notes

1. The source code for _u01() is given in the Appendix.
2. Uniform is the basis for most distributions in the Random class.
2. Uniform is a special case of the beta distribution when $v = w = 1$.

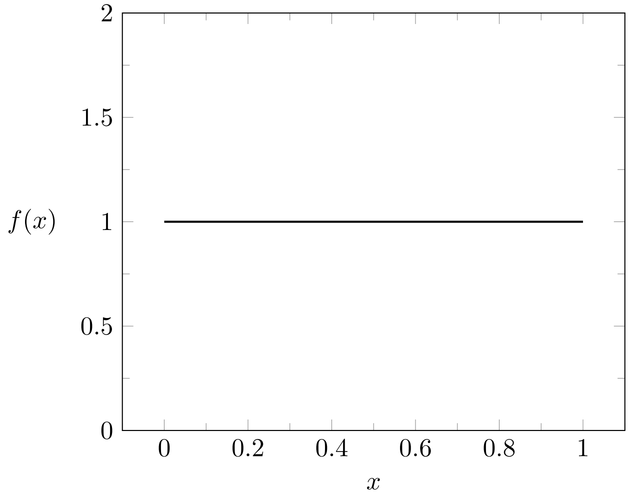

Figure 55. Plot of uniform PDF

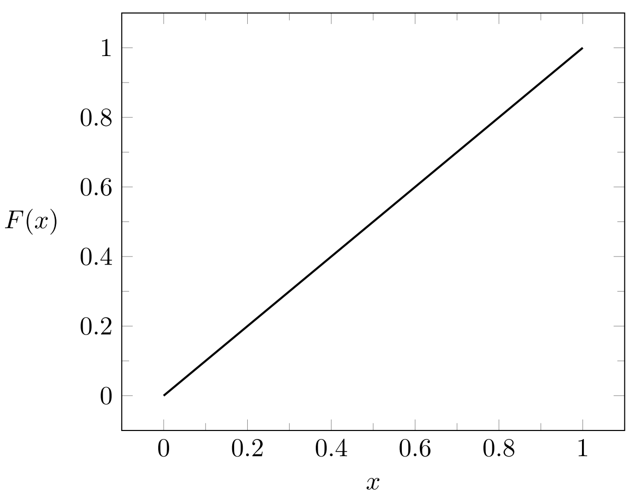

Figure 56. Plot of uniform CDF

5.1.27 User-Specified

Density Function

User-specified, nonnegative function $f(x)$.

Input

$f(x)$, nonnegative function; $x_{\min}$ and $x_{\max}$, minimum and maximum value of domain; $y_{\min}$ and $y_{\max}$, minimum and maximum value of function.

Output

$x \in [x_{\min}, x_{\max})$

Algorithm

1. Generate $A \sim \mathrm{U}(0, A_{\max})$ and $Y \sim \mathrm{U}(y_{\min}, y_{\max})$,
where $A_{\max} \equiv (x_{\max} - x_{\min})(y_{\max} - y_{\min})$ is the area of the rectangle that encloses the function over its specified domain and range.
2. Return $X = x_{\min} + A/(y_{\max} - y_{\min})$ if $f(X) \leq Y$; otherwise, go back to step 1.

Source Code

```
double
Random::userSpecified( double( *usf )( double,       // function
                                       double,       // xMin
                                       double ),     // xMax
                       double xMin, double xMax,     // domain
                       double yMin, double yMax ) {  // range

   assert( xMin < xMax && yMin < yMax );
   double x, y, areaMax = ( xMax - xMin ) * ( yMax - yMin );

   // acceptance-rejection method
   do {
      x = uniform( 0.0, areaMax ) / ( yMax - yMin ) + xMin;
      y = uniform( yMin, yMax );

   } while ( y > usf( x, xMin, xMax ) );
   return x;
}
```

Notes

In order to qualify as a true probability density function, the integral of $f(x)$ over its domain must equal 1, but that is not a requirement here. As long as $f(x)$ is non-negative over its specified domain, it is not necessary to normalize the function. Notice also that an analytical formula is not necessary for this algorithm. Indeed, $f(x)$ could be an arbitrarily-complex computer program. As long as it returns a real value in the range $[y_{\min}, y_{\max}]$, it is suitable as a generator of a random number distribution.

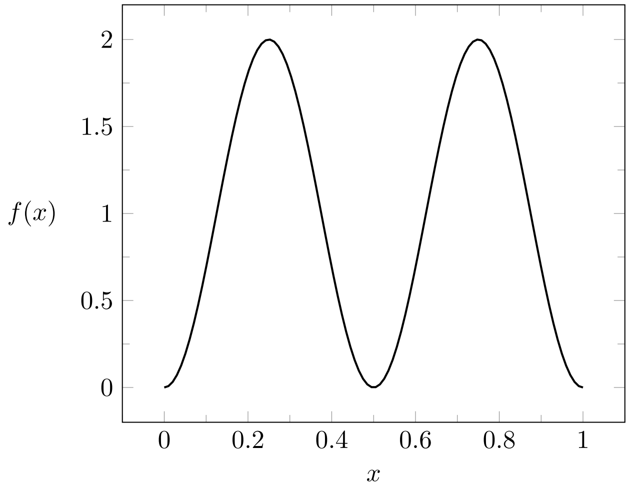

Figure 57. Plot of user-specified PDF

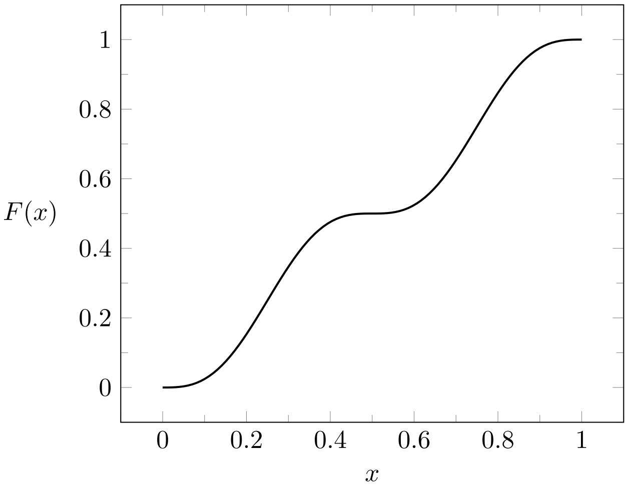

Figure 58. Plot of user-specified CDF

5.1.28 Weibull

Density Function

$$f(x) = \begin{cases} \dfrac{c}{x-a}\left(\dfrac{x-a}{b}\right)^c \exp\left[-\left(\dfrac{x-a}{b}\right)^c\right] & x > a \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(x) = \begin{cases} 1 - \exp\left[-\left(\dfrac{x-a}{b}\right)^c\right] & x > a \\ 0 & \text{otherwise} \end{cases}$$

Input

Location a, any real number; scale $b > 0$; shape $c > 0$

Output

$x \in [a, \infty)$

Mode

$$\begin{cases} a + b(1 - 1/c)^{1/c} & \text{if } c \geq 1 \\ a & \text{if } c \leq 1 \end{cases}$$

Median

$a + b(\ln 2)^{1/c}$

Mean

$a + b\Gamma[(c+1)/c]$

Variance

$b^2\{\Gamma[(c+2)/c] - (\Gamma[(c+1)/c])^2\}$

Regression Equation

$\ln[-\ln(1 - F_i)] = c\ln(x_i - a) - c\ln b$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \ldots, N$.

Algorithm

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Return $X = a + b(-\ln U)^{1/c}$.

Source Code

```
double weibull( double a, double b, double c ) {

   assert( b > 0 && c > 0 );

   return a + b * pow( -log( uniform( 0, 1 ) ), 1 / c );
}
```

Notes

1. When $c = 1$, this becomes the exponential distribution with scale b.
2. When $c = 2$ for general b, it becomes the Rayleigh distribution.

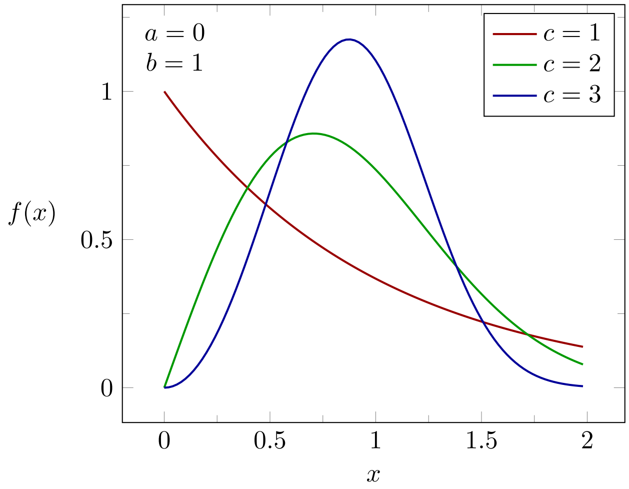

Figure 59. Plot of Weibull PDF

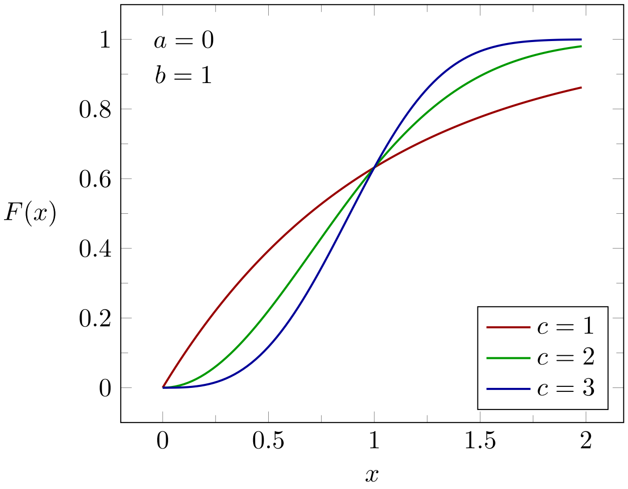

Figure 60. Plot of Weibull CDF

5.2 Discrete Distributions

The discrete distributions make use of one or more of the following parameters:

p	–	the probability of success in a single trial.
n	–	the number of trials performed or number of samples selected.
k	–	the number of successes in n trials or number of trials before first success.
N	–	the number of elements in the sample (population).
K	–	the number of successes contained in the sample.
m	–	the number of distinct events.
μ	–	the success rate.
i	–	smallest integer to consider.
j	–	largest integer to consider.

To aid in selecting an appropriate distribution, Table 2 summarizes some characteristics of the discrete distributions. The subsections that follow describe each distribution in more detail.

Table 2. Parameters and Description for Selecting the Appropriate Discrete Distribution

Distribution Name	**Parameters**	**Output**
Bernoulli	p	success (1) or failure (0)
Binomial	n and p	number of successes ($0 \leq k \leq n$)
Geometric	p	number of trials before first success ($0 \leq k < \infty$)
Hypergeometric	n, N, and K	number of successes ($0 \leq k \leq \min(n, K)$)
Multinomial	n, m, $p_1, \ldots, p_m$	number of successes of each event ($1 \leq k_i \leq m$)
Negative Binomial	p and K	number of failures before K accumulated successes ($0 \leq k < \infty$)
Pascal	p and K	number of trials before K accumulated successes ($1 \leq k < \infty$)
Poisson	μ	number of successes ($0 \leq k < \infty$)
Uniform Discrete	i and j	integer selected ($i \leq k \leq j$)

5.2.1 Bernoulli

A Bernoulli trial is the simulation of a probabilistic event with two possible outcomes: success ($X = 1$) or failure ($X = 0$), where the probability of success in a single trial is p. It is the basis for a number of other discrete distributions.

Density Function

$$f(k) = \begin{cases} 1 - p & \text{if } 0 \\ p & \text{if } 1 \end{cases}$$

Distribution Function

$$F(k) = \begin{cases} 1 - p & \text{if } 0 \le k < 1 \\ 1 & \text{if } k \ge 1 \end{cases}$$

Input

Probability of event, p, where $0 \le p \le 1$

Output

$k \in \{0, 1\}$

Mode

$$\begin{cases} 0 & \text{if } p < 1/2 \\ 0, 1 & \text{if } p = 1/2 \\ 1 & \text{if } p > 1/2 \end{cases}$$

Mean

p

Variance

$p(1 - p)$

Maximum Likelihood

$p = \bar{X}$, the mean value of the IID Bernoulli variates.

Algorithm

1. Generate $U \sim \text{U}(0, 1)$.

2. Return $X = \begin{cases} 1 & \text{if } U < p \\ 0 & \text{if } U \ge p \end{cases}$

Source Code

```
bool bernoulli( double p ) {

   assert( 0 <= p && p <= 1 );

   return uniform( 0, 1 ) < p;
}
```

Notes

1. Notice that if p is strictly zero, the the algorithm above always returns $X = 0$, and if p is strictly one, it always returns $X = 1$, as it should.
2. The sum of n IID Bernoulli variates generates a binomial distribution. Thus, the Bernoulli distribution is a special case of the binomial distribution when the number of trials is one.
3. The number of failures before the first success in a sequence of Bernoulli trials generates a geometric distribution.
4. The number of failures before the first n successes in a sequence of Bernoulli trials generates a negative binomial distribution.
5. The number of Bernoulli trials required to produce the first n successes generates a Pascal distribution.

5.2.2 Binomial

Density Function

$$f(k) = \begin{cases} \binom{n}{k} p^k (1-p)^{n-k} & k \in \{0, 1, \cdots, n\} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(k) = \begin{cases} \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i} & \text{if } 0 \le k \le n \\ 1 & \text{if } k > n \end{cases}$$

where the *binomial coefficient* $\binom{n}{k} \equiv \frac{n!}{k!(n-k)!}$.

Input: Probability of event, p, where $0 \le p \le 1$ and number of trials, $n \ge 1$

Output: The number of successes $k \in \{0, 1, \cdots, n\}$

Mode: The integer k that satisfies $p(n+1) - 1 \le k \le p(n+1)$

Mean: np

Variance: $np(1-p)$

Maximum Likelihood: $p = \bar{X}/n$, where $\bar{X}$ is the mean value of the random variates.

Algorithm

1. Generate n IID Bernoulli trials, $X_i \sim \text{bernoulli}(p)$, where $i = 1, \cdots, n$.
2. Return $X = X_1 + \cdots X_n$.

Source Code

```
int binomial( int n, double p ) {

   assert( 0 <= p && p <= 1 && n >= 1 );

   int sum = 0;
   for ( int i = 0; i < n; i++ ) sum += bernoulli( p );
   return sum;
}
```

Notes

1. The binomial reduces to the bernoulli when $n = 1$.
2. Poisson(np) approximates binomial(n, p) when $p \ll 1$ and $n \gg 1$.
3. For large n, the binomial can be approximated by normal(np, np), provided $np > 5$ and $0.1 \le p \le 0.9$—and for all values of p when $np > 25$.

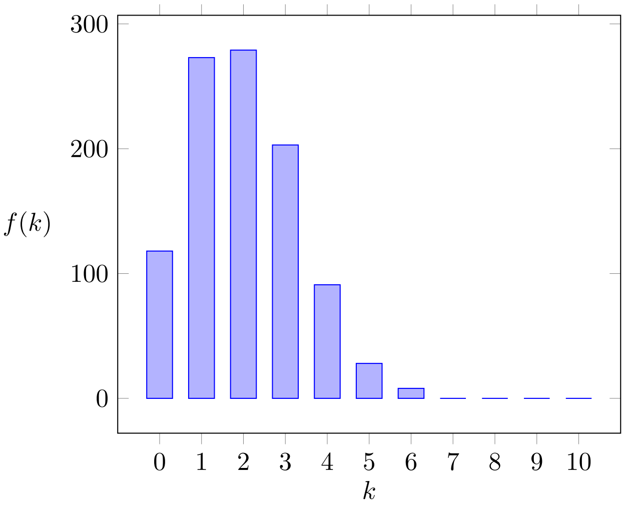

Figure 61. Histogram of binomial PDF

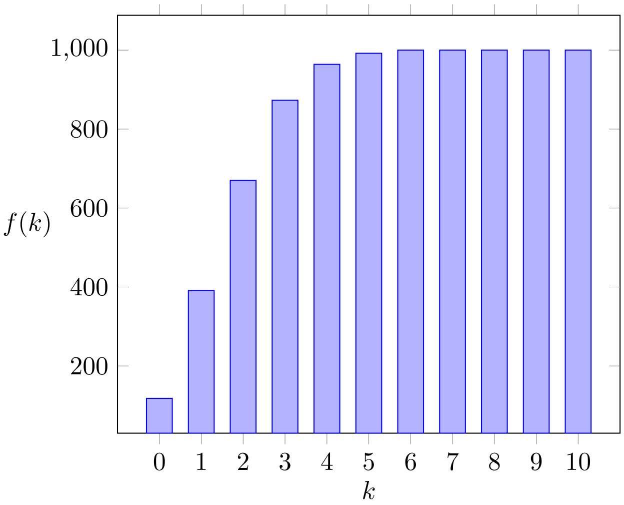

Figure 62. Histogram of binomial CDF

5.2.3 Geometric

The geometric distribution represents the probability of obtaining k failures before the first success in independent Bernoulli trials, where the probability of success in a single trial is p. Or, to state it in a slightly different way, it is the probability of having to perform k trials *before* achieving a success.

Density Function

$$f(k) = \begin{cases} p(1-p)^k & k \in \{0, 1, \cdots\} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(k) = \begin{cases} 1-(1-p)^{k+1} & \text{if } k \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Input: Probability of event, p, where $0 \leq p \leq 1$

Output: The number of trials before a success $k \in \{0, 1, \ldots\}$

Mode: 0

Mean: $(1-p)/p$

Variance: $(1-p)/p^2$

Maximum Likelihood: $p = 1/(1+\bar{X})$, where $\bar{X}$ is the mean value of the IID geometric variates.

Algorithm: 1. Generate $U \sim \mathrm{U}(0,1)$.
2. 2. Return $X = \mathrm{int}(\ln U/(\ln(1-p))$.

Source Code

```
int geometric( double p ) {
   assert( 0 < p && p < 1 );
   return int( log( uniform( 0, 1 ) ) / log( 1 - p ) );
}
```

Notes: 1. *A word of caution:* There are two different definitions that are in common use for the geometric distribution. The other definition is the number of failures *up to and including* the first success.
2. The geometric distribution is the discrete analog of the exponential distribution.
3. If $X_1, X_2, \ldots$ is a sequence of independent Bernoulli(p) random variates and $X = \min\{i \mid X_i = 1\} - 1$, then $X \sim$ geometric(p).

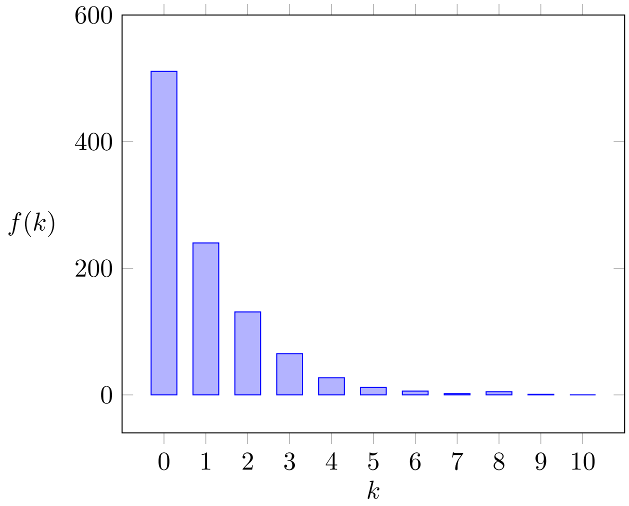

Figure 63. Histogram of geometric PDF

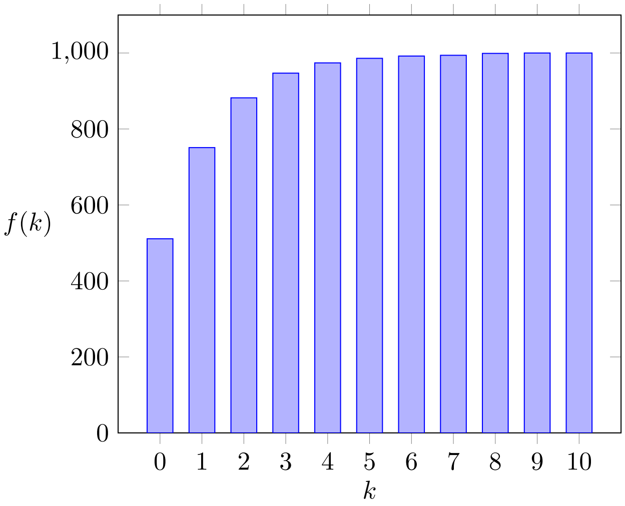

Figure 64. Histogram of Geometric CDF

5.2.4 Hypergeometric

The hypergeometric distribution represents the probability of k successes in n Bernoulli trials, drawn *without replacement*, from a population of N elements that contain K successes.

Density Function $$f(k) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}, \quad \text{where } \binom{n}{k} \equiv \frac{n!}{k!(n-k)!} \text{ is the binomial coefficient}$$

Distribution Function $$F(k) = \sum_{i=0}^{k} \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}, \quad \text{where } 0 \le k \le \min(K, n)$$

Input Number of trials, n; population size, N; successes contained in the population, K.

Output The number of successes $k \in \{0, 1, \cdots, \min(K, n)\}$

Mean np, where $p = K/N$

Variance $np(1-p)\dfrac{N-n}{N-1}$

Algorithm The distribution is generated through simulation of bernoulli trials.

Source Code

```
int hypergeometric( int nTrials, int nPopulation, int nSuccess ) {

    assert( 0 <= nTrials && nTrials <= nPopulation );
    assert( nPopulation >= 1 && nSuccess >= 0 );

    int count = 0;
    for ( int i = 0; i < nTrials; i++, nPopulation-- ) {

        double p = double( nSuccess ) / double( nPopulation );
        if ( bernoulli( p ) ) { count++; nSuccess--; }
    }
    return count;
}
```

Note hypergeometric$(n, N, K) \approx$ binomial$(n, K/N)$ provided $n/N < 0.1$.

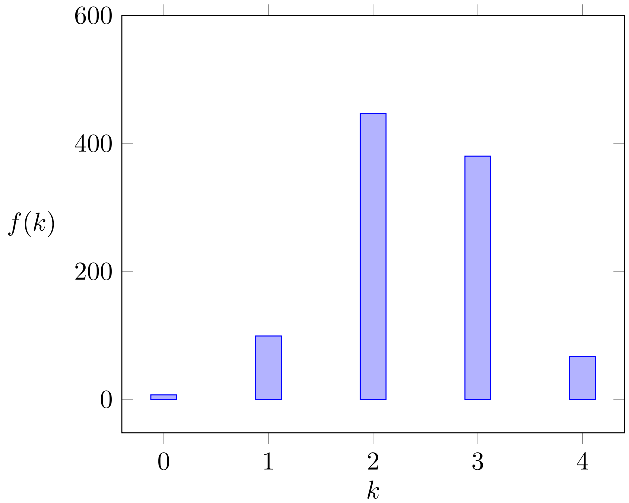

Figure 65. Histogram of hypergeometric PDF

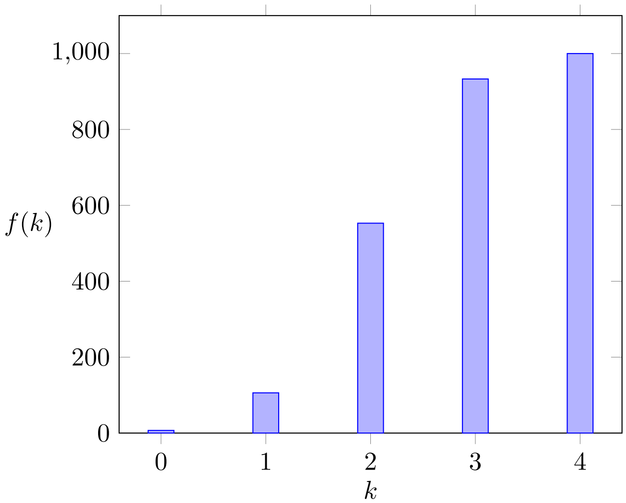

Figure 66. Histogram of hypergeometric CDF

5.2.5 Multinomial

The multinomial distribution is a generalization of the binomial so that instead of two outcomes (success or failure), there are now m possible outcomes, with corresponding probabilities p_i, where $i \in \{1, 2, \ldots, m\}$, and where $p_1 + p_2 + \cdots + p_m = 1$. The density function represents the probability that event 1 occurs k_1 times, event 2 occurs k_2 times, …, and event m occurs k_m times in $k_1 + \cdots + k_m = n$ trials.

Density Function

$$f(k_1, k_2, \ldots, k_m) = \frac{n!}{k_1! k_2! \cdots k_m!} p_1^{k_1} p_2^{k_2} \cdots p_m^{k_m} = n! \prod_{i=1}^{m} \frac{p_i^{k_i}}{k_i!}$$

Input

Number of trials, $n \geq 1$;
number of disjoint events, $m \geq 2$;
probability of each event, p_i, with $p_1 + \cdots + p_m = 1$.

Output

The number of times each of the m events occurs, $k_i \in \{0, \ldots, n\}$,
where $i = 1, \ldots, m$ and $k_1 + \cdots + k_m = n$.

Algorithm

The distribution is generated through simulation.
1. Generate $U_i \sim \mathrm{U}(0, 1)$ for $i = 1, \ldots, n$.
2. For each U_i, locate probability subinterval that contains it and increment counts.

Source Code

```
void multinomial( int    n,            // Multinomial
                  double p[],          // trials n, probability vector p,
                  int    count[],      // success vector count,
                  int    m ) {         // number of disjoint events m

   assert( m >= 2 );   // at least 2 events
   double sum = 0.;
   for ( int bin = 0; bin < m; bin++ ) sum += p[ bin ];    // probabilities
   assert( sum == 1 );                                     // must sum to 1

   for ( int bin = 0; bin < m; bin++ ) count[ bin ] = 0;   // initialize

   // generate n uniform variates in the interval [0,1) and bin the results

   for ( int i = 0; i < n; i++ ) {

      double lower = 0, upper = 0, u = _u01();

      for ( int bin = 0; bin < m; bin++ ) {

      // locate subinterval, which is of length p[ bin ],
      // that contains the variate and increment the corresponding counter

         lower = upper;
         upper += p[ bin ];
         if ( lower <= u && u < upper ) { count[ bin ]++; break; }
      }
   }
}
```

Notes

The multinomial distribution reduces to the binomial distribution when $m = 2$.

5.2.6 Negative Binomial

The negative binomial distribution represents the probability of k failures before the sth success in a sequence of independent Bernoulli trials, where the probability of success in a single trial is p.

Density Function

$$f(k) = \begin{cases} \binom{s+k-1}{k} p^s (1-p)^k & k \in \{0, 1, \ldots\} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(k) = \begin{cases} \sum_{i=0}^{k} \binom{s+i-1}{i} p^k (1-p)^i & \text{if } k \in \{0, 1, \ldots\} \\ 0 & \text{otherwise} \end{cases}$$

Input: Probability of event, p, where $0 \le p \le 1$ and number of successes, $s \ge 1$

Output: The number of failures $k \in \{0, 1, \ldots\}$

Mode:

$$\begin{cases} y \text{ and } y+1 & \text{if } y \text{ is an integer} \\ \text{int}(y) = 1 & \text{otherwise} \end{cases}$$

where $y = [s(1-p) - 1]/p$ and $\text{int}(y)$ is the smallest integer $\le y$

Mean: $s(1-p)/p$

Variance: $s(1-p)/p^2$

Maximum Likelihood: $p = s/(s + \bar{X})$, where $\bar{X}$ is the mean value of the IID variates.

Algorithm: This algorithm is based on the convolution formula.

1. Generate s IID geometric variates, $X_i \sim \text{geometric}(p)$.
2. Return $X = X_1 + \cdots X_s$.

Source Code

```
int negativeBinomial( int s, double p ) {
   assert( s >= 1 );
   int sum = 0;
   for ( int i = 0; i < s; i++ ) sum += geometric( p );
   return sum;
}
```

Notes

1. If $X_1, \ldots, X_s$ are geometric(p) variates, then the sum is negativeBinomial(s, p).
2. The negativeBinomial($1, p$) reduces to geometric(p).

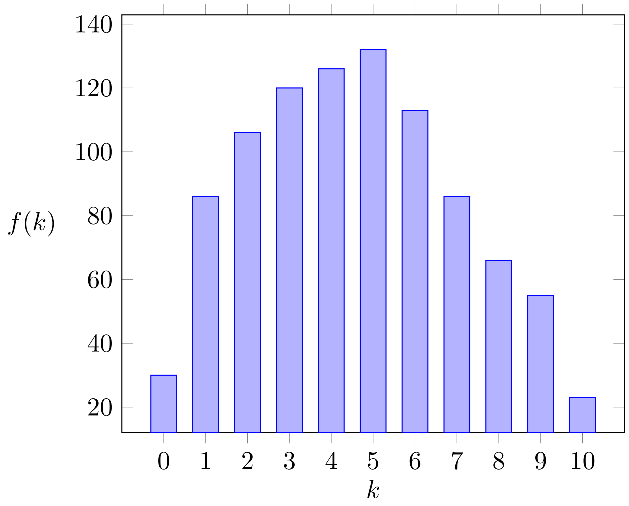

Figure 67. Histogram of negative binomial PDF

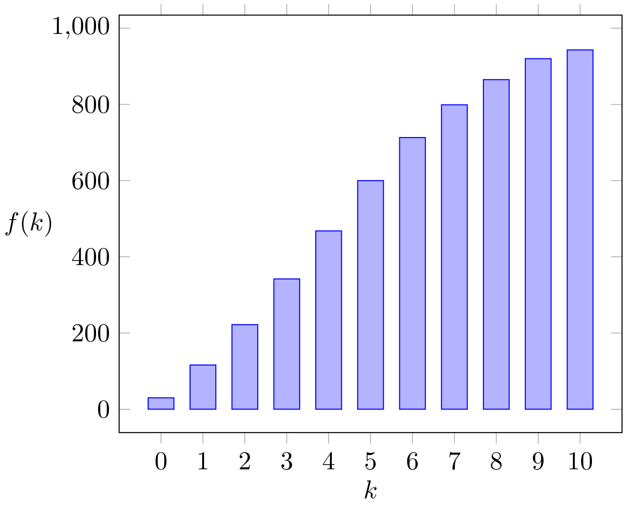

Figure 68. Histogram of negative binomial CDF

5.2.7 Pascal

The Pascal distribution represents the probability of having to perform k trials in order to achieve s successes in a sequence of n independent Bernoulli trials, where the probability of success in a single trial is p.

Density Function

$$f(k) = \begin{cases} \dbinom{k-1}{k-s} p^s (1-p)^{k-s} & k \in \{s, s+1, \ldots\} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(k) = \begin{cases} \displaystyle\sum_{i=0}^{k} \binom{i-1}{i-s} p^s (1-p)^{i-s} & \text{if } k \ge s \\ 0 & \text{otherwise} \end{cases}$$

where the *binomial coefficient* $\dbinom{n}{k} \equiv \dfrac{n!}{k!(n-k)!}$.

Input: Probability of event, p, where $0 \le p \le 1$ and number of successes, $s \ge 1$

Output: The number of failures $k \in \{s, s+1, \ldots\}$

Mode: The integer n that satisfies $1 + np \ge s \ge 1 + (n-1)p$

Mean: s/p

Variance: $s(1-p)/p^2$

Maximum Likelihood: $p = s/n$, where n is the number of trials [unbiassed estimate is $(s-1)/(n-1)$].

Algorithm: This algorithm takes advantage of the logical relationship to the negative binomial.

Source Code

```
int pascal( int s, double p ) {
   return negativeBinomial( s, p ) + s;
}
```

Notes: The Pascal and binomial are inverses of each other in that the binomial returns the number of successes in a given number of trials, whereas the Pascal returns the number of trials required for a given number of successes.

2. Pascal(s, p) = negativeBinomial$(s, p) + s$ and Pascal$(p, 1)$ = geometric$(p) + 1$.

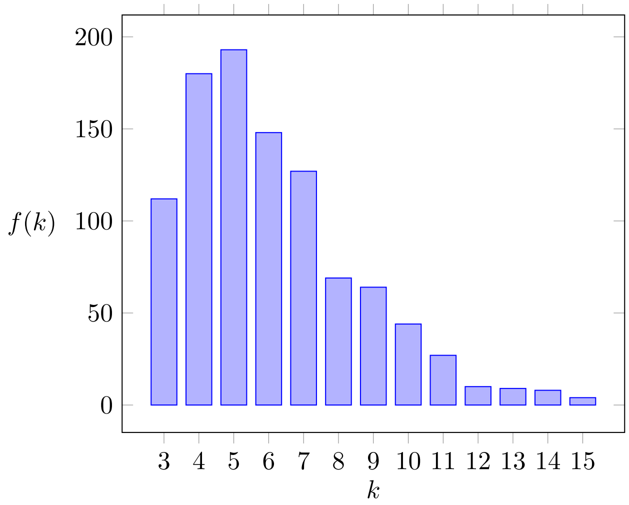

Figure 69. Histogram of Pascal PDF

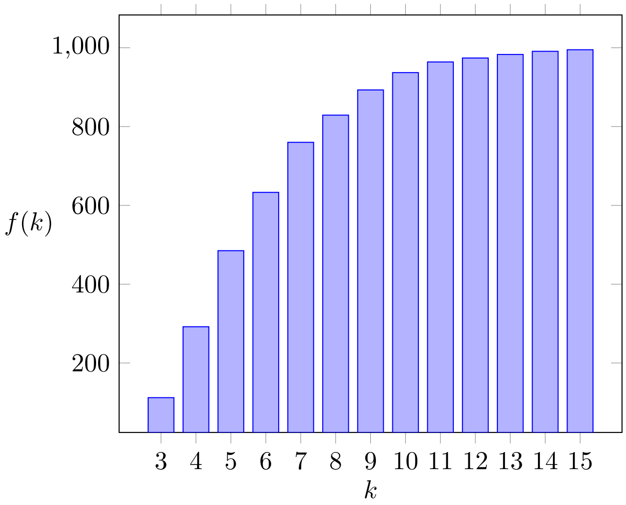

Figure 70. Histogram of Pascal CDF

5.2.8 Poisson

The Poisson distribution represents the probability of k successes when the probability of success in each trial is small and the rate of occurrence, μ, is constant.

Density Function $$f(k) = \begin{cases} \dfrac{\mu^k}{k!} e^{-\mu} & k \in \{0, 1, \ldots\} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function $$F(k) = \begin{cases} \displaystyle\sum_{i=0}^{k} \frac{\mu^i}{i!} e^{-\mu} & \text{if } k \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Input Rate of occurrence, $\mu > 0$

Output The number of successes $k \in \{0, 1, \ldots\}$

Mode $$\begin{cases} \mu - 1 \text{ and } \mu & \text{if } \mu \text{ is an integer} \\ \text{int}(\mu) & \text{otherwise} \end{cases}$$

Mean μ

Variance μ

Algorithm
1. Set $a = e^{-\mu}$, $b = 1$, and $i = 0$.
2. Generate $U_{i+1} \sim \text{U}(0, 1)$ and replace b by bU_{i+1}.
3. If $b < a$, return $X = i$; otherwise, replace i by $i + 1$ and go back to step 2.

Source Code

```
int poisson( double mu ) {
   assert( mu > 0 );
   double b = 1;
   int i;
   for ( i = 0; b >= exp( -mu ); i++ ) b *= uniform( 0, 1 );
   return i - 1;
}
```

Notes
1. The Poisson distribution is the limiting case of the binomial distribution as $n \to \infty$, $p \to 0$, and $np \to \mu$; binomial$(n, p) \approx$ Poisson(μ), where $\mu = np$.
2. For $\mu > 9$, Poisson(μ) may be approximated with N(μ, μ) if we round to the nearest integer and reject negative values.

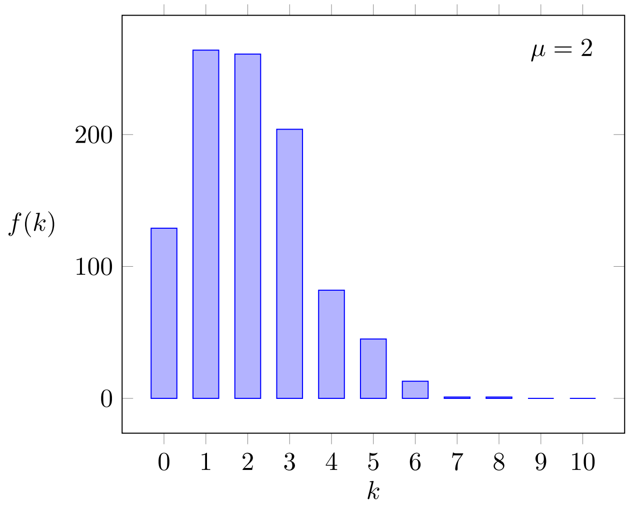

Figure 71. Histogram of Poisson PDF

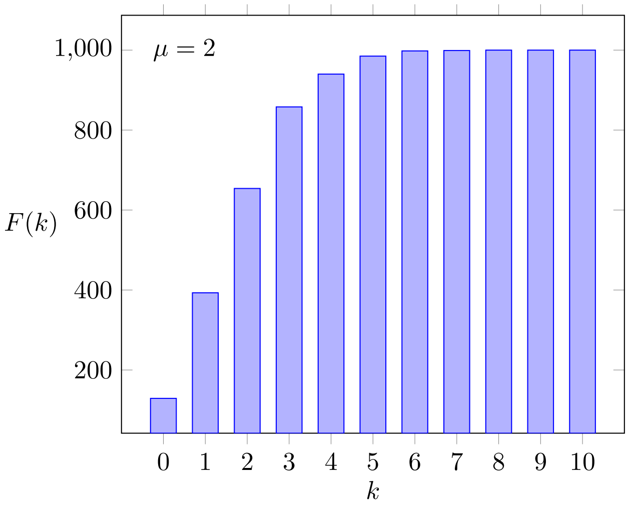

Figure 72. Histogram of Poisson CDF

5.2.9 Uniform Discrete

The Uniform Discrete distribution represents the probability of selecting a particular item from a set of equally-probable items.

Density Function $$f(k) = \begin{cases} \dfrac{1}{i_{\max} - i_{\min} + 1} & k \in \{i_{\min}, \dots, i_{\max}\} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function $$F(k) = \begin{cases} \dfrac{k - i_{\min} + 1}{i_{\max} - i_{\min} + 1} & i_{\min} \le k \le i_{\max} \\ 1 & k \ge i_{\max} \end{cases}$$

Input Minimum integer, $i_{\min}$; maximum integer $i_{\max}$

Output $k \in \{i_{\min}, \dots, i_{\max}\}$

Mode Does not uniquely exist, as all values in the domain are equally probable.

Mean $(i_{\min} + i_{\max})/2$

Variance $[(i_{\max} - i_{\min} + 1)^2 - 1]/12$

Algorithm 1. Generate $U \sim \mathrm{U}(0, 1)$.

2. Return $X = i_{\min} + \mathrm{int}([i_{\max} - i_{\min} + 1]U)$.

Source Code

```
int uniformDiscrete( int i, int j ) {

   assert( i < j );
   return i + int( ( j - i + 1 ) * uniform( 0, 1 ) );
}
```

Notes 1. The distribution uniformDiscrete$(0, 1)$ is the same as bernoulli$(1/2)$.

2. Uniform Discrete is the discrete analog of the continuous Uniform distribution.

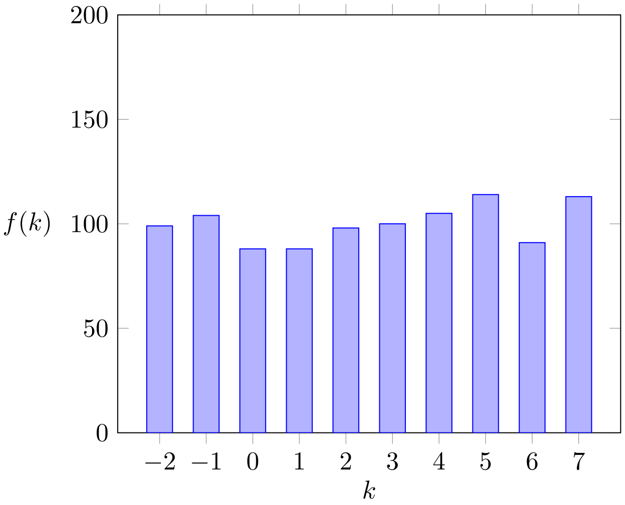

Figure 73. Histogram of uniform discrete PDF

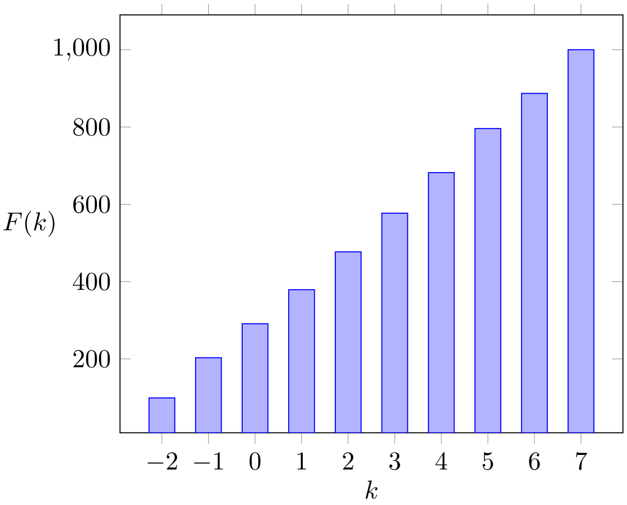

Figure 74. Histogram of uniform discrete CDF

5.3 Empirical and Data–Driven Distributions

The empirical and data-driven distributions make use of one or more of the following parameters:

x	–	data point in a continuous distribution.
F	–	cumulative distribution function for a continuous distribution.
k	–	data point in a discrete distribution.
p	–	probability value at a discrete data point for a discrete distribution.
P	–	cumulative probability for a discrete distribution.

To aid in selecting an appropriate distribution, Table 3 summarizes some characteristics of these distributions. The subsections that follow describe each distribution in more detail.

Table 3. Parameters and Description for Selecting the Appropriate Empirical Distribution

Distribution Name	Input	Output
Empirical	file of (x_i, F_i)	interpolated data point x
Empirical Discrete	file of (k_i, p_i) data pairs	selection of a data point k
Sampling with and without Replacement	file of k_i data	selection of a data point k
Stochastic Interpolation	file of 2-D data points (x_i, y_i)	new 2-D data point (x, y)

5.3.1 Empirical

Distribution Function

The distribution function is specified at a number of distinct data points and is linearly interpolated at other points:

$$F(x) = F(x_i) + [F(x_{i+1}) - F(x_i)]\frac{x - x_i}{x_{i+1} - x_i} \qquad \text{for } x_i < x < x_{i+1}$$

where x_i, $i = 0, 1, \ldots n$ are the data points, and $F(x_i)$ is the fractional number of observed data points less than x_i.

Input

We assume that the empirical data is in the form of a histogram of $n + 1$ pairs of data points along with the corresponding cumulative probability value:

$$\begin{array}{cc} x_0 & F(x_0) \\ \vdots & \vdots \\ x_n & F(x_n) \end{array}$$

where $i = 0, 1, \ldots n$, $F(x_0) = 0$, $F(x_n) = 1$, and $F(x_i) < F(x_{i+1})$. The data points must be in *ascending order* but need not be equally spaced.

Output

$x \in [x_0, x_n)$

Algorithm

This algorithm works by the inverse transform method.

1. Generate $U \sim \mathrm{U}(0, 1)$.
2. Locate index i such that $F(x_i) \le U < F(x_{i+1})$.
3. Return $X = x_i + \dfrac{U - F(x_i)}{F(x_{i+1}) - F(x_i)}(x_{i+1} - x_i)$.

Source Code

```
double Random::empirical( void ) {

   static vector< double > x, cdf;
   static int              n;
   static bool             init = false;

   if ( !init ) {
      ifstream in( "empiricalDistribution" );
      if ( !in ) {
         cerr << "Cannot open \\"empiricalDistribution\\" file" << endl;
         exit( 1 );
      }
      double value, prob;
      while ( in >> value >> prob ) {   // read in empirical data
         x.push_back( value );
         cdf.push_back( prob );
      }
      n = x.size();
      init = true;

      // check that this is indeed a cumulative distribution

      for ( int i = 1; i < n; i++ )
         assert( cdf[ i - 1 ] < cdf[ i ] );
      assert( cdf[ n - 1 ] == 1.0 );
   }

   double p = uniform( 0., 1. );
   for ( int i = 0; i < n - 1; i++ )
      if ( cdf[ i ] <= p && p < cdf[ i + 1 ] )
         return x[ i ] + ( x[ i + 1 ] - x[ i ] ) * ( p - cdf[ i ] ) /
                         ( cdf[ i + 1 ] - cdf[ i ] );
   return x[ n - 1 ];
}
```

Notes

1. The data must reside in a file named `empiricalDistribution`.
2. The number of data pairs in the file is arbitrary (and is not a required input, as the code dynamically allocates the memory required).

5.3.2 Empirical Discrete

Density Function

This is specified by a list of data pairs, (k_i, p_i), where each pair consists of an integer data point, k_i, and the corresponding probability value, p_i.

Distribution Function

$F(k_j) = \sum_{i=1}^{j} p_i = P_j$.

Input

Data pairs (k_i, p_i), where $i = 1, 2, \dots, n$. The data points must be in *ascending order* by data point but need not be equally spaced and the probabilities must sum to one:

$$k_i < k_j \text{ if and only if } i < j \text{ and } \sum_{i=1}^{n} p_i = 1.$$

Output

$x \in \{k_1, k_2, \dots, k_n\}$

Algorithm

1. Generate $U \sim \mathrm{U}(0, 1)$.

2. Locate index j such that $\sum_{i=1}^{j-1} p_i \le U < \sum_{i=1}^{j} p_i$.

3. Return $X = k_j$.

Source Code

```
int empiricalInt( void ) {

   static vector< int >    k;
   static vector< double > f[ 2 ];   // pdf is f[ 0 ] and cdf is f[ 1 ]
   static double           max;
   static int              n;
   static bool             init = false;

   if ( !init ) {
      ifstream in ( "empiricalDiscrete" );
      if ( !in ) {
         cerr << "Cannot open \\"empiricalDiscrete\\" file" << endl;
         exit( 1 );
      }
      int value;
      double freq;
      while ( in >> value >> freq ) {   // read in empirical data
         k.push_back( value );
         f[ 0 ].push_back( freq );
      }
      n = k.size();
      init = true;

      // form the cumulative distribution

      f[ 1 ].push_back( f[ 0 ][ 0 ] );
      for ( int i = 1; i < n; i++ )
         f[ 1 ].push_back( f[ 1 ][ i - 1 ] + f[ 0 ][ i ] );

      // check that the integer points are in ascending order and that
      // the cumulative distribution has a maximum in the interval (0,1]

      for ( int i = 1; i < n; i++ ) assert( k[ i - 1 ] < k[ i ] );
      assert( 0. < f[ 1 ][ n - 1 ] && f[ 1 ][ n - 1 ] <= 1. );

      max = f[ 1 ][ n - 1 ];
   }

   // select a uniform random number between 0 and the maximum value
   // of the cumulative distribution

   double p = uniform( 0., max );

   // locate and return the corresponding index

   for ( int i = 0; i < n; i++ ) if ( p <= f[ 1 ][ i ] ) return k[ i ];
   return k[ n - 1 ];
}
```

Notes

1. The data must reside in a file named `empiricalDiscrete`.
2. The number of data pairs in the file is arbitrary (and is not a required input, as the code dynamically allocates the memory required).

5.3.3 Sampling with and without Replacement

Suppose a population of size N contains K items having some attribute in common. We want to know the probability of getting exactly k items with this attribute in a sample size of n, where $0 \le k \le n$. Sampling *with replacement* effectively makes each sample independent and the probability is given by the formula

$$P(k) = \binom{n}{k} \frac{K^k (N-K)^{n-k}}{N^n}, \quad \text{where} \quad \binom{n}{k} \equiv \frac{n!}{k!(n-k)!}.$$

(See the binomial distribution for comparison.) Let the data be represented by $\{x_1, x_2, \ldots, x_N\}$. Then an algorithm for sampling with replacement is as follows:
1. Generate index $i \sim \text{UniformDiscrete}(1, N)$.
2. Return data element x_i.
And, in the case of sampling *without replacement*, the probability is given by the formula

$$P(k, n) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}.$$

(See the hypergeometric distribution for comparison.) An algorithm for this case is as follows:
1. Perform a random shuffle of the data points $\{x_1, x_2, \ldots, x_N\}$. (See section 3.4.2 of Knuth[1969].)
2. Store the shuffled data in a vector.
3. Retrieve data by sequentially indexing the vector.
The following code implements both methods—i.e., sampling with and without replacement.

```cpp
double sample( bool replace = true ) { // Sample w or w/o replacement from a
                                       // distribution of 1-D data in a file
   static vector< double > v;          // vector for sampling with replacement
   static bool init = false;           // flag that file has been read in
   static int n;                       // number of data elements in the file
   static int index = 0;               // subscript in the sequential order

   if ( !init ) {
      ifstream in( "sampleData" );
      if ( !in ) {
         cerr << "Cannot open \"sampleData\" file" << endl;
         exit( 1 );
      }
      double d;
      while ( in >> d ) v.push_back( d );
      in.close();
      n = v.size();
      init = true;
      if ( replace == false ) {   // sample without replacement

         // shuffle contents of v once and for all
         // Ref: Knuth, D. E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms. London: Addison-Wesley, 1969.
         for ( int i = n - 1; i > 0; i-- ) {
            int j = int( ( i + 1 ) * _u() );
            swap( v[ i ], v[ j ] );
         }
      }
   }

   // return a random sample
   if ( replace )                                // sample w/ replacement
      return v[ uniformDiscrete( 0, n - 1 ) ];
   else {                                        // sample w/o replacement
      assert( index < n );                       // retrieve elements
      return v[ index++ ];                       // in sequential order
   }
}
```

5.3.4 Stochastic Interpolation

Sampling (with or without replacement) can only return some combination of the original data points. Stochastic interpolation is a more sophisticated technique that will generate new data points. It is designed to give the new data the same local statistical properties as the original data and is based on the following algorithm.

1. Translate and scale multivariate data so that each dimension has the same range:

$$\mathbf{x} \Rightarrow \frac{\mathbf{x} - \mathbf{x}_{\min}}{\mathbf{x}_{\max} - \mathbf{x}_{\min}}.$$

2. Randomly select (with replacement) one of the n data points along with its nearest $m - 1$ neighbors $\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_{m-1}$ and compute the sample mean:

$$\bar{\mathbf{x}} \;=\; \frac{1}{m}\sum_{i=1}^{m}\mathbf{x}_i.$$

3. Generate m IID uniform variates

$$U_i \sim \mathrm{U}\left(\frac{1-\sqrt{3(m-1)}}{m}, \frac{1+\sqrt{3(m-1)}}{m}\right)$$

and set

$$\mathbf{X} = \bar{\mathbf{x}} + \sum_{i=1}^{m}(\mathbf{x}_i - \bar{\mathbf{x}})U_i.$$

4. Rescale $\mathbf{X}$ by $(\mathbf{x}_{\max} - \mathbf{x}_{\min})$ and shift to $\mathbf{x}_{\min}$.

The following code implements both methods—i.e., sampling with and without replacement.

```
struct dSquared : public binary_function< point, point, bool > {
   bool operator()( point p, point q ) {
      return p.x * p.x + p.y * p.y < q.x * q.x + q.y * q.y;
   }
};

point stochasticInterpolation( void ) {

// Refs: Taylor, M. S. and J. R. Thompson, Computational Statistics & Data
//       Analysis, Vol. 4, pp. 93-101, 1986; Thompson, J. R., Empirical Model
//       Building, pp. 108-114, Wiley, 1989; Bodt, B. A. and M. S. Taylor,
//       A Data Based Random Number Generator for A Multivariate Distribution -
//       A User's Manual, ARBRL-TR-02439, BRL, APG, MD, Nov. 1982.

   static vector< point > data;
   static point           min, max;
   static int             m;
   static double          lower, upper;
   static bool            init = false;

   if ( !init ) {
      ifstream in( "stochasticData" );
      if ( !in ) {
         cerr << "Cannot open \\"stochasticData\\" input file" << endl;
         exit( 1 );
      }

      // read in the data and set min and max values

      min.x = min.y = FLT_MAX;
      max.x = max.y = FLT_MIN;
      point p;
      while ( in >> p.x >> p.y ) {

         min.x = ( p.x < min.x ? p.x : min.x );
         min.y = ( p.y < min.y ? p.y : min.y );
         max.x = ( p.x > max.x ? p.x : max.x );
         max.y = ( p.y > max.y ? p.y : max.y );

         data.push_back( p );
      }
      in.close();
      init = true;

      // scale the data so that each dimension will have equal weight

      for ( int i = 0; i < data.size(); i++ ) {

         data[ i ].x = ( data[ i ].x - min.x ) / ( max.x - min.x );
         data[ i ].y = ( data[ i ].y - min.y ) / ( max.y - min.y );
      }

      // set m, the number of points in a neighborhood of a given point

      m = data.size() / 20;       // 5% of all the data points
      if ( m < 5  ) m = 5;        // but no less than 5
      if ( m > 20 ) m = 20;       // and no more than 20

      lower = ( 1. - sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
      upper = ( 1. + sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
   }

   // uniform random selection of a data point (with replacement)

   point origin = data[ uniformInt( 0, data.size() - 1 ) ];

   // make this point the origin of the coordinate system

   for ( int n = 0; n < data.size(); n++ ) data[ n ] -= origin;

   // sort the data with respect to its distance (squared) from this origin

   sort( data.begin(), data.end(), dSquared() );

   // find the mean value of the data in the neighborhood about this point

   point mean;
   mean.x = mean.y = 0.;
   for ( int n = 0; n < m; n++ ) mean += data[ n ];
   mean /= double( m );

   // select a random linear combination of the points in this neighborhood

   point p;
   p.x = p.y = 0.;
   for ( int n = 0; n < m; n++ ) {

      double rn;
      if ( m == 1 ) rn = 1.;
      else          rn = uniform( lower, upper );

      p.x += rn * ( data[ n ].x - mean.x );
      p.y += rn * ( data[ n ].y - mean.y );
   }

   // restore the data to its original form

   for ( int n = 0; n < data.size(); n++ ) data[ n ] += origin;

   // use the mean and the original point to translate the randomly-chosen point

   p += mean;
   p += origin;

   // scale the randomly-chosen point to the dimensions of the original data

   p.x = p.x * ( max.x - min.x ) + min.x;
   p.y = p.y * ( max.y - min.y ) + min.y;

   return p;
}
```

Notes: 1. Notice that the particular range on the uniform distribution in step 3 of the algorithm is chosen to give a mean value of $1/m$ and a variance of $(m-1)/m^2$.
2. When $m = 1$, this reduces to the bootstrap method of sampling with replacement.

5.4 Bivariate Distributions

The bivariate distributions described in this section make use of one or more of the following parameters:

`cartesianCoord` – a Cartesian point (x, y) in two dimensions.

`polarCoord` – a point (r, θ) in two dimensions in polar coordinates.

`sphericalCoord` – the angles (θ, ϕ), where θ is the polar angle as measured from the z-axis, and ϕ is the azimuthal angle as measured counterclockwise from the x-axis.

ρ – correlation coefficient, where $-1 \leq \rho \leq 1$.

To aid in selecting an appropriate distribution, Table 4 summarizes some characteristics of these distributions. The subsections that follow describe each distribution in more detail.

Table 4. Description and Output for Selecting the Appropriate Bivariate Distribution

Distribution Name	**Description**	**Output**
Bivariate Normal	normal distribution in two dimensions	`cartesianCoord`
Bivariate Uniform	uniform distribution in two dimensions	`cartesianCoord`
Correlated Normal	normal distribution in two dimensions with correlation	`cartesianCoord`
Correlated Uniform	uniform distribution in two dimensions with correlation	`cartesianCoord`
Circular Uniform	uniform distribution over the unit circle	`polarCoord`
Spherical Uniform	uniform distribution over the surface of the unit sphere	`sphericalCoord`
SphericalND	uniform distribution over the surface of the N-D unit sphere	`sphericalCoord`

5.4.1 Bivariate Normal

Density Function

$$f(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left\{ -\left[\frac{(x-\mu_x)^2}{2\sigma_x^2} + \frac{(y-\mu_y)^2}{2\sigma_y^2} \right] \right\}$$

Input

Location parameters (μ_x, μ_y), any real numbers;
scale parameters (σ_x, σ_y), any positive numbers.

Output

$x \in (-\infty, \infty)$ and $y \in (-\infty, \infty)$

Mode

(μ_x, μ_y)

Variance

(σ_x^2, σ_y^2)

Algorithm

1. Independently generate $X \sim \mathrm{N}(0, 1)$ and $Y \sim \mathrm{N}(0, 1)$.
2. Return $(\mu_x + \sigma_x X, \mu_y + \sigma_y Y)$.

Source Code

```
std::pair<double,double> bivariateNormal( double muX, double sigmaX,
                                          double muY, double sigmaY ) {

   assert( sigmaX > 0 && sigmaY > 0 );

   return std::make_pair( normal( muX, sigmaX ), normal( muY, sigmaY ) );
}
```

Notes

The variables are assumed to be uncorrelated. For correlated variables, use the correlated normal distribution.

Two examples of the distribution of points obtained via calls to this function are shown in Figs. 75 and 76.

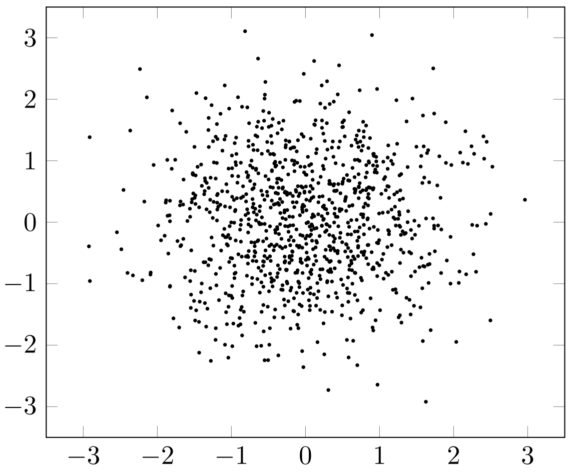

Figure 75. bivariateNormal(0, 1, 0, 1)

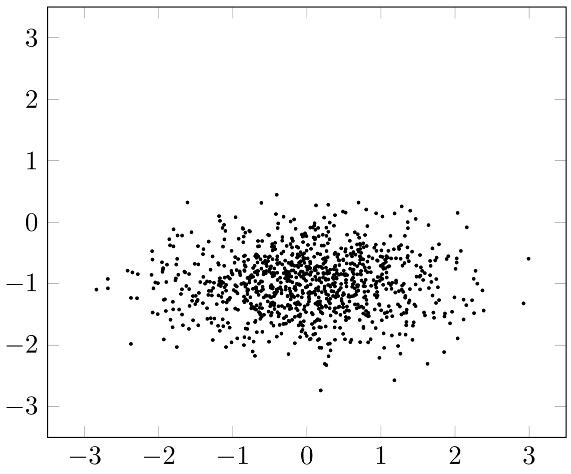

Figure 76. bivariateNormal(0, 1, -1, 0.5)

5.4.2 Bivariate Uniform

Density Function

$$f(x,y) = \begin{cases} \dfrac{1}{\pi ab} & 0 \le \dfrac{(x-x_0)^2}{a^2} + \dfrac{(y-y_0)^2}{b^2} \le 1 \\ 0 & \text{otherwise} \end{cases}$$

Input

$[x_{\min}, x_{\max})$, bounds along x-axis; $[y_{\min}, y_{\max})$, bounds along y-axis; Location parameters (x_0, y_0), where $x_0 = (x_{\min}+x_{\max})/2$ and $y_0 = (y_{\min}+y_{\max})/2$; scale parameters (a, b), where $a = (x_{\max} - x_{\min})/2$ and $b = (y_{\max} - y_{\min})/2$ are derived.

Output

Point (x, y) inside the ellipse bounded by the rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$

Algorithm

1. Independently generate $X \sim \mathrm{U}(-1, 1)$ and $Y \sim \mathrm{U}(-1, 1)$.
2. If $X^2 + Y^2 > 1$, go back to step 1; otherwise go to step 3.
3. Return $(x_0 + aX, y_0 + bY)$.

Source Code

```
std::pair<double,double> bivariateUniform( double xMin, double xMax,
                                           double yMin, double yMax ) {

   assert( xMin < xMax && yMin < yMax );
   double x0 = 0.5 * ( xMin + xMax );
   double y0 = 0.5 * ( yMin + yMax );
   double a  = 0.5 * ( xMax - xMin );
   double b  = 0.5 * ( yMax - yMin );
   double x, y;

   do {
      x = uniform( -1, 1 );
      y = uniform( -1, 1 );
   } while ( x * x + y * y > 1 );

   return std::make_pair( x0 + a * x, y0 + b * y );
}
```

Notes

Another choice is to use a bounding rectangle instead of a bounding ellipse.

Two examples of the distribution of points obtained via calls to this function are shown in Figs. 77 and 78.

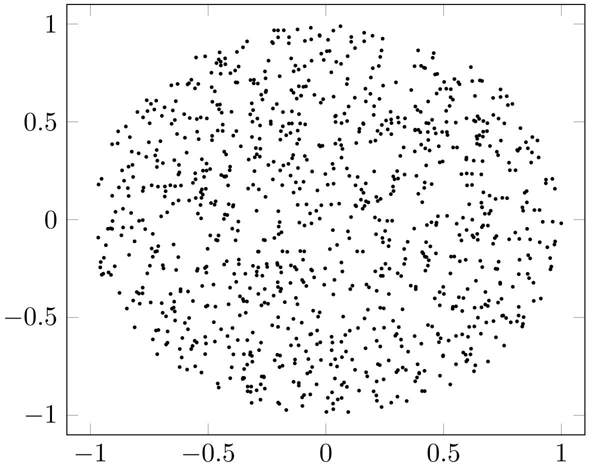

Figure 77. bivariateUniform(0, 1, 0, 1)

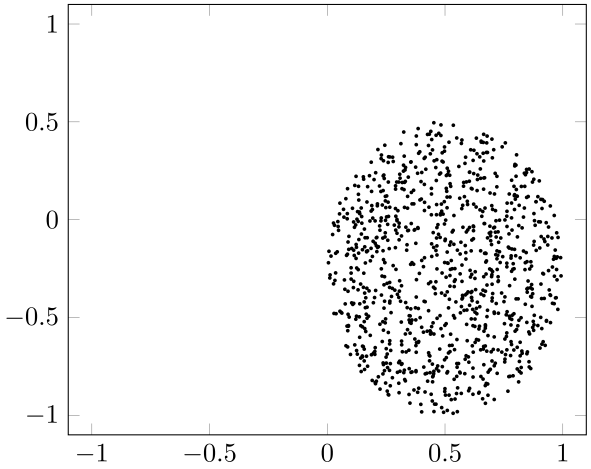

Figure 78. bivariateUniform(0, 1, -1, 0.5)

5.4.3 Circular Uniform

Density Function

$$f(r,\theta) = \begin{cases} \dfrac{2}{(r_{\max}^2 - r_{\min}^2)(\theta_{\max} - \theta_{\min})} & 0 \le r_{\min} \le r_{\max} \text{ and } \theta_{\min} \le \theta_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function

$$F(r,\theta) = \begin{cases} \dfrac{(r^2 - r_{\min}^2)(\theta - \theta_{\min})}{(r_{\max}^2 - r_{\min}^2)(\theta_{\max} - \theta_{\min})} & r_{\min} \le r \le r_{\max} \text{ and } \theta_{\min} \le \theta \le \theta_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Input

$[r_{\min}, r_{\max})$, bounds for the radius; $[\theta_{\min}, \theta_{\max})$, bounds for the polar angle.

Output

Point (r, θ) in polar coordinates

Algorithm

1. Independently generate $R \sim \sqrt{\mathrm{U}(r_{\min}^2, r_{\max}^2)}$ and $\Theta \sim \mathrm{U}(\theta_{\min}, \theta_{\max})$.
2. Return (R, Θ).

Source Code

```
std::pair<double,double> circularUniform( double rMin, double rMax,
                                          double thMin, double thMax ) {

   assert( 0 <= rMin && rMin <= rMax && thMin <= thMax );

   double r = sqrt( uniform( rMin * rMin, rMax * rMax ) );
   double th = uniform( thMin, thMax );

   return make_pair( r, th );
}
```

Notes

Unlike the bivariateUniform, which uses acceptance-rejection, this is a direct method of achieving circular uniform.

Two examples of the distribution of points obtained via calls to this function are shown in Figs. 79 and 80.

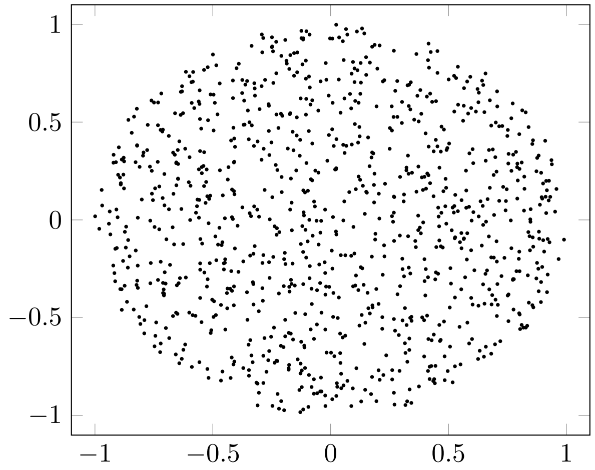

Figure 79. circularUniform()

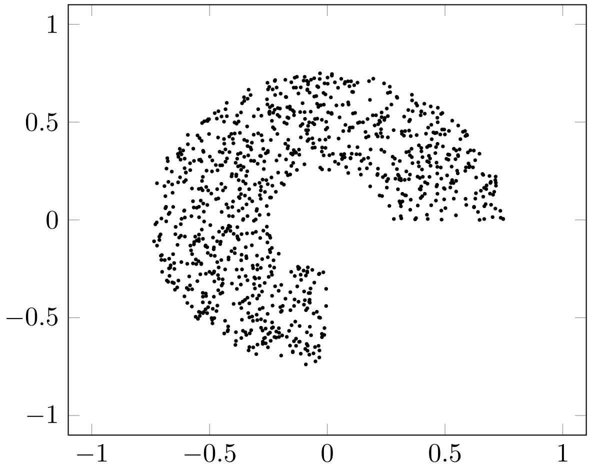

Figure 80. circularUniform(0.25, 0.75, 0, 270 * M_PI / 180)

5.4.4 Correlated Normal

Density Function

$$f(x,y) = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}} \exp\left\{-\frac{1}{1-\rho^2}\left[\frac{(x-\mu_x)^2}{2\sigma_x^2} - \frac{\rho(x-\mu_x)(y-\mu_y)}{\sigma_x\sigma_y} + \frac{(y-\mu_y)^2}{2\sigma_y^2}\right]\right\}$$

Input

Location parameters (μ_x, μ_y), any real numbers; scale parameters (σ_x, σ_y), any positive numbers;
correlation coefficient, $-1 \le \rho \le 1$.

Output: Point (x, y), where $x \in (-\infty, \infty)$ and $y \in (-\infty, \infty)$

Mode: (μ_x, μ_y)

Variance: (σ_x^2, σ_y^2)

Correlation Coefficient: ρ

Algorithm:
1. Independently generate $X \sim \mathrm{N}(0,1)$ and $Z \sim \mathrm{N}(0,1)$.
2. Set $Y = \rho X + \sqrt{1-\rho^2}Z$.
2. Return $(\mu_x + \sigma_x X, \mu_y + \sigma_y Y)$.

Source Code

```
std::pair<double,double> corrNormal( double r, double muX, double sigmaX,
                                               double muY, double sigmaY ) {

   assert( -1 <= r && r <= 1 );
   assert( sigmaX > 0 && sigmaY > 0 );

   double x = normal(0,1);
   double y = normal(0,1);

   y = r * x + sqrt( 1 - r * r ) * y;

   return std::make_pair( muX + sigmaX * x, muY + sigmaY * y );
}
```

Notes: This reduces to the bivariate normal distribution when $\rho = 0$.

Two examples of the distribution of points obtained via calls to this function are shown in Figs. 81 and 82.

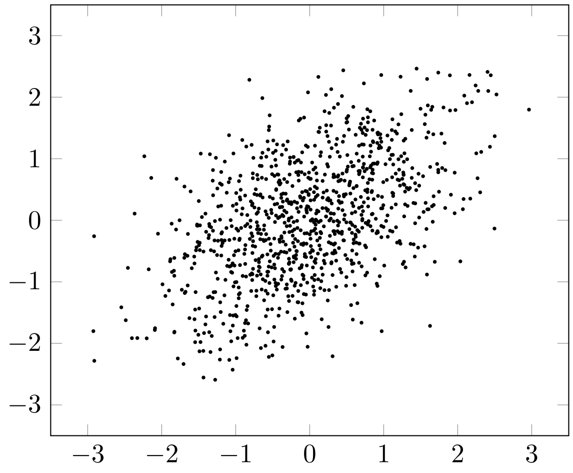

Figure 81. corrNormal(0.5, 0, 1, 0, 1)

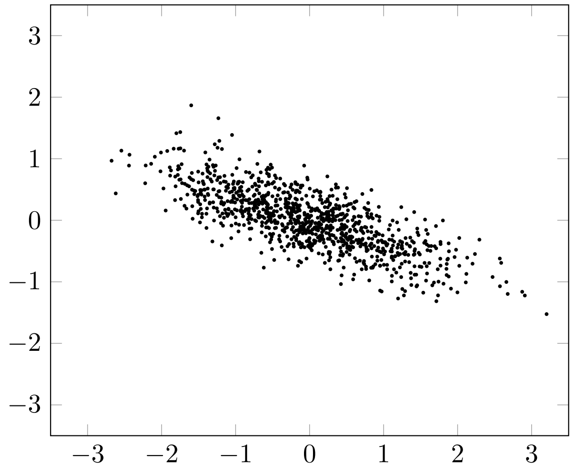

Figure 82. corrNormal(-0.75, 0, 1, 0, 0.5)

5.4.5 Correlated Uniform

Input

ρ, correlation coefficient, where $-1 \le \rho \le 1$;
$[x_{\min}, x_{\max})$, bounds along x-axis; $[y_{\min}, y_{\max})$, bounds along y-axis;
Location parameters (x_0, y_0), where $x_0 = (x_{\min}+x_{\max})/2$ and $y_0 = (y_{\min}+y_{\max})/2$; scale parameters (a, b), where $a = (x_{\max} - x_{\min})/2$ and $b = (y_{\max} - y_{\min})/2$ are derived.

Output

Correlated points (x, y) inside the ellipse bounded by the rectangle $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$

Algorithm

1. Independently generate $X \sim \mathrm{U}(-1, 1)$ and $Z \sim \mathrm{U}(-1, 1)$.
2. If $X^2 + Z^2 > 1$, go back to step 1; otherwise go to step 3.
3. Set $Y = \rho X + \sqrt{1 - \rho^2} Z$.
3. Return $(x_0 + aX, y_0 + bY)$.

Source Code

```
std::pair<double,double> corrUniform( double r, double xMin, double xMax,
                                                double yMin, double yMax ) {

   assert( -1 <= r && r <= 1 );
   assert( xMin < xMax && yMin < yMax );
   double x0 = 0.5 * ( xMin + xMax );
   double y0 = 0.5 * ( yMin + yMax );
   double a  = 0.5 * ( xMax - xMin );
   double b  = 0.5 * ( yMax - yMin );
   double x, y;

   do {
      x = uniform( -1, 1 );
      y = uniform( -1, 1 );
   } while ( x * x + y * y > 1 );

   y = r * x + sqrt( 1 - r * r ) * y;   // correlate variables

   return std::make_pair( x0 + a * x, y0 + b * y );
}
```

Notes

Another choice is to use a bounding rectangle instead of a bounding ellipse.

Two examples of the distribution of points obtained via calls to this function are shown in Figs. 83 and 84.

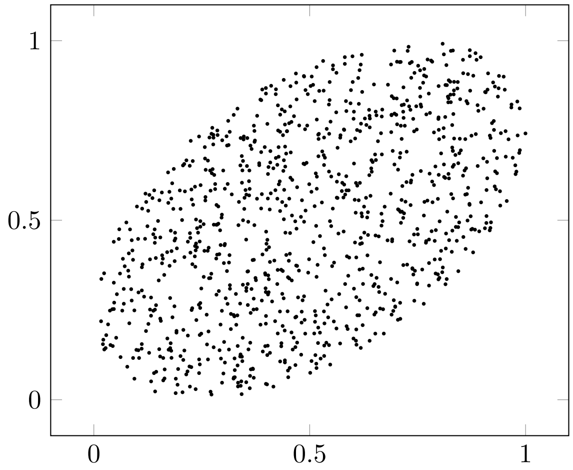

Figure 83. corrUniform(0.5, 0, 1, 0, 1)

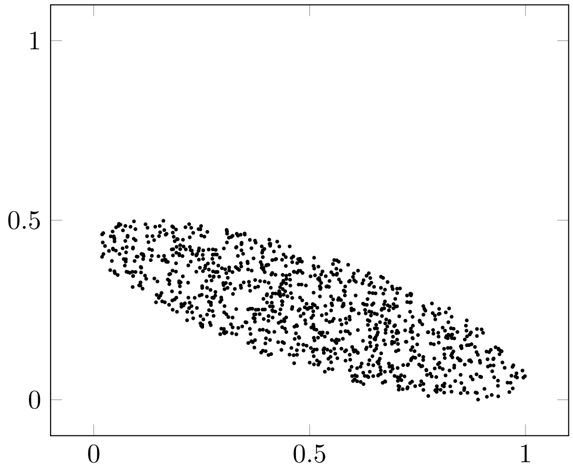

Figure 84. corrUniform(-0.75, 0, 1, 0, 0.5)

5.4.6 Spherical Uniform

Density Function

$$f(\theta,\phi) = \frac{\sin\theta}{(\phi_{\max}-\phi_{\min})(\cos\theta_{\min}-\cos\theta_{\max})} \quad \text{for} \quad \begin{cases} 0 \le \theta_{\min} < \theta < \theta_{\max} \le \pi \\ 0 \le \phi_{\min} < \phi < \phi_{\max} \le 2\pi \end{cases}$$

Distribution Function

$$F(\theta,\phi) = \frac{(\phi-\phi_{\min})(\cos\theta_{\min}-\cos\theta)}{(\phi_{\max}-\phi_{\min})(\cos\theta_{\min}-\cos\theta_{\max})} \quad \text{for} \quad \begin{cases} 0 \le \theta_{\min} < \theta < \theta_{\max} \le \pi \\ 0 \le \phi_{\min} < \phi < \phi_{\max} \le 2\pi \end{cases}$$

Input

minimum polar angle, $\theta_{\min} \ge 0$; maximum polar angle, $\theta_{\max} \le \pi$;
minimum azimuthal angle, $\phi_{\min} \ge 0$; maximum azimuthal angle, $\phi_{\max} \le 2\pi$

Output

(θ, ϕ) pair, where $\theta \in [\theta_{\min}, \theta_{\max}]$ and $\phi \in [\phi_{\min}, \phi_{\max}]$

Mode

Does not uniquely exist, as angles are uniformly distributed over the unit sphere

Mean

$((\theta_{\min}+\theta_{\max})/2, (\phi_{\min}+\phi_{\max})/2)$

Variance

$((\theta_{\max}-\theta_{\min})^2/12, (\phi_{\max}-\phi_{\min})^2/12)$

Algorithm

1. Independently generate $U_1 \sim \mathrm{U}(\cos\theta_{\max}, \cos\theta_{\min})$ and $U_2 \sim \mathrm{U}(\phi_{\min}, \phi_{\max})$.
2. Return $(\Theta, \Phi) = (\cos^{-1}(U_1), U_2)$.

Source Code

```
std::pair<double,double> spherical( double thMin, double thMax,
                                    double phMin, double phMax ) {

   assert( 0 <= thMin && thMin < thMax && thMax <= M_PI &&
           0 <= phMin && phMin < phMax && phMax <= 2 * M_PI );

   return std::make_pair( acos( uniform( cos( thMax ), cos( thMin ) ) ),
                          uniform( phMin, phMax ) );
}
```

Fig. 85 shows the uniform random distribution of 1000 points on the surface of the unit sphere obtained via repeated calls to this function.

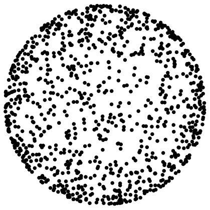

Figure 85. Uniform spherical distribution via calls to spherical()

5.4.7 Spherical Uniform in N-Dimensions

The following algorithm will generate uniformly-distributed points on the surface of the unit sphere in n dimensions. Whereas the spherical uniform distribution is designed to return the *angles* of the points on the surface of the three-dimensional unit sphere, this distribution returns the *Cartesian coordinates* of the points and will work for an arbitrary number of dimensions.

Input — Vector $\mathbf{X}$ to receive values; number of dimensions, n

Output — Vector $\mathbf{X}$ of unit length (i.e., $X_1^2 + X_2^2 + \cdots + X_n^2 = 1$)

Algorithm

1. Generate n IID normal variates $X_1, X_2, \ldots, X_n \sim \mathrm{N}(0, 1)$.
2. Compute the distance from the origin, $d = \sqrt{X_1^2 + X_2^2 + \cdots + X_n^2}$.
2. Return $\mathbf{X}/d$, which now has unit length.

Source Code

```
void sphericalND( double x[],    // x array returns point
                  int    n ) {    // n is number of dimensions

// generate a point inside the unit n-sphere by normal polar method

   double r2 = 0.;
   for ( int i = 0; i < n; i++ ) {
      x[ i ] = normal();
      r2 += x[ i ] * x[ i ];
   }

// project the point onto the surface of the n-sphere by scaling

   const double A = 1. / sqrt( r2 );
   for ( int i = 0; i < n; i++ ) x[ i ] *= A;
}
```

Notes

1. When $n = 1$, this algorithm returns $\{-1, +1\}$.
2. When $n = 2$, it generates points on the unit circle.
3. When $n = 3$, it generates points on the unit 3-sphere.

5.5 Distributions Generated From Number Theory

This section contains two recipes for generating pseudo-random numbers through the application of number theory:[*]

5.5.1 Tausworthe Random Bit Generator

Very fast random bit generators have been developed based on the theory of *Primitive Polynomials Modulo Two* (Tausworthe 1965). These are polynomials of the form

$$P_n(x) = (x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0) \pmod 2,$$

where n is the order and each coefficient a_i is either 1 or 0. The polynomials are *prime* in the sense that they cannot be factored into lower order polynomials and they are *primitive* in the sense that the recurrence relation

$$a_n = (x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0) \pmod 2$$

will generate a string of 1s and 0s that has a maximal cycle length of $2^n - 1$ (i.e., all possible values excluding the case of all zeroes). Primitive polynomials of order n from 1 to 100 have been tabluated (Watson 1962). Since the truth table of integer addition modulo 2 is the same as "exclusive or" (XOR), it is very easy to implement these recurrence relations in computer code. And, using the separate bits of a computer word to store a primitive polynomial allows us to deal with polynomials up to order 32, to give cycle lengths up to $2^{32} - 1 = 4{,}294{,}967{,}295$.

The following code is overloaded in the C++ sense that there are actually two versions of this random bit generator. The first one will return a bit vector of length n, and the second version will simply return a single random bit. Both versions are guaranteed to have a cycle length of $2^n - 1$.

Input

Random number seed (not zero), order n, where $1 \le n \le 32$, and, for the first version, an array to hold the bit vector

Output

Bit vector of length n *or* a single bit (i.e., 1 or 0)

Source Code

```
void tausworthe( bool* bitvec, unsigned n ) {   // returns bit vector of length n

// It is guaranteed to cycle through all possible combinations of n bits
// (except all zeros) before repeating, i.e., cycle is of maximal length 2^n-1.
// Ref: Press, W. H., B. P. Flannery, S. A. Teukolsky and W. T. Vetterling,
//      Numerical Recipes in C, Cambridge Univ. Press, Cambridge, 1988.

   assert( 1 <= n && n <= 32 );   // length of bit vector

   if ( _seed2 & BIT[ n ] )
      _seed2 = ( ( _seed2 ^ MASK[ n ] ) << 1 ) | BIT[ 1 ];
   else
      _seed2 <<= 1;
   for ( int i = 0; i < n; i++ ) bitvec[ i ] = _seed2 & ( BIT[ n ] >> i );
}

bool tausworthe( unsigned n )   // returns a single random bit
{
   assert( 1 <= n && n <= 32 );

   if ( _seed2 & BIT[ n ] ) {
      _seed2 = ( ( _seed2 ^ MASK[ n ] ) << 1 ) | BIT[ 1 ];
      return true;
   }
   else {
      _seed2 <<= 1;
      return false;
   }
}
```

Notes

1. The constants used in the above source code are defined in `Random.h`.
2. This generator is 3.6 times faster than `bernoulli( 0.5 )`.

[*] The theory underlying these techniques is quite involved, but Press et al. (1992) and sources cited therein provide a starting point.

5.5.2 Maximal Avoidance (Quasi-Random)

Maximal avoidance is a technique for generating points in a multidimensional space that are simultaneously self-avoiding, while appearing to be random. For example, the first three plots in Figure 84 show points generated with this technique to demonstrate how they tend to avoid one another. The last plot shows a typical distribution obtained by a uniform random generator, where the clustering of points is apparent.

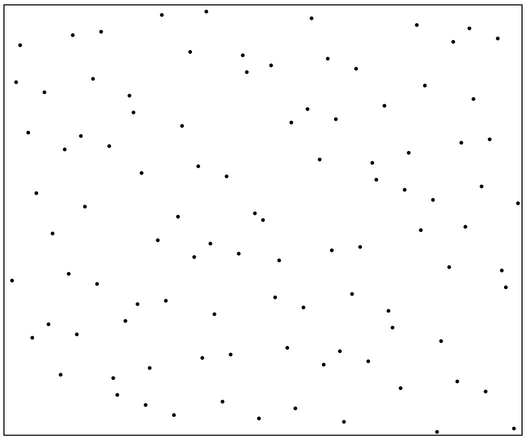

Figure 86. 100 maximal avoidance data points

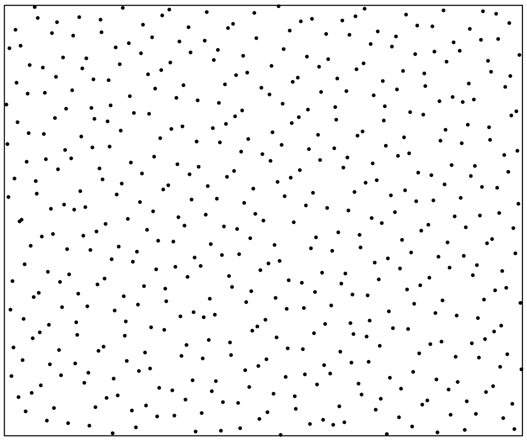

Figure 87. 500 maximal avoidance data points

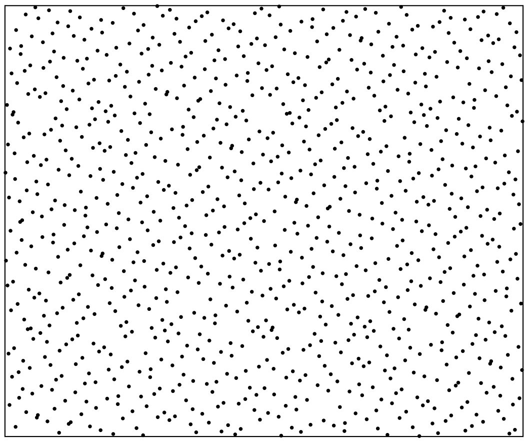

Figure 88. 1000 maximal avoidance data points

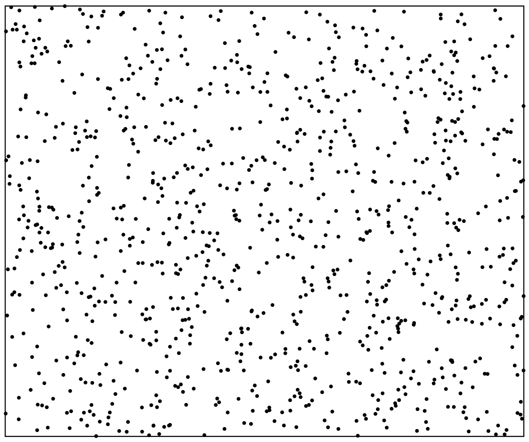

Figure 89. 1000 uniformly distributed data points

The placement of points is actually not pseudo-random at all but rather *quasi-random*, through the clever application of number theory. The theory behind this technique can be found in Press et al. (1992) and the sources cited therein, but we can give a sense of it here. It is somewhat like imposing a Cartesian mesh over the space and then choosing points at the mesh points. By basing the size of the mesh on successive prime numbers and then reducing its spacing as the number of points increases, successive points will avoid one another and tend to fill the space in an hierarchical manner. The actual application is much more involved than this and uses some other techniques (such as primitive polynomials modulo 2, and Gray codes) to make the whole process very efficient. The net result is that it provides a method of sampling a space that represents a compromise between systematic Cartesian sampling and uniform random sampling. Monte Carlo sampling on a Cartesian grid has an error term that decreases faster than $N^{-1/2}$ that one ordinarily gets with uniform random sampling. The drawback is that one needs to know how many Cartesian points to select beforehand. As a consequence, one usually samples uniform randomly until a convergence criterion is met. Maximal avoidance can be considered as the best of both of these techniques. It produces an error term that decreases faster than $N^{-1/2}$ while at the same time providing a mechanism to stop when a tolerance criterion is met. The following code is an implementation of this technique.

```
double avoidance( void ) {   // 1-dimension (overloaded for convenience)

   double x[ 1 ];
   avoidance( x, 1 );
   return x[ 0 ];
}
void avoidance( double x[], int ndim ) {   // multi-dimensional

   static const int MAXBIT = 30;
   static const int MAXDIM = 6;

   assert( ndim <= MAXDIM );
   static unsigned long ix[ MAXDIM + 1 ] = { 0 };
   static unsigned long *u[ MAXBIT + 1 ];
   static unsigned long mdeg[ MAXDIM + 1 ] = { // degree of primitive polynomial
      0, 1, 2, 3, 3, 4, 4
   };
   static unsigned long p[ MAXDIM + 1 ] = {   // decimal encoded interior bits
      0, 0, 1, 1, 2, 1, 4
   };
   static unsigned long v[ MAXDIM * MAXBIT + 1 ] = {
       0,  1,  1, 1,  1,  1,  1,
           3,  1, 3,  3,  1,  1,
           5,  7, 7,  3,  3,  5,
          15, 11, 5, 15, 13,  9
   };
   static double fac;
   static int in = -1;
   int j, k;
   unsigned long i, m, pp;

   if ( in == -1 ) {
      in = 0;
      fac = 1. / ( 1L << MAXBIT );
      for ( j = 1, k = 0; j <= MAXBIT; j++, k += MAXDIM ) u[ j ] = &v[ k ];
      for ( k = 1; k <= MAXDIM; k++ ) {
         for ( j = 1; j <= mdeg[ k ]; j++ ) u[ j ][ k ] <<= ( MAXBIT - j );
         for ( j = mdeg[ k ] + 1; j <= MAXBIT; j++ ) {
            pp = p[ k ];
            i = u[ j - mdeg[ k ] ][ k ];
            i ^= ( i >> mdeg[ k ] );
            for ( int n = mdeg[ k ] - 1; n >= 1; n-- ) {
               if ( pp & 1 ) i ^= u[ j - n ][ k ];
               pp >>= 1;
            }
            u[ j ][ k ] = i;
         }
      }
   }
   m = in++;
   for ( j = 0; j < MAXBIT; j++, m >>= 1 ) if ( !( m & 1 ) ) break;
   if ( j >= MAXBIT ) exit( 1 );
   m = j * MAXDIM;
   for ( k = 0; k < ndim; k++ ) {
      ix[ k + 1 ] ^= v[ m + k + 1 ];
      x[ k ] = ix[ k + 1 ] * fac;
   }
}
```

6 Discussion and Examples

This section presents some example applications in order to illustrate and facilitate the use of the various distributions. Certain distributions, such as the normal and the Poisson, are probably over used and others, due to lack of familiarity, are probably under used. In the interests of improving this situation, the examples make use of the less familiar distributions. Before we present example applications, however, we first discuss some differences between the discrete distributions.

6.1 Making Sense of the Discrete Distributions

Due to the number of different discrete distributions, it can be a little confusing to know when each distribution is applicable. To help mitigate this confusion, let us illustrate the difference between the binomial, geometric, negative binomial, and Pascal distributions. Consider, then, the following sequence of trials, where 1 signifies a success and 0 a failure.

Trial:	1	2	3	4	5	6	7	8
Outcome:	1	0	1	1	1	0	0	1

The binomial(n, p) represents the number of successes in n trials, so it would evaluate as follows:

```
binomial( 1 , p ) = 1
binomial( 2 , p ) = 1
binomial( 3 , p ) = 2
binomial( 4 , p ) = 3
binomial( 5 , p ) = 4
binomial( 6 , p ) = 4
binomial( 7 , p ) = 4
binomial( 8 , p ) = 5
```

The geometric(p) represents the number of failures before the first success. Since we have a success on the first trial, it evaluates as follows:

```
geometric( p ) = 0
```

The negativeBinomial(s, p) represents the number of failures before the sth success in n trials, so it would evaluate as follows:

```
negativeBinomial( 1 , p ) = 0
negativeBinomial( 2 , p ) = 1
negativeBinomial( 3 , p ) = 1
negativeBinomial( 4 , p ) = 1
negativeBinomial( 5 , p ) = 3
```

The pascal(s, p) represents the number of trials in order to achieve s successes, so it would evaluate as follows:

```
pascal( 1 , p ) = 1
pascal( 2 , p ) = 3
pascal( 3 , p ) = 4
pascal( 4 , p ) = 5
pascal( 5 , p ) = 8
```

6.2 Adding New Distributions

We show here how it is possible to extend the list of distributions. Suppose that we want to generate random numbers according to the probability density function shown in Figure 85.
The figure is that of a semi-ellipse, and its equation is

$$f(x) = \frac{2}{\pi}\sqrt{1 - x^2}, \quad \text{where } -1 \le x \le 1. \tag{45}$$

Integrating, we find that the cumulative distribution function is

$$F(x) = \frac{1}{2} + \frac{x\sqrt{1 - x^2} + \sin^{-1}(x)}{\pi}, \quad \text{where } -1 \le x \le 1. \tag{46}$$

Now, this expression involves trancendental functions in a nonalgebraic way, which precludes inverting. But, we can still use the acceptance-rejection method to turn this into a random number generator. We have to do two things:

1. Define a function that returns a value for y, given a value for x.

2. Define a circular distribution that passes the function pointer to the *User-Specified* distribution.

Here is the resulting source code in a form suitable for inclusion in the Random class.

```
double ellipse( double x, double, double ) {   // Ellipse Function

   return sqrt( 1. - x * x ) / M_PI_2;
}

double Random::elliptical( void ) {   // Elliptical Distribution

   const double X_MIN = -1.;
   const double X_MAX = 1.;
   const double Y_MIN = 0.;
   const double Y_MAX = 1. / M_PI_2;

   return userSpecified( ellipse, X_MIN, X_MAX, Y_MIN, Y_MAX );
}
```

And here is source code to make use of this distribution:

```
#include "Random.h"
#include <iostream>

int main( void ) {

   rng::Random rng;
   for ( int i = 0; i < 1000; i++ ) std::cout << rng.elliptical() << std::endl;

   return 0;
}
```

7 Comparison of the Generators

Tables 5, 6, and 7 show how well the generators perform.

Table 5. Performance of RNGs (in Millions/s)

Generator	32-bit unsigned ints	32-bit doubles	64-bit unsigned ints	64-bit long doubles
kiss	228	161	70	57
jkiss	224	158	70	56
jlkiss	223	168	111	84
jlkiss64	223	167	98	82
lfsr88	223	146	73	60
lfsr113	186	131	63	53
lfsr258	184	123	98	76

The kiss family of generators are producing a 32-bit int every 4.5 nanoseconds and are just as fast as the linear feedback shift registers.

Table 6. Results from TestU01 battery of tests

Generator	Small Crush	Crush	Big Crush
kiss	All tests were passed	Failed Permutation and RandomWalk1	Failed RandomWalk1
jkiss	All tests were passed	All tests were passed	All tests were passed
jlkiss	All tests were passed	All tests were passed	All tests were passed
jlkiss64	All tests were passed	All tests were passed	All tests were passed
lfsr88	All tests were passed	Failed MatrixRank and LinearComp	Failed MatrixRank and LinearComp
lfsr113	All tests were passed	Failed MatrixRank and LinearComp	Failed MatrixRank and LinearComp
lfsr258	All tests were passed	Failed MatrixRank and LinearComp	Failed MatrixRank, LinearComp & RandomWalk1

Table 7. Cycle Length and Jump Time

Generator	Approximate Cycle Length	Time to Jump 2^{59}	Time to Jump a Full Cycle
kiss	$2^{124} \approx 10^{37}$	0.000255	0.011 sec
jkiss	$2^{127} \approx 10^{38}$	0.000242	0.007 sec
jlkiss	$2^{191} \approx 10^{58}$	0.000918	0.038 sec
jlkiss64	$2^{251} \approx 10^{76}$	0.000918	0.223 sec
lfsr88	$2^{88} \approx 10^{26}$	0.000377	0.002 sec
lfsr113	$2^{113} \approx 10^{34}$	0.000504	0.008 sec
lfsr258	$2^{258} \approx 10^{78}$	0.002614	0.185 sec

We see that these generators are capable of generating pseudorandom numbers on the order of one-quarter of a billion per second. Let's suppose that computers get much faster and could generate, not 1 billion per second, but 10 billion per second. And let's further suppose that the application will run continuously, non-stop, for an entire year. This will require $10^{10} \times 60 \times 60 \times 24 \times 365 = 3.1536 \times 10^{17} < 2^{59}$ numbers per stream. Thus, if we jump ahead 2^{59} for every stream, we can be pretty confident that the streams will not overlap and thus will be independent of one another. How many streams would that give us? In the case of LFSR88, which has the shortest period of 2^{88}, that still gives us $2^{88}/2^{59} = 2^{88-59} = 2^{29} = 536,870,912$, or well over 500 million independent streams. Surely, this is more than enough streams for our applications. Other generators have vastly more streams.

References

[1] Saucier R. Computer generation of statistical distributions. Aberdeen Proving Ground (MD): Army Research Laboratory (US); 2000 Mar. Report No.: ARL-TR-2168.

[2] Bodt, B. A., and M. S. Taylor. "A Data Based Random Number Generator for a Multivariate Distribution—A User's Manual." ARBRL-TR-02439, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD, November 1982.

[3] Diaconis, P., and B. Efron. "Computer-Intensive Methods in Statistics." Scientific American, pp. 116–130, May, 1983.

[4] Efron, B., and R. Tibshirani. "Statistical Data Analysis in the Computer Age." Science, pp. 390–395, 26 July 1991.

[5] Hammersley, J. M., and D. C. Handscomb. Monte Carlo Methods. London: Methuen & Co. Ltd., 1967.

[6] Hastings, N. A. J., and J. B. Peacock. Statistical Distributions: Handbook for Students and Practitione. New York: John Wiley & Sons, 1975.

[7] Helicon Publishing. The Hutchinson Encyclopedia. http://www.helicon.co.uk, 1999.

[8] Knuth, D. E. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. London: Addison-Wesley, 1969.

[9] Law, A. M., and W. D. Kelton. Simulation Modeling and Analysis. New York: McGraw-Hill, Second Edition, 1991.

[10] New York Times. New York, C1, C6, 8 November 1988.

[11] Pitman, J. Probability. New York: Springer-Verlag, 1993.

[12] Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical Recipes in C: The Art of Scientific Computing. New York: Cambridge University Press, Second Edition, 1992.

[13] Ripley, B. D. Stochastic Simulation. New York: John Wiley & Sons, 1987.

[14] Sachs, L. Applied Statistics: Handbook of Techniques. New York: Springer-Verlag, Second Edition, 1984.

[15] Spiegel, M. R. Probability and Statistics. New York: McGraw-Hill, 1975.

[16] Tausworthe, R. C. "Random Numbers Generated by Linear Recurrence Modulo Two." Mathematics of Computation. Vol. 19, pp. 201–209, 1965.

[17] Taylor, M. S., and J. R. Thompson. "A Data Based Algorithm for the Generation of Random Vectors." Computational Statistics & Data Analysis. Vol. 4, pp. 93–101, 1986.

[18] Thompson, J. R. Empirical Model Building. New York: John Wiley & Sons, 1989.

[19] Watson, E. J. "Primitive Polynomials (Mod 2)." Mathematics of Computation. Vol. 16, pp. 368–369, 1962.

Appendices

Appendix A Linear Congruential Generator

The linear congruential generator (LCG) is used as one of the components in the KISS, JKISS, JLKISS, and JLKISS64 generators. The LCG is defined by the sequence

$$x_{i+1} = ax_i + c \pmod{m} \tag{A-1}$$

for $i \geq 0$, fixed multiplier a, constant c, and modulus m.

Jump Ahead

If x_0 denotes the seed, then the sequence is

$$\begin{aligned}
x_1 &= ax_0 + c \\
x_2 &= ax_1 + c &&= a(ax_0 + c) + c &&= a^2x_0 + ac + c \\
x_3 &= ax_2 + c &&= a(a^2x_0 + ac + c) + c &&= a^3x_0 + a^2c + ac + c \\
&\vdots \\
x_n &= ax_{n-1} + c = && \cdots &&= a^nx_0 + c(a^{n-1} + \cdots + a^2 + a + 1)
\end{aligned}$$

and provides a method for computing the nth term directly from the seed. Thus, the *jump ahead* formula is

$$\boxed{x_n = a^nx_0 + c\sum_{i=0}^{n-1} a^i \pmod{m}}\,, \tag{A-2}$$

where $n \geq 1$. Notice that the summation is over the first n terms of the geometric series. For the case when $a < 1$ the summation is easily carried out by noting that

$$S_n(a) \equiv \sum_{i=0}^{n-1} a^i = 1 + a + a^2 + \cdots + a^{n-1} = aS_n(a) + 1 - a^n, \tag{A-3}$$

which can readily be solved for $S_n(a)$:

$$S_n(a) = \frac{1 - a^n}{1 - a}. \tag{A-4}$$

For our case, though, a is a positive integer such that $1 \leq a < m$ so this formula doesn't help us. Instead, we can use the technique of *exponentiation by squaring.*
First consider the case when n is even. By regrouping terms, we have

$$\begin{aligned}
S_n(a) &= 1 + a + a^2 + a^3 + a^4 + a^5 + a^6 + a^7 + \cdots + a^{n-2} + a^{n-1} \\
&= 1 + a + (1 + a)a^2 + (1 + a)a^4 + (1 + a)a^6 + \cdots + (1 + a)a^{n-2} \\
&= (1 + a)[1 + a^2 + a^4 + a^6 + \cdots + a^{n-2}] \\
&= (1 + a)[1 + (a^2) + (a^2)^2 + (a^2)^3 + \cdots + (a^2)^{(n/2-1}] \\
&= (1 + a)S_{n/2}(a^2). \qquad \text{(A-5)}
\end{aligned}$$

So the series now has exactly half the number of terms, where each term is the square of the previous value. When n is odd we simply add the last term to the sum and then apply the formula to the even number of terms that remain. This leads to the following algorithm:

Algorithm 1 Sum the geometric series $1 + a + a^2 + \cdots + a^{n-1} \pmod{m}$ (n terms)

```
$p \leftarrow 1$, $r \leftarrow 0$
while ($n > 1$) do
    if ($n$ is odd) then
        $r \leftarrow r + pa^{n-1} \pmod{m}$
    end if
    $p \leftarrow p(1 + a) \pmod{m}$
    $a \leftarrow a^2 \pmod{m}$
    $n \leftarrow n/2$
end while
$r \leftarrow r + p \pmod{m}$
return $r$
```

The following C++ code implements this algorithm by making use of the modular functions contained in `mod_math.h`:

```cpp
// 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
uint64_t gs_mod64( uint64_t a, uintmax_t n, uint64_t m ) {

   if ( n == 0 ) return 0;

   uint64_t t = a % m;
   uint64_t p = 1;
   uint64_t r = 0;

   while ( n > 1 ) {

      if ( n & 1 ) r = add_mod64( r, mul_mod64( p, pow_mod64( t, n - 1, m ), m ), m );
      p = mul_mod64( p, add_mod64( 1, t, m ), m );
      t = mul_mod64( t, t, m );
      n >>= 1;
   }
   r = add_mod64( r, p, m );
   return r;
}
```

The following checks were made

```
gs_mod64( 123456789, 0, 4294967296 ) = 0
gs_mod64( 123456789, 1, 4294967296 ) = 1
gs_mod64( 123456789, 10, 4294967296 ) = 1382346382
gs_mod64( 123456789, 1024, 4294967296 ) = 3101645824
gs_mod64( 123456789, 1000000, 4294967296 ) = 2009531328

gs64( 1490024343005336237, 0 ) = 0
gs64( 1490024343005336237, 1 ) = 1
gs64( 1490024343005336237, 10 ) = 7987679512244350278
gs64( 1490024343005336237, 1024 ) = 9396580604419943424
gs64( 1490024343005336237, 12345 ) = 2047449762047247049
```

and verified in Mathematica as follows:

Mathematica Session

```
          (* Sum the first n terms of a geometric series mod m *) :
  In[1]:= gs[a_, n_, m_] := Mod[Sum[a^i, {i, 0, n - 1}], m]
  In[2]:= gs[123456789, 0, 2^32]
 Out[2]= 0
  In[3]:= gs[123456789, 1, 2^32]
 Out[3]= 1
  In[4]:= gs[123456789, 10, 2^32]
 Out[4]= 1382346382
  In[5]:= gs[123456789, 1024, 2^32]
 Out[5]= 3101645824
  In[6]:= gs[123456789, 10^6, 2^32]
 Out[6]= 2009531328
  In[7]:= gs[1490024343005336237, 0, 2^64]
 Out[7]= 0
  In[8]:= gs[1490024343005336237, 1, 2^64]
 Out[8]= 1
  In[9]:= gs[1490024343005336237, 10, 2^64]
 Out[9]= 7987679512244350278
 In[10]:= gs[1490024343005336237, 1024, 2^64]
Out[10]= 9396580604419943424
 In[11]:= gs[1490024343005336237, 12345, 2^64]
Out[11]= 2047449762047247049
```

which sums the first n terms of the geometric series $1 + a + a^2 + \ldots + a^{n-1} \pmod{2^{32}}$. Also, note that

```
(uint64_t)gs_mod( 123456789, 10, 0, 4294967296 ) = 3101645824
(uint64_t)gs_mod( 123456789, 19, 475712, 4294967296 ) = 2009531328
```

Large Jumps

We also need jumps that are greater than what we are able to express with a 64-bit integer, which is $2^{64} - 1$. If we want to jump an entire cycle, we will need jumps as high as 2^{258}. This is handled by allowing for jumps of the form $n = 2^e + c$, where e and c are 32-bit integers (64-bit certainly aren't needed here, nor do we need the full range of 32-bit). Now consider summing the geometric series. We have

$$S_{2^e+c}(a) = \underbrace{1 + a + a^2 + \cdots + a^{2^e-1}}_{S_{2^e}(a)} + \underbrace{a^{2^e} + a^{2^e+1} + \cdots + a^{2^e+c-1},}_{a^{2^e}(1+a+\cdots+a^{c-1})} \tag{A-6}$$

so that

$$\boxed{S_{2^e+c}(a) = S_{2^e}(a) + a^{2^e} S_c(a)}. \tag{A-7}$$

This can be implemented as follows:

```
// 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
uint64_t gs_mod64( uint64_t a, uint32_t e, uint32_t c, uint64_t m ) {

   if ( e == 0 ) return gs_mod64( a, 1 + c, m );

   uint64_t t = a;
   uint64_t r = 1;

  for ( uint32_t i = 0; i < e; ++i ) {

      r = mul_mod64( r, add_mod64( 1, t, m ), m );
      t = mul_mod64( t, t, m );
   }
   if ( c == 0 ) return r;

   return add_mod64( r, mul_mod64( t, gs_mod64( a, c, m ), m ), m );
}
```

Jump Back

It is also possible to jump backwards. Inverting the equation

$$x_{i+1} = a x_i + c \pmod{m}, \tag{A-8}$$

gives

$$x_{i-1} = a^{-1}(x_i - c) \pmod{m}, \tag{A-9}$$

where we substituted $i \to i - 1$ and a^{-1} is the multiplicative inverse in the sense that

$$a^{-1} a = a a^{-1} \equiv 1 \pmod{m}.^{*} \tag{A-10}$$

To find a^{-1}, we first check to make sure that a and m are relatively prime, which means that the greatest common divisor is 1 or $\gcd(a, m) = 1$. Then

$$a^{\phi(m)} \equiv 1 \pmod{m}, \tag{A-11}$$

where $\phi(m)$ is *Euler's Totient* or *Phi function*, which then implies that

$$a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m}. \tag{A-12}$$

The actual computation is performed with the `mod_math` code and will be shown later. But now that we know how to compute a^{-1}, we return to Eq. A-9. If x is the current value, then the n^{th} previous value is given by applying this formula successively, which gives

$$x_{-n} = a^{-n}(x - c) + c - c[1 + a^{-1} + a^{-2} + \ldots + a^{-(n-1)}] \pmod{m} \tag{A-13}$$

Hence, the *jump back* formula is

$$\boxed{x_{-n} = a^{-n}(x - c) + c - c\sum_{i=0}^{n-1} a^{-i} \pmod{m}}, \tag{A-14}$$

[*] The notation $a \equiv b \pmod{m}$ is read "a is congruent to b modulo m" and means that $a - b$ is divisible by m.

where $n \geq 1$. The sum is a simple geometric series $S_n(a^{-1})$ and Algorithm 1 can be used to sum the first n terms.

Notice, incidentally, that Eq. A-9 also provides a method for operating the random number generator in reverse:

```
namespace kiss {   // period is 2^32 = 4294967296
static const uint32_t A     = 0x00010dcd;   // 69069UL;
static const uint32_t C     = 0x00003039;   // 12345UL;
static const uint32_t A_INV = 0xa5e2a705;   // 2783094533UL;
};

uint32_t A     = kiss::A;
uint32_t C     = kiss::C;
uint32_t A_INV = kiss::A_INV;

static uint32_t s;

uint32_t rng( void ) {   // random number generator

   s = A * s + C;
   return s;
}

uint32_t rev( void ) {   // random number generator in reverse

   s = A_INV * ( s - C );
   return s;
}
```

Mathematica Session

```
  In[1]:= gcd[69069, 2^32]
 Out[1]= 1
  In[2]:= gcd[314527869, 2^32]
 Out[2]= 1
  In[3]:= gcd[1490024343005336237, 2^32]
 Out[3]= 1
  In[4]:= gcd[698769069, 3001190298811367423]
 Out[4]= 1
  In[5]:= gcd[4294584393, 18445099517847011327]
 Out[5]= 1
  In[6]:= gcd[4246477509, 18445099517847011327]
 Out[6]= 1
  In[7]:= EulerPhi[2^32]
 Out[7]= 2147483648
  In[8]:= EulerPhi[2^64]
 Out[8]= 9223372036854775808
  In[9]:= EulerPhi[3001190298811367423]
 Out[9]= 3001190298811367422
 In[10]:= EulerPhi[18445099517847011327]
Out[10]= 18445099517847011326
```

Using these Mathematica results and the functions in `mod_math`, we find

$$\begin{aligned} a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m} &= 2783094533 \\ &= \text{a5e2a705}_{16}, \end{aligned} \tag{A-15}$$

when $a = 69069$ and $m = 2^{32}$,

$$\begin{aligned} a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m} &= 1644210389 \\ &= \text{6200a8d5}_{16}, \end{aligned} \tag{A-16}$$

when $a = 314527869$ and $m = 2^{32}$,

$$\begin{aligned} a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m} &= 14241175500494512421 \\ &= \text{c5a2d1aa2af8a125}_{16}, \end{aligned} \tag{A-17}$$

when $a = 1490024343005336237$ and $m = 2^{64}$,

$$\begin{aligned} a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m} &= 4294967296 \\ &= 100000000_{16}, \end{aligned} \tag{A-18}$$

when $a = 698769069$ and $m = 3001190298811367423$,

$$\begin{aligned} a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m} &= 4294967296 \\ &= 100000000_{16}, \end{aligned} \tag{A-19}$$

when $a = 4294584393$ and $m = 18445099517847011327$, and

$$\begin{aligned} a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m} &= 11628298268156854590 \\ &= \mathrm{a16003aa5fc7813e}_{16}, \end{aligned} \tag{A-20}$$

when $a = 4246477509$ and $m = 18445099517847011327$.
We can summarize all these results in Table A-1.

Table 8. Constants for "Jump Back" formula

$\boldsymbol{a}$	$\boldsymbol{m}$	$\boldsymbol{\phi(m)}$	$\boldsymbol{a^{-1}}$ **(mod** $\boldsymbol{m}$**)**
69069	2^{32}	2147483648	2783094533
314527869	2^{32}	2147483648	1644210389
1490024343005336237	2^{64}	9223372036854775808	14241175500494512421
698769069	3001190298811367423	3001190298811367422	$2^{32} = 4294967296$
4294584393	18445099517847011327	18445099517847011326	$2^{32} = 4294967296$
4246477509	18445099517847011327	18445099517847011326	11628298268156854590

According to the Hull-Dobell Theorem,[*] the LCG will have a full period (cycle length) of m for all seed values if and only if the following three conditions are met:

1. m and c are relatively prime (i.e., the greatest common divisor is 1),
2. $a - 1$ is divisible by all prime factors of m,
3. $a - 1$ is divisible by 4 if m is divisible by 4.

Hull and Dobell further point out that with m a power of 2, we need only have c odd and $a \equiv 1 \pmod 4$. Consequently, it is easy to check that the values used in the various RNGs, as listed in Table A-2, satisfy the Hull-Dobell theorem and therefore have a full period of m for all seed values.

Table 9. Constants for Linear Congruential Generators

LCG	$\boldsymbol{a}$	$\boldsymbol{c}$	$\boldsymbol{m}$	$\boldsymbol{a^{-1}}$ **(mod** $\boldsymbol{m}$**)**
`kiss`	69069	12345	2^{32}	2783094533
`jkiss`	314527869	1234567	2^{32}	1644210389
`jlkiss`, `jlkiss64`	1490024343005336237	123456789	2^{64}	14241175500494512421

[*] Hull, T. E.; Dobell, A. R. (1962-01-01). Random Number Generators. SIAM Review. 4 (3): 230–254.

Appendix B Linear Feedback Shift Generator

The linear feedback shift register (LFSR) is used as one of the components in all seven of the generators. The simplest LFSR is contained in the KISS generator, which is coded as follows:

```
x ^= ( x << 13 ), x ^= ( x >> 17 ), x ^= ( x << 5 );
```

When this code is applied to 1, represented by the 32-bit bitstring

$$0000\,0000\,0000\,0000\,0000\,0000\,0000\,000\mathbf{1}_2 \tag{B-1}$$

it gets transformed into the bitstring

$$0000\,0000\,0000\,0000\,0000\,0000\,0000\,000\mathbf{1}_2 \Rightarrow 0000\,0000\,0000\,0\mathbf{1}00\,00\mathbf{1}0\,0000\,00\mathbf{1}0\,000\mathbf{1}_2 \tag{B-2}$$

which is 00042021_{16} in hexadecimal. When the shift register code is applied to 2, it becomes the bit string

$$0000\,0000\,0000\,0000\,0000\,0000\,0000\,00\mathbf{1}0_2 \Rightarrow 0000\,0000\,0000\,\mathbf{1}000\,0\mathbf{1}00\,0000\,0\mathbf{1}00\,00\mathbf{1}0_2 \tag{B-3}$$

which is 00084042_{16} in hexadecimal, and so on. Finally, when this code is applied to $2^{32} - 1$, it becomes the bit string

$$\mathbf{1}000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_2 \Rightarrow \mathbf{1}000\,0000\,0000\,\mathbf{1}000\,0\mathbf{1}00\,0000\,0000\,0000_2 \tag{B-4}$$

which is 80084000_{16} in hexadecimal. We can store these as a matrix of hex values, which encodes how each bit from 1 to 32 gets transformed by the shift register.

The complete set of transformations is given here.

$$\begin{array}{l}
0000\;0000\;0000\;0000\;0000\;0000\;0000\;0001_2 \Rightarrow 0000\;0000\;0000\;0100\;0010\;0000\;0010\;0001_2 = 00042021_{16}\\
0000\;0000\;0000\;0000\;0000\;0000\;0000\;0010_2 \Rightarrow 0000\;0000\;0000\;1000\;0100\;0000\;0100\;0010_2 = 00084042_{16}\\
0000\;0000\;0000\;0000\;0000\;0000\;0000\;0100_2 \Rightarrow 0000\;0000\;0001\;0000\;1000\;0000\;1000\;0100_2 = 00108084_{16}\\
0000\;0000\;0000\;0000\;0000\;0000\;0000\;1000_2 \Rightarrow 0000\;0000\;0010\;0001\;0000\;0001\;0000\;1000_2 = 00210108_{16}\\
0000\;0000\;0000\;0000\;0000\;0000\;0001\;0000_2 \Rightarrow 0000\;0000\;0100\;0010\;0000\;0010\;0011\;0001_2 = 00420231_{16}\\
0000\;0000\;0000\;0000\;0000\;0000\;0010\;0000_2 \Rightarrow 0000\;0000\;1000\;0100\;0000\;0100\;0110\;0010_2 = 00840462_{16}\\
0000\;0000\;0000\;0000\;0000\;0000\;0100\;0000_2 \Rightarrow 0000\;0001\;0000\;1000\;0000\;1000\;1100\;0100_2 = 010808c4_{16}\\
0000\;0000\;0000\;0000\;0000\;0000\;1000\;0000_2 \Rightarrow 0000\;0010\;0001\;0000\;0001\;0001\;1000\;1000_2 = 02101188_{16}\\
0000\;0000\;0000\;0000\;0000\;0001\;0000\;0000_2 \Rightarrow 0000\;0100\;0010\;0000\;0010\;0011\;0001\;0000_2 = 04202310_{16}\\
0000\;0000\;0000\;0000\;0000\;0010\;0000\;0000_2 \Rightarrow 0000\;1000\;0100\;0000\;0100\;0110\;0010\;0000_2 = 08404620_{16}\\
0000\;0000\;0000\;0000\;0000\;0100\;0000\;0000_2 \Rightarrow 0001\;0000\;1000\;0000\;1000\;1100\;0100\;0000_2 = 10808c40_{16}\\
0000\;0000\;0000\;0000\;0000\;1000\;0000\;0000_2 \Rightarrow 0010\;0001\;0000\;0001\;0001\;1000\;1000\;0000_2 = 21011880_{16}\\
0000\;0000\;0000\;0000\;0001\;0000\;0000\;0000_2 \Rightarrow 0100\;0010\;0000\;0010\;0011\;0001\;0000\;0000_2 = 42023100_{16}\\
0000\;0000\;0000\;0000\;0010\;0000\;0000\;0000_2 \Rightarrow 1000\;0100\;0000\;0100\;0110\;0010\;0000\;0000_2 = 84046200_{16}\\
0000\;0000\;0000\;0000\;0100\;0000\;0000\;0000_2 \Rightarrow 0000\;1000\;0000\;1000\;1100\;0100\;0000\;0000_2 = 0808c400_{16}\\
0000\;0000\;0000\;0000\;1000\;0000\;0000\;0000_2 \Rightarrow 0001\;0000\;0001\;0001\;1000\;1000\;0000\;0000_2 = 10118800_{16}\\
0000\;0000\;0000\;0001\;0000\;0000\;0000\;0000_2 \Rightarrow 0010\;0000\;0010\;0011\;0001\;0000\;0000\;0000_2 = 20231000_{16}\\
0000\;0000\;0000\;0010\;0000\;0000\;0000\;0000_2 \Rightarrow 0100\;0000\;0100\;0110\;0010\;0000\;0010\;0001_2 = 40462021_{16}\\
0000\;0000\;0000\;0100\;0000\;0000\;0000\;0000_2 \Rightarrow 1000\;0000\;1000\;1100\;0100\;0000\;0100\;0010_2 = 808c4042_{16}\\
0000\;0000\;0000\;1000\;0000\;0000\;0000\;0000_2 \Rightarrow 0000\;0001\;0000\;1000\;0000\;0000\;1000\;0100_2 = 01080084_{16}\\
0000\;0000\;0001\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0000\;0010\;0001\;0000\;0000\;0001\;0000\;1000_2 = 02100108_{16}\\
0000\;0000\;0010\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0000\;0100\;0010\;0000\;0000\;0010\;0001\;0000_2 = 04200210_{16}\\
0000\;0000\;0100\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0000\;1000\;0100\;0000\;0000\;0100\;0010\;0000_2 = 08400420_{16}\\
0000\;0000\;1000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0001\;0000\;1000\;0000\;0000\;1000\;0100\;0000_2 = 10800840_{16}\\
0000\;0001\;0000\;0000\;0000\;0000\;0000\;0001_2 \Rightarrow 0010\;0001\;0000\;0000\;0001\;0000\;1000\;0000_2 = 21001080_{16}\\
0000\;0010\;0000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0100\;0010\;0000\;0000\;0010\;0001\;0000\;0000_2 = 42002100_{16}\\
0000\;0100\;0000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 1000\;0100\;0000\;0000\;0100\;0010\;0000\;0000_2 = 84004200_{16}\\
0000\;1000\;0000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0000\;1000\;0000\;0000\;1000\;0100\;0000\;0000_2 = 08008400_{16}\\
0001\;0000\;0000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0001\;0000\;0000\;0001\;0000\;1000\;0000\;0000_2 = 10010800_{16}\\
0010\;0000\;0000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0010\;0000\;0000\;0010\;0001\;0000\;0000\;0000_2 = 20021000_{16}\\
0100\;0000\;0000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 0100\;0000\;0000\;0100\;0010\;0000\;0000\;0000_2 = 40042000_{16}\\
1000\;0000\;0000\;0000\;0000\;0000\;0000\;0000_2 \Rightarrow 1000\;0000\;0000\;1000\;0100\;0000\;0000\;0000_2 = 80084000_{16}
\end{array}$$

This is represented in the C++ code as an array of 32 words (stored in hexadecimal form), where each word is 32 bits and represents a whole row. We call such a structure a *bitmatrix* and the 32-bit word it operates on a *bitvector*. We can also have an array of 64 words, where each row consists of a 64-bit word.

Now let A represent this particular 32×32 bitmatrix, and consider applying the shift register again, but instead of using the shift register directly, we are going to use the bitmatrix. First consider applying A to the first bitvector, which is

$$0000\;0000\;0000\;0\mathbf{1}00\;00\mathbf{1}0\;0000\;00\mathbf{1}0\;000\mathbf{1}_2 \tag{B-5}$$

It has a 1 bit in positions 1, 6, 14, and 19 and thus may be considered as a linear combination of the rows 1, 6, 14 and 19.

This means that A should change this bitvector into

$$\begin{aligned}
0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_2 &\Rightarrow 0000\,0000\,0000\,0100\,0010\,0000\,0010\,0001_2\\
0000\,0000\,0000\,0000\,0000\,0000\,0010\,0000_2 &\Rightarrow 0000\,0000\,1000\,0100\,0000\,0100\,0110\,0010_2\\
0000\,0000\,0000\,0000\,0010\,0000\,0000\,0000_2 &\Rightarrow 1000\,0100\,0000\,0100\,0110\,0010\,0000\,0000_2\\
0000\,0000\,0000\,0100\,0000\,0000\,0000\,0000_2 &\Rightarrow 1000\,0000\,1000\,1100\,0100\,0000\,0100\,0010_2
\end{aligned}$$

which are then added together mod 2. In C++ code, mod 2 arithmetic is equivalent to “exclusive-or” (XOR):

$$a + b \pmod 2 \quad \equiv \quad a \oplus b \quad \text{which in C++ code is} \quad \texttt{a \^{} b}, \tag{B-6}$$

where “ ^ ” is the bitwise XOR. Thus, when these are XORed together, we get

$$\begin{aligned}
& 0000\,0000\,0000\,0100\,0010\,0000\,0010\,0001_2\\
\oplus\; & 0000\,0000\,1000\,0100\,0000\,0100\,0110\,0010_2\\
\oplus\; & 1000\,0100\,0000\,0100\,0110\,0010\,0000\,0000_2\\
\oplus\; & \underline{1000\,0000\,1000\,1100\,0100\,0000\,0100\,0010_2}\\
=\; & 0000\,0100\,0000\,1000\,0000\,0110\,0000\,0001_2
\end{aligned} \tag{B-7}$$

Thus, the following C++ code will perform multiplication by the matrix A:

```
// multiply a bitmatrix times a vector and return the result
uint32_t bitmatrix_mul( const bitmatrix_t& A, uint32_t v ) {

   uint32_t result = 0;
   for ( size_t i = 0; i < 32; i++, v >>= 1 ) if ( v & 1 ) result ^= A.row[i];
   return result;
}
```

This is efficient since the XOR operates on all 32 bits at the same time, making use of the inherent parallelism of bitwise operations. So bitmatrix multiplication is relatively fast, but certainly not as fast as the code in the random number generator itself.

To multiply two bitmatrices, A and B, to form a new bitmatrix $C = A \times B$, we form each row of C by multiplying A times each row of B in turn. This then gives us the capability of raising a bitmatrix A to a power A^n, where we use the technique of “exponentiation by squaring.” This will give us the capability of jumping ahead.

We will now show how the inverse bitmatrix, A^{-1}, may be computed. We’ve been writing the bitvector with the least significant bit (LSB) on the right and the most significant it (MSB) on the left, which is the convention in “little endian.” But now let’s write it in “big endian” convention with MSB on the right and LSB on the left:

$$A = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1\\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0\\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix} \tag{B-8}$$

The characteristic polynomial is determined by

$$p(\lambda) \equiv \det(A - \lambda I) \pmod 2. \tag{B-9}$$

If the matrix A is input into Mathematica, then it is easy to compute $p(\lambda)$ using Eq. B-9, but an even easier way to do this is as follows:

Mathematica Session

```
 In[1]:= = PolynomialMod[CharacteristicPolynomial[A, λ], 2]
Out[1]= = 1 + λ^6 + λ^9 + λ^14 + λ^15 + λ^17 + λ^18 + λ^19 + λ^20 + λ^21 + λ^32
 In[2]:= = << FiniteFields`
 In[3]:= = IrreduciblePolynomialQ[%1]
Out[3]= = True
```

Thus we find that the characteristic polynomial of A, given by

$$p(\lambda) = 1 + \lambda^6 + \lambda^9 + \lambda^{14} + \lambda^{15} + \lambda^{17} + \lambda^{18} + \lambda^{19} + \lambda^{20} + \lambda^{21} + \lambda^{32}, \tag{B-10}$$

is also an *irreducible polynomial*[*] in the Galois Field GF(2^{32}). Now since every matrix satisfies its own characteristic equation (Cayley-Hamilton theorem), we have

$$1 + A^6 + A^9 + A^{14} + A^{15} + A^{17} + A^{18} + A^{19} + A^{20} + A^{21} + A^{32} = 0 \pmod 2 \tag{B-11}$$

or, since addition and subtraction are equivalent in mod 2 arithmetic,

$$A^6 + A^9 + A^{14} + A^{15} + A^{17} + A^{18} + A^{19} + A^{20} + A^{21} + A^{32} = 1 \pmod 2 \tag{B-12}$$

so that

$$A(A^5 + A^8 + A^{13} + A^{14} + A^{16} + A^{17} + A^{18} + A^{19} + A^{20} + A^{31}) = 1 \pmod 2, \tag{B-13}$$

and therefore the inverse bitmatrix is given by

$$\begin{aligned} A^{-1} &= A^5 + A^8 + A^{13} + A^{14} + A^{16} + A^{17} + A^{18} + A^{19} + A^{20} + A^{31} \pmod 2 \\ &= A^5 \oplus A^8 \oplus A^{13} \oplus A^{14} \oplus A^{16} \oplus A^{17} \oplus A^{18} \oplus A^{19} \oplus A^{20} \oplus A^{31}. \end{aligned} \tag{B-14}$$

This is easily computed with the functions in `mod_math`, and we get

$$A^{-1} = \left[\begin{array}{cccccccccccccccccccccccccccccccc}
1&0&0&1&0&1&0&0&1&0&1&0&0&0&0&1&1&0&1&0&1&1&0&1&0&1&0&0&1&1&1&1\\
0&1&0&0&1&0&1&0&0&1&0&1&0&0&0&0&1&1&0&1&0&1&1&0&1&0&1&0&0&1&1&1\\
1&0&1&0&0&1&0&1&0&0&1&0&1&1&0&1&0&1&1&0&1&0&1&1&0&1&1&1&1&0&1&1\\
0&1&0&1&0&0&1&0&1&0&0&1&0&1&1&0&1&0&1&1&0&1&0&1&1&0&1&1&1&1&0&1\\
0&0&1&0&1&0&0&1&0&1&0&0&1&0&1&1&0&1&0&1&1&0&1&0&1&1&0&1&1&1&1&0\\
0&0&0&1&0&1&0&0&1&0&1&0&0&1&0&1&1&0&1&0&1&1&0&1&0&1&1&0&1&1&1&1\\
0&0&0&0&1&0&1&0&0&1&0&1&0&0&1&0&1&1&0&1&0&1&1&0&1&0&1&1&0&1&1&1\\
1&0&0&0&0&1&0&1&0&0&1&0&1&1&0&0&0&1&1&0&1&0&1&1&0&1&1&1&0&0&1&1\\
0&1&0&0&0&0&1&0&1&0&0&1&0&1&1&0&0&0&1&1&0&1&0&1&1&0&1&1&1&0&0&1\\
0&0&1&0&0&0&0&1&0&1&0&0&1&0&1&1&0&0&0&1&1&0&1&0&1&1&0&1&1&1&0&0\\
0&0&0&1&0&0&0&0&1&0&1&0&0&1&0&1&1&0&0&0&1&1&0&1&0&1&1&0&1&1&1&0\\
0&0&0&0&1&0&0&0&0&1&0&1&0&0&1&0&1&1&0&0&0&1&1&0&1&0&1&1&0&1&1&1\\
1&0&0&0&0&1&0&0&0&0&1&0&1&1&0&0&0&1&1&0&0&0&1&1&0&1&1&1&0&0&1&1\\
0&1&0&0&0&0&1&0&0&0&0&1&0&1&1&0&0&0&1&1&0&0&0&1&1&0&1&1&1&0&0&1\\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&1&1&0&0&0&1&1&0&0&0&1&1&0&1&1&1&0&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&1&1&0&0&0&1&1&0&0&0&1&1&0&1&1&1&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&1&1&0&0&0&1&1&0&0&0&1&1&0&1&1&1\\
1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0&1&1&0&0&0&1&1&0&0&1&1&0&0&1&1\\
0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0&1&1&0&0&0&1&1&0&0&1&1&0&0&1\\
0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0&1&1&0&0&0&1&1&0&0&1&1&0&0\\
0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0&1&1&0&0&0&1&1&0&0&1&1&0\\
0&0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&0&1&0&0&0&1&1&0&0&0&1&1&0&0&1&1\\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0&0&0&1\\
0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&1\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1
\end{array}\right]. \tag{B-15}$$

It can be verified that $AA^{-1} = A^{-1}A = I$, the identity matrix. We can also verify that A allows us to go forward and gives the same results as the shift register and that A^{-1} allows us to go backward:

[*] Irreducible polynomials in a finite field cannot be factored further and play the same role as prime numbers for the integers.

```
Done with shift register
0       0x1234cafe
1       0xe602a62b
2       0xea347648
3       0xfb55c2f6
4       0x226f4d13
5       0xb2645655
6       0x2d73aa42
7       0x5f430dbf
8       0xe06aaa65
9       0x11ec4e36
10      0x9d728643

Done with bitmatrix, A
0       0x1234cafe
1       0xe602a62b
2       0xea347648
3       0xfb55c2f6
4       0x226f4d13
5       0xb2645655
6       0x2d73aa42
7       0x5f430dbf
8       0xe06aaa65
9       0x11ec4e36
10      0x9d728643

Done with inverse bitmatrix, A_INV
0       0x9d728643
1       0x11ec4e36
2       0xe06aaa65
3       0x5f430dbf
4       0x2d73aa42
5       0xb2645655
6       0x226f4d13
7       0xfb55c2f6
8       0xea347648
9       0xe602a62b
10      0x1234cafe
```

We now list the characteristic polynomials for each of the shift registers. The shift registers in KISS, JKISS, JLKISS, and JLKISS64 are all of the form

$$A = (\mathbf{I} \oplus \mathbf{L}^a)A, \quad A = (\mathbf{I} \oplus \mathbf{R}^b)A, \quad A = (\mathbf{I} \oplus \mathbf{L}^c)A \tag{B-16}$$

where $\mathbf{I}$ is the identity transformation, $\mathbf{L}^q$ is a shift left of q bits, $\mathbf{R}^q$ is a shift right of q bits, and $\oplus$ is bitwise XOR. For KISS, $a = 13$, $b = 17$, $c = 5$, and

$$p(x) = 1 + x^6 + x^9 + x^{14} + x^{15} + x^{17} + x^{18} + x^{19} + x^{20} + x^{21} + x^{32}, \tag{B-17}$$

$$A^{-1} = A^5 + A^8 + A^{13} + A^{14} + A^{16} + A^{17} + A^{18} + A^{19} + A^{20} + A^{31}. \tag{B-18}$$

For JKISS, $a = 5$, $b = 7$, $c = 22$, and

$$p(x) = 1 + x^2 + x^8 + x^{10} + x^{11} + x^{12} + x^{14} + x^{20} + x^{21} + x^{22} + x^{23} + x^{24} + x^{32}, \tag{B-19}$$

$$A^{-1} = A + A^7 + A^9 + A^{10} + A^{11} + A^{13} + A^{19} + A^{20} + A^{21} + A^{22} + A^{23} + A^{31}. \tag{B-20}$$

For JLKISS and JLKISS64, $a = 21$, $b = 17$, $c = 30$, and

$$\begin{aligned} p(x) =& 1 + x + x^4 + x^{12} + x^{13} + x^{14} + x^{16} + x^{19} + x^{25} + x^{27} + x^{30} + \\ & x^{33} + x^{35} + x^{37} + x^{40} + x^{43} + x^{52} + x^{53} + x^{57} + x^{61} + x^{64}, \qquad \text{(B-21)} \\ A^{-1} =& 1 + A^3 + A^{11} + A^{12} + A^{13} + A^{15} + A^{18} + A^{24} + A^{26} + A^{29} + \\ & A^{32} + A^{34} + A^{36} + A^{39} + A^{42} + A^{51} + A^{52} + A^{56} + A^{60} + A^{63}. \qquad \text{(B-22)} \end{aligned}$$

The shift registers in the LFSRs have a different form, but they also make use of bit shifts with constants a, b, and c. LFSR88 has three shift registers. The first one has $a = 12$, $b = 13$, $c = 19$, and

$$p_1(x) = 1 + x^{13} + x^{19} + x^{25} + x^{31}. \tag{B-23}$$

The second one has $a = 4$, $b = 2$, $c = 25$, and

$$p_2(x) = 1 + x^2 + x^{29}. \tag{B-24}$$

The third one has $a = 17$, $b = 3$, $c = 11$, and

$$p_3(x) = 1 + x^2 + x^3 + x^6 + x^{10} + x^{15} + x^{17} + x^{19} + x^{28}. \tag{B-25}$$

For LFSR113, there are four shift registers, and the characteristic polynomials are

$$p_1(x) = 1 + x^2 + x^4 + x^6 + x^{11} + x^{22} + x^{31} \tag{B-26}$$

$$p_2(x) = 1 + x^2 + x^{29} \tag{B-27}$$

$$p_3(x) = 1 + x^4 + x^8 + x^{12} + x^{13} + x^{16} + x^{20} + x^{24} + x^{28} \tag{B-28}$$

$$p_4(x) = 1 + x^3 + x^4 + x^5 + x^6 + x^{11} + x^{12} + x^{18} + x^{25} \tag{B-29}$$

For LFSR258, there are five shift registers, and the characteristic polynomials are

$$p_1(x) = 1 + x + x^{13} + x^{38} + x^{63} \tag{B-30}$$

$$p_2(x) = 1 + x^{11} + x^{24} + x^{44} + x^{55} \tag{B-31}$$

$$p_3(x) = 1 + x^2 + x^3 + x^6 + x^8 + x^{14} + x^{16} + x^{18} + x^{27} + x^{35} + x^{52} \tag{B-32}$$

$$\begin{aligned} p_4(x) = 1 &+ x^4 + x^5 + x^7 + x^{11} + x^{14} + x^{17} + x^{21} + x^{24} + x^{27} + x^{34} + \\ & x^{37} + x^{47} \end{aligned} \tag{B-33}$$

$$p_5(x) = 1 + x^3 + x^{41} \tag{B-34}$$

The inverse bitmatrix $A^{-1} \pmod 2$ is easily computed from the characteristic polynomial. For example, consider the characteristic equation

$$p_1(x) = 1 + x + x^{13} + x^{38} + x^{63} = 0 \pmod 2 \tag{B-35}$$

Adding 1 mod 2 to both sides gives

$$x + x^{13} + x^{38} + x^{63} = x(1 + x^{12} + x^{37} + x^{62}) = 1 \pmod 2 \tag{B-36}$$

and since the bitmatrix satisfies its own characteristic polynomial (Cayley-Hamilton theorem), we get

$$A^{-1} = 1 + A^{12} + A^{37} + A^{62} \pmod 2. \tag{B-37}$$

So it is easy to compute the inverse by simple inspection of the characteristic polynomial. It was also verified in all these cases that the *characteristic polynomial* is also the *irreducible polynomial.*

Collins* provides another method for computing the characteristic polynomial and jumping ahead, which we will simply summarize here.

- Select any particular bit of the 32-bit word and run it through the given shift register approximately100 times to form a random bit stream.
- Feed this stream to the Berlekamp-Massey algorithm and it will output the characteristic polynomial $p(x)$, which also happens to be the irreducible polynomial.
- To jump ahead n steps, compute the jump polynomial $j(x) = x^n \pmod{p(x)}$.
- The jump state is then obtained by treating the jump polynomial as a bitvector and computing $A \times j$, in the notation here. (Collins basically describes computing A on the fly, whereas in this report, we precompute it and store it.)

Collins not only describes the procedure in detail for a number of random number generators (including the Mersenne Twistor), but also provides explicit C++ code which implements it.

We now have a method of jumping ahead, jumping backward, and also running the random number generator in reverse.

*Collins J. *Testing, Selection, and Implementation of Random Number Generators* US Army Research Laboratory, ARL-TR-4498, Aberdeen Proving Ground, MD July, 2008

Cycle Length

It is straightforward to compute the cycle length of the LFSRs. LFSR88 consists of three independent shift registers, so its period is the product of the three separate periods.

$$\begin{aligned} P_{\text{LFSR88}} &= (2^a - 1)(2^b - 1)(2^c - 1) \\ &= 2^{a+b+c} - 2^{a+b} - 2^{a+c} - 2^{b+c} + 2^a + 2^b + 2^c - 1, \end{aligned} \tag{B-38}$$

where $a = 31$, $b = 29$, and $c = 28$. We can code this as follows:

```
virtual void jump_cycle( void ) { // jump ahead a full cycle of lfsr88
   const uint32_t A = 31, B = 29, C = 28;
   jump_ahead( A + B + C, 0 );
   jump_back( A + B, 0 ); jump_back( A + C, 0 ); jump_back( B + C, 0 );
   jump_ahead( A, 0 ); jump_ahead( B, 0 ); jump_ahead( C, 0 );
   jump_back( 1 );
}
```

LFSR113 consists of four independent shift registers, and its period is

$$\begin{aligned} P_{\text{LFSR113}} =& (2^a - 1)(2^b - 1)(2^c - 1)(2^d - 1) \\ =& 2^{a+b+c+d} - 2^{b+c+d} - 2^{a+c+d} - 2^{a+b+d} - 2^{a+b+c} + \\ & 2^{a+b} + 2^{a+c} + 2^{a+d} + 2^{b+c} + 2^{b+d} + 2^{c+d} - \\ & 2^a - 2^b - 2^c - 2^d + 2^e + 1, \end{aligned} \tag{B-39}$$

where $a = 31$, $b = 29$, $c = 28$, $d = 25$.

Finally, LFSR258 consists of five independent shift registers, and its period is

$$\begin{aligned} P_{\text{LFSR258}} =& (2^a - 1)(2^b - 1)(2^c - 1)(2^d - 1)(2^e - 1), \\ =& 2^{a+b+c+d+e} - 2^{b+c+d+e} - 2^{a+c+d+e} - 2^{a+b+d+e} - 2^{a+b+c+e} - 2^{a+b+c+d} + \\ & 2^{c+d+e} + 2^{b+d+e} + 2^{b+c+e} + 2^{b+c+d} + 2^{a+d+e} + \\ & 2^{a+c+e} + 2^{a+c+d} + 2^{a+b+e} + 2^{a+b+d} + 2^{a+b+c} - \\ & 2^{a+b} - 2^{a+c} - 2^{a+d} - 2^{a+e} - 2^{b+c} - \\ & 2^{b+d} - 2^{b+e} - 2^{c+d} - 2^{c+e} - 2^{d+e} + \\ & 2^a + 2^b + 2^c + 2^d + 2^e - 1, \end{aligned} \tag{B-40}$$

where $a = 63$, $b = 55$, $c = 52$, $d = 47$, and $e = 41$.

These are all coded much the same way.

Jumping Ahead to provide Independent Streams

Independent streams of pseudorandom numbers can be obtained with the Jump Ahead and Jump Back methods. Let's consider the size of the jumps to ensure independence and still provide many such streams. We've seen that the RNGs considered here are capable of delivering one-quarter of a billion numbers per second. Let's suppose that computers get much faster in the near future and we can generate not 1 billion, but 10 billion pseudorandom numbers per second. And let's suppose further that we need to have our application run continuously, non-stop, for one month. That would require a stream of

$$10^{10} \times 60 \times 60 \times 24 \times 30 = 2.592 \times 10^{16} \tag{B-41}$$

pseudorandom numbers. Now since $2^{54} < 2.592 \times 10^{16} < 2^{55}$, a jump of 2^{55} would ensure that there is no overlap between streams. And since $2^{88}/2^{55} = 2^{33} > 8.5 \times 10^9$, we would still have well over 8 billion independent streams for our application.

So how do we jump ahead $2^{55} = 36028797018963968$? First, let's describe the procedure used by Collins to compute a jump of $2^{20} = 1048576$ for LFSR88, where he shows that

$$\begin{aligned} x_{2^{20}} = & x_{30} \oplus x_{27} \oplus x_{26} \oplus x_{25} \oplus x_{24} \oplus x_{23} \oplus x_{21} \oplus \\ & x_{20} \oplus x_{19} \oplus x_{18} \oplus x_{14} \oplus x_{12} \oplus x_{9} \oplus x_{8} \oplus x_{5}. \end{aligned} \tag{B-42}$$

Now LFSR88 has three shift registers, with corresponding characteristic polynomials

$$\begin{aligned} p_1(x) &= 1 + x^{13} + x^{19} + x^{25} + x^{31} \\ p_2(x) &= 1 + x^2 + x^{29} \\ p_3(x) &= 1 + x^2 + x^3 + x^6 + x^{10} + x^{15} + x^{17} + x^{19} + x^{28} \end{aligned} \tag{B-43}$$

Using Mathematica, we can verify this result:

Mathematica Session

```
 In[1]:= = PolynomialMod[PolynomialMod[x^2^20, 1 + x^13 + x^19 + x^25 + x^31], 2]
Out[1]= = x^5 + x^8 + x^9 + x^12 + x^14 + x^18 + x^19 + x^20 + x^21 + x^23 + x^24 + x^25 + x^26 + x^27 + x^30
 In[2]:= = IrreduciblePolynomialQ[%]
Out[2]= = True
```

The technique used is modular exponentiation by squaring in a finite field. We describe this in a systematic manner beginning with ordinary exponentiation by squaring.

Exponentiation by Squaring

For example, suppose we want to raise a base b to a power 25. We first express the exponent in binary form:

$$b^{25} = b^{16+8+1} = b^{2^4+2^3+2^0} = b^{11001_2}. \tag{B-44}$$

Then the algorithm proceeds as follows:
Initialize $r = 1$, $t = b$.

	$r \leftarrow r \cdot t$	$t \leftarrow t^2$
11001_2	b	b^2
11001_2		b^4
11001_2		b^8
11001_2	$b \cdot b^8$	b^{16}
11001_2	$bb^8 \cdot b^{16}$	b^{32}

Return $r = bb^8b^{16}$.
Another example:

$$b^{62} = b^{32+16+8+4+2} = b^{2^5+2^4+2^3+2^2+2^1} = b^{111110_2}. \tag{B-45}$$

Given base b and exponent n.
Express n in binary form and initialize $r = 1$, $t = b$.

	$r \leftarrow r \cdot t$	$t \leftarrow t^2$
111110_2	1	b^2
111110_2	b^2	b^4
111110_2	$b^2 \cdot b^4$	b^8
111110_2	$b^2b^4 \cdot b^8$	b^{16}
111110_2	$b^2b^4b^8 \cdot b^{16}$	b^{32}
111110_2	$b^2b^4b^8b^{16} \cdot b^{32}$	b^{64}

Return $r = b^2b^4b^8b^{16}b^{32}$.
Result obtained with six squarings and five multiplies instead of 62 multiplies. (The last squaring is unnecessary.)

Modular Exponentiation by Squaring

Example

$$\begin{aligned} b^{62} \pmod{m} &= b^{32+16+8+4+2} \pmod{m} \\ &= b^{2^5+2^4+2^3+2^2+2^1} \pmod{m} \\ &= b^{111110_2} \pmod{m}. \end{aligned} \tag{B-46}$$

Given base b, modulus m, and exponent n.
Express n in binary form and initialize $r = 1$, $t = b$.

	$r \leftarrow r \cdot t \pmod{m}$	$t \leftarrow t^2 \pmod{m}$
$11111\mathbf{0}_2$	1	b^2
$1111\mathbf{1}0_2$	b^2	b^4
$111\mathbf{1}10_2$	$b^2 \cdot b^4$	b^8
$11\mathbf{1}110_2$	$b^2 b^4 \cdot b^8$	b^{16}
$1\mathbf{1}1110_2$	$b^2 b^4 b^8 \cdot b^{16}$	b^{32}
$\mathbf{1}11110_2$	$b^2 b^4 b^8 b^{16} \cdot b^{32}$	b^{64}

Return $r = b^2 b^4 b^8 b^{16} b^{32}$.
Result obtained with six squarings and five multiplies instead of 62 multiplies. (The last squaring is unnecessary since it's not used.)

Modular Exponentiation by Squaring in a Finite Field

Given an irreducible polynomial $p(x)$ and exponent n.
Find $x^n \pmod{p(x)}$.
Example

$$x^{62} = x^{32+16+8+4+2} = x^{2^5+2^4+2^3+2^2+2^1} = x^{111110_2} \quad \text{and} \quad p(x) = x^{31} + x^{25} + x^{19} + x^{13} + 1.$$

Express n in binary form and initialize $r = 1$, $t = x$.

	$r \leftarrow r \cdot t \pmod{p(x)}$	$t \leftarrow t^2 \pmod{p(x)}$
$11111\mathbf{0}_2$	1	x^2
$1111\mathbf{1}0_2$	x^2	x^4
$111\mathbf{1}10_2$	$x^2 \cdot x^4$	x^8
$11\mathbf{1}110_2$	$x^6 \cdot x^8$	x^{16}
$1\mathbf{1}1110_2$	$x^{14} \cdot x^{16}$	$x^{26} + x^{20} + x^{14} + x$
$\mathbf{1}11110_2$	$x^{30} \cdot (x^{26} + x^{20} + x^{14} + x)$	$x^{21} + x^{16} + x^{15} + x^9 + x^3 + x$

We made use of the following:

$$x^{32} \pmod{p(x)} = x^{26} + x^{20} + x^{14} + x \tag{B-47}$$

$$x^{64} \pmod{p(x)} = x^{21} + x^{16} + x^{15} + x^9 + x^3 + x \tag{B-48}$$

$$x^{30}(x^{26} + x^{20} + x^{14} + x) \pmod{p(x)} = x^{19} + x^{14} + x^{13} + x^7 + x + 1 \tag{B-49}$$

where

$$p(x) = x^{31} + x^{25} + x^{19} + x^{13} + 1. \tag{B-50}$$

There are two things we need at this point:

- an algorithm for modular squaring a polynomial, $a(x)^2 \pmod{p(x)}$, while making use of the fact that $a(x)^2 = a(x^2)$ in mod 2 arithmetic.
- an algorithm for modular multiplication of two polynomials: $a(x) \cdot b(x) \pmod{p(x)}$.

Let

$$a(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \quad \text{and} \quad b(x) = b_{m-1}x^{m-1} + \ldots + b_1 x + b_0$$

Then, writing $a(x)$ in reverse order, we have

$$\begin{aligned} a(x) \cdot b(x) &= (a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1}) \cdot b(x) \\ &= a_0 b(x) + x a_1 b(x) + x^2 a_2 b(x) + \ldots + x^{n-2} a_{n-2} b(x) + x^{n-1} a_{n-1} b(x) \\ &= a_0 b(x) + x(a_1 b(x) + x(a_2 b(x) + \ldots + x(a_{n-2} b(x) + x(a_{n-1} b(x))) \ldots)). \end{aligned} \tag{B-51}$$

This is all described very succinctly in the Collins report. Here is an implementation of the complete procedure, which includes modular multiplication, squaring, and raising to a power:

```
const size_t N_BITS = 33;   // degree of the irreducible polynomial, p

// returns a * b mod p
bitset<N_BITS> poly_mul( const bitset<N_BITS>& a, const bitset<N_BITS>& b, const bitset<N_BITS>& p ) {

   static size_t msb = 0;
   bitset<N_BITS> r;
   int c;
   if ( msb == 0 ) { // get msb of irreducible polynomial
      for ( size_t k = p.size()-1; k >= 0; --k ) if ( p.test(k) ) { msb = k; break; }
   }
   for ( int i = N_BITS-1; i >= 0; --i ) {

      r <<= 1;
      c = r[msb];
      if ( c ) r ^= p;
      if ( a[i] ) r ^= b;
   }
   return r;
}

// returns a^2 mod p
bitset<N_BITS> poly_sqr( const bitset<N_BITS>& a, const bitset<N_BITS>& p ) {

   static size_t msb = 0;
   bitset<N_BITS> r;
   int c;
   if ( msb == 0 ) { // get msb of irreducible polynomial
      for ( size_t k = p.size()-1; k >= 0; --k ) if ( p.test(k) ) { msb = k; break; }
   }
   for ( int i = N_BITS-1; i >= 0; --i ) {

      r <<= 1;
      c = r[msb];
      if ( c ) r ^= p;
      if ( a[i] ) r ^= a;
   }
   return r;
}

// returns a^n mod p
bitset<N_BITS> poly_pow( uintmax_t n, const bitset<N_BITS>& p ) {

   bitset<N_BITS> r( 0x1 );   // 1
   bitset<N_BITS> t( 0x2 );   // a

   while ( n > 0 ) {

      if ( n & 1 ) r = poly_mul( r, t, p );
      t = poly_sqr( t, p );
      n >>= 1;
   }
   return r;
}

// returns a^n mod p, where n = 2^e + c
bitset<N_BITS> poly_pow( uint32_t e, uint32_t c, const bitset<N_BITS>& p ) {

   bitset<N_BITS> t( 0x2 );   // a
   if ( e > 0 ) {
      for ( uint32_t i = 0; i < e; ++i ) t = poly_sqr( t, p );
   }
   if ( e == 0 ) c++;
   bitset<N_BITS> r = poly_pow( c, p );
   if ( e ) r = poly_mul( r, t, p );

   return r;
}
```

Appendix C Multiply with Carry Generator

The multiply-with-carry (MWC) method* is defined by the sequence

$$x_{i+1} = a x_i + c \pmod{m} \tag{C-1}$$

*Marsaglia, G. Random Number Generators, Journal of Modern Applied Statistical Methods, May 2003, Vol. 2., No. 1, 2-13.

for $i \geq 0$, fixed multiplier a, variable carry c, and constant modulus m. It looks very much like the LCG (cf. Eq. (A-1)) except that c is not constant, but is instead encoded in a. The constant a is 64-bit, x is 32-bit, and the multiply and add in Eq. (C-1) is performed using 64-bit arithmetic. The value of c is then taken from the upper 32-bits of a. Thus, all we really need is to keep track of all 64-bits of a. Jumping ahead n steps simply consists of raising a to a power n, as shown in the following code snippet:

```
1  static const uint64_t A      = 0x0000000029a65ead;     // 698769069ULL;
2  static const uint64_t MOD    = 0x29a65eacffffffff;     // 3001190298811367423ULL
3  static const uint64_t A_INV  = 0x0000000100000000;     // 4294967296ULL;
4  static const uint64_t PERIOD = 1500595149405683711ULL; // = 2^60 + 347673644798836735
5
6  static uint32_t s1;   // lower 32 bits
7  static uint32_t s2;   // upper 32 bits
8
9  uint32_t rng( void ) {   // random number generator
10
11    uint64_t a = A * s1 + s2;
12    s2 = ( a >> 32u );    // upper
13    s1 = uint32_t( a );   // lower
14    return s1;
15 }
16
17 uint32_t rev( void ) {   // random number generator in rev
18
19    uint64_t a = s1 + ( (uint64_t)s2 << 32u );
20    a = mul_mod64( A_INV, a, MOD );
21    s2 = ( uint32_t )( a >> 32u );
22    s1 = ( uint32_t )( a );
23    return s1;
24 }
25
26 void jump_ahead( uintmax_t n ) { // jump ahead n
27
28    uint64_t a = s1 + ( (uint64_t)s2 << 32u );
29    a = mul_mod64( pow_mod64( A, n, MOD ), a, MOD );
30    s2 = ( uint32_t )( a >> 32u );
31    s1 = ( uint32_t )( a );
32 }
```

The state of the MWC consists of the two 32-bit seeds, `s1` and `s2` on lines 6 and 7. The carry is then taken from the upper 32-bits on line 12 and the lower 32 bits on line 13 are returned as the next number in the sequence in line 14. Lines 17-24 show how we can run the generator in reverse. Lines 26-32 shows the jump ahead code.

The period is computed from the formula

$$m = (a \times 2^{32} - 2)/2 = a \times 2^{31} - 1. \tag{C-2}$$

Cycle Length of KISS Family Generators

The KISS family of generators consist of three separate generators: LCG, LFSR, and MWC. We also can compute the period exactly of these generators and jump the entire cycle.
The period of KISS is given by

$$P_{\text{KISS}} = 2^{32}(2^{32} - 1)(698769069 \times 2^{31} - 1). \tag{C-3}$$

Using Mathematica, we find that

$$\begin{aligned} P_{\text{KISS}} &= 2^{32}(2^{32} - 1)(698769069 \times 2^{31} - 1) \\ &= 276810946728915880903908138444600115 20 \\ &= 10100110100110010111101010110011010110010110011010 \\ &\ 00010101000100000000000000000000000000000010000000 \\ &\ 0000000000000000000000000_2 \quad \text{(125 digits in base 2)}. \end{aligned} \tag{C-4}$$

The number is obviously too large to express in a 32-bit, or even a 64-bit, computer word. Instead, the C++ `std::bitset<125>` data structure is tailor made for this purpose, and the following code then allows us to jump ahead an entire cycle length:

```
virtual void jump_cycle( void ) { // jump ahead a full cycle of kiss

   std::bitset<125> p( std::string( "10100110100110010111101010110011010110010110011010" ) +
                       std::string( "00010101000100000000000000000000000000000010000000" ) +
                       std::string( "0000000000000000000000000" ) );
   for ( size_t i = 0; i < p.size(); ++i ) if ( p.test(i) ) jump_ahead( i, 0 );
}
```

The `for` loop on line 6 tests whether each bit is a 0 or a 1. If it's a 0, it does nothing; if it's a 1, it jumps ahead 2^i in the sequence, and so by the end of the loop it has jumped the binary representation of the complete cycle. This gives us a way of verifying the actual period of the particular generator.
The period of JKISS is

$$\begin{aligned} P_{\text{JKISS}} &= 2^{32}(2^{32}-1)(4294584393 \times 2^{31}-1) \\ &= 170\,126\,015\,070\,303\,082\,434\,102\,628\,274\,311\,004\,160 \\ &= 11111111111110100010100001001000000000000000010111 \\ &\quad 01011110110101000000000000000000000000000000100000 \\ &\quad 000000000000000000000000000_2 \quad (127 \text{ digits in base } 2\text{ }). \end{aligned} \tag{C-5}$$

The period of JLKISS is

$$\begin{aligned} P_{\text{JLKISS}} &= 2^{64}(2^{64}-1)(4294584393 \times 2^{31}-1) \\ &= 3138271061012620924047441856806230331094853687768430673920 \\ &= 11111111111110100010100001001000111111111111111111 \\ &\quad 11111111111101000000000000000101110101111011011100000 \\ &\quad 00000000000000000000000000100000000000000000000000 \\ &\quad 00000000000000000000000000000000000000000_2 \quad (191 \text{ digits in base } 2\text{ }). \end{aligned} \tag{C-6}$$

The period of JLKISS64 is

$$\begin{aligned} P_{\text{JLKISS64}} &= 2^{64}(2^{64}-1)(4294584393 \times 2^{31}-1)(698769069 \times 2^{31}-1) \\ &= 47092743316757674365569961704867972203 \\ &\quad 43483431369386641683085078522096517120 \\ &= 10100110100101011010110101010010100111001011100000 \\ &\quad 01110101001010000011000110011000011010111111010110 \\ &\quad 00110100011111100010110001010100110100000100001101 \\ &\quad 11101011111111111111111111111111111111000000000000 \\ &\quad 00000000000000000000000000000000000000000000000000 \\ &\quad 00_2 \quad (252 \text{ digits in base } 2). \end{aligned} \tag{C-7}$$

These are all handled with `bitset` data structures of size, 127, 191, and 252, respectively.

Appendix D Code Listings

Table 10. mod_math Reference Guide

Mathematical Notation	mod_math
$a + b \pmod{m}$	uint32_t add_mod32(uint32_t a,uint32_t b,uint32_t m) uint64_t add_mod64(uint64_t a,uint64_t b,uint64_t m)
$a + b \pmod{2^{32}}$	uint32_t add32(uint32_t a,uint32_t b)
$a + b \pmod{2^{64}}$	uint64_t add64(uint64_t a,uint64_t b)
$a \cdot b \pmod{m}$	uint32_t mul_mod32(uint32_t a,uint32_t b,uint32_t m) uint64_t mul_mod64(uint64_t a,uint64_t b,uint64_t m)
$a \cdot b \pmod{2^{32}}$	uint32_t mul32(uint32_t a,uint32_t b)
$a \cdot b \pmod{2^{64}}$	uint64_t mul64(uint64_t a,uint64_t b)
$a^n \pmod{m}$	uint32_t pow_mod32(uint32_t a,uint32_t b,uint32_t m) uint64_t pow_mod64(uint64_t a,uint64_t b,uint64_t m)
$a^n \pmod{2^{32}}$	uint32_t pow32(uint32_t a,uint32_t n)
$a^n \pmod{2^{64}}$	uint64_t pow64(uint64_t a,uint64_t n)
$a^{2^e+c} \pmod{m}$	uint32_t pow_mod32(uint32_t a,uint32_t e, uint32_t c,uint32_t m) uint64_t pow_mod64(uint64_t a,uint64_t e,uint64_t c,uint64_t m)
$a^{2^e+c} \pmod{2^{32}}$	uint32_t pow32(uint32_t a,uint32_t e, uint32_t c)
$a^{2^e+c} \pmod{2^{64}}$	uint64_t pow64(uint64_t a,uint64_t e, uint64_t c)
$\sum_{i=0}^{n-1} a^i \pmod{m}$	uint32_t gs_mod32(uint32_t a,uint32_t n,uint32_t m) uint64_t gs_mod64(uint64_t a,uint64_t n,uint64_t m)
$\sum_{i=0}^{n-1} a^i \pmod{2^{32}}$	uint32_t gs32(uint32_t a,uint32_t n)
$\sum_{i=0}^{n-1} a^i \pmod{2^{64}}$	uint64_t gs64(uint64_t a,uint64_t n)
$\sum_{i=0}^{2^e+c-1} a^i \pmod{m}$	uint32_t gs_mod32(uint32_t a,uint32_t e,uint32_t c,uint32_t m) uint64_t gs_mod64(uint64_t a,uint64_t e,uint64_t c,uint64_t m)
$\sum_{i=0}^{2^e+c-1} a^i \pmod{2^{32}}$	uint32_t gs32(uint32_t a,uint32_t e,uint32_t c)
$\sum_{i=0}^{2^e+c-1} a^i \pmod{2^{64}}$	uint64_t gs64(uint64_t a,uint64_t e,uint64_t c)

Listing D-1. mod_math.h

```
// mod_math.h: modular math for 32-bit and 64-bit unsigned integers
// Ref: https://github.com/cmcqueen/simplerandom
// R. Saucier, 14 October 2016

#include <cstdint>
#include <cassert>

static const uint64_t    M         = 4294967296ULL;                       // 2^32
static const uint64_t    TWO32     = 4294967296ULL;                       // 2^32
static const long double TWO17     = 131072.0L;                           // 2^17
static const long double TWO35     = 34359738368.0L;                      // 2^35
static const long double TWO53     = 9007199254740992.0L;                 // 2^53
static const long double TWO32_INV = 2.3283064365386962890625000e-10L;   // 2^(-32)
static const long double TWO64_INV = 5.4210108624275221700372640e-20L;   // 2^(-64)

// a + b mod m
uint32_t add_mod32( uint32_t a, uint32_t b, uint32_t m ) {

#ifdef UINT64_C // use the native 64-bit capability

   if ( b <= UINT32_MAX - a )
      return ( a + b ) % m;
   else
      return ( uint64_t( a ) + b ) % m;

#else // native 64-bit not available, so perform addition using 32-bit

```

```
   a %= m;
   b %= m;
   uint32_t t;
   if ( b <= UINT32_MAX - a )
      return ( a + b ) % m;

   if ( m <= ( UINT32_MAX >> 1 ) )
      return ( ( a % m ) + ( b % m ) ) % m;

   t = a + b;
   if ( t > uint32_t( m * 2 ) ) // m*2 must be truncated before compare
      t -= m;
   t -= m;
   return t % m;

#endif // UINT64_C
}

// a + b mod 2^32
uint32_t add32( uint32_t a, uint32_t b ) {

#ifdef UINT64_C // use the native 64-bit capability

   if ( b <= UINT32_MAX - a )
      return ( a + b ) % M;
   else
      return ( uint64_t( a ) + b ) % M;

#else // native 64-bit not available, so perform addition using 32-bit

   a %= M;
   b %= M;
   uint32_t t;
   if ( b <= UINT32_MAX - a )
      return ( a + b ) % M;

   if ( m <= ( UINT32_MAX >> 1 ) )
      return ( ( a % M ) + ( b % M ) ) % M;

   t = a + b;
   if ( t > uint32_t( M * 2 ) ) // m*2 must be truncated before compare
      t -= M;
   t -= M;
   return t % M;

#endif // UINT64_C
}

// a * b mod m
uint32_t mul_mod32( uint32_t a, uint32_t b, uint32_t m ) {

#ifdef UINT64_C // use the native 64-bit capability

   uint64_t t = uint64_t( a ) * b;
   return uint32_t( t % m );

#else // native 64-bit not available, so perform multiplication using 32-bit

   a %= m;
   b %= m;
   uint32_t r = 0;
   uint32_t t;

   if ( b >= m ) {

      if ( m > UINT32_MAX / 2u ) b -= m;
      else                       b %= m;
   }

   while ( a != 0 ) {

      if ( a & 1u ) {

         if ( b >= m - r ) r -= m;
         r += b;
      }
      a >>= 1u;

      t = b;
      if ( b >= m - t ) t -= m;
      b += t;
   }
   return r;

#endif // UINT64_C
}

// a * b mod 2^32
uint32_t mul32( uint32_t a, uint32_t b ) {

#ifdef UINT64_C // use the native 64-bit capability

   uint64_t t = uint64_t( a ) * b;
   return uint32_t( t % M );

#else // native 64-bit not available, so perform multiplication using 32-bit

   a %= M;
   b %= M;
   uint32_t r = 0;
   uint32_t t;

   if ( b >= m ) {

      if ( m > UINT32_MAX / 2u ) b -= M;
      else                       b %= M;
   }

   while ( a != 0 ) {

      if ( a & 1u ) {

         if ( b >= M - r ) r -= M;
         r += b;
      }
      a >>= 1u;

      t = b;
      if ( b >= M - t ) t -= M;
      b += t;
   }
   return r;

#endif // UINT64_C
}

// 32-bit methods

// a^n mod m
uint32_t pow_mod32( uint32_t a, uintmax_t n, uint32_t m ) {

   uint32_t r = 1;
   uint32_t t = a;

   for (;;) {

      if ( n & 1 ) r = mul_mod32( r, t, m );
      n >>= 1;
      if ( n == 0 ) break;
      t = mul_mod32( t, t, m );
   }
   return r;
}

// a^n mod m, where n = 2^e + c
uint32_t pow_mod32( uint32_t a, uintmax_t e, uintmax_t c, uint32_t m ) {

   if ( e == 0 ) return pow_mod32( a, c + 1, m );
   uint32_t t = a;
   for ( uintmax_t i = 0; i < e; ++i ) t = mul_mod32( t, t, m );
   return mul_mod32( pow_mod32( a, c, m ), t, m );
}

// a^n mod 2^32
uint32_t pow32( uint32_t a, uintmax_t n ) {

   uint32_t r = 1;
   uint32_t t = a;

   for (;;) {

      if ( n & 1 ) r *= t;
      n >>= 1;
      if ( n == 0 ) break;
      t *= t;
   }
   return r;
}

// a^n mod 2^32, where n = 2^e + c
uint64_t pow32( uint32_t a, uint32_t e, uint32_t c ) {

   if ( e == 0 ) return pow32( a, c + 1 );
   uint32_t t = a;
   for ( uint32_t i = 0; i < e; ++i ) t = mul32( t, t );
   return mul32( pow32( a, c ), t );
}

// sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
uint32_t gs_mod32( uint32_t a, uint32_t n, uint32_t m ) {

   if ( n == 0 ) return 0;

   uint32_t t = a % m;
   uint32_t p = 1;
   uint32_t r = 0;

   while ( n > 1 ) {

      if ( n & 1 ) r = add_mod32( r, mul_mod32( p, pow_mod32( t, n - 1, m ), m ), m );
      p = mul_mod32( p, add_mod32( 1, t, m ), m );
      t = mul_mod32( t, t, m );
      n >>= 1;
   }
   r = add_mod32( r, p, m );
   return r;
}

// sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
uint32_t gs_mod32( uint32_t a, uint32_t e, uint32_t c, uint32_t m ) {

   if ( e == 0 ) return gs_mod32( a, 1 + c, m );

   uint32_t t = a;
   uint32_t r = 1;

  for ( uint32_t i = 0; i < e; ++i ) {

      r = mul_mod32( r, add_mod32( 1, t, m ), m );
      t = mul_mod32( t, t, m );
   }
   if ( c == 0 ) return r;

   return add_mod32( r, mul_mod32( t, gs_mod32( a, c, m ), m ), m );
```

```
}

// sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^32
uint32_t gs32( uint32_t a, uintmax_t n ) {

   if ( n == 0 ) return 0;
   if ( n == 1 ) return 1;

   uint32_t t = a;
   uint32_t p = 1;
   uint32_t r = 0;

   while ( n > 1 ) {

      if ( n & 1 ) r += p * pow32( t, n - 1 );
      p *= ( 1 + t );
      t *= t;
      n >>= 1;
   }
   r += p;
   return r;
}

// sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^32, where n = 2^e + c
uint32_t gs32( uint32_t a, uint32_t e, uint32_t c ) {

   if ( e == 0 ) return gs32( a, 1 + c );

   uint32_t t = a;
   uint32_t r = 1;

  for ( uint32_t i = 0; i < e; ++i ) {

      r = mul32( r, add32( 1, t ) );
      t = mul32( t, t );
   }
   if ( c == 0 ) return r;

   return add32( r, mul32( t, gs32( a, c ) ) );
}

// 64-bit methods

#ifdef UINT64_C // the following require 64-bit capability

// 64-bit computation of a + b mod m
uint64_t add_mod64( uint64_t a, uint64_t b, uint64_t m ) {

   a %= m;
   b %= m;
   uint64_t t;
   if ( b <= UINT64_MAX - a )
      return ( a + b ) % m;

   if ( m <= ( UINT64_MAX >> 1 ) )
      return ( ( a % m ) + ( b % m ) ) % m;

   t = a + b;
   if ( t > uint64_t( m * 2 ) ) // m*2 must be truncated before compare
      t -= m;
   t -= m;
   return t % m;
}

// 64-bit computation of a + b mod 2^64
uint64_t add64( uint64_t a, uint64_t b ) {

   return a + b;
}

// 64-bit computation of a * b mod m
uint64_t mul_mod64( uint64_t a, uint64_t b, uint64_t m ) {

   uint64_t r = 0;
   uint64_t t;

   if ( b >= m ) {

      if ( m > UINT64_MAX / 2u ) b -= m;
      else                       b %= m;
   }

   while ( a != 0 ) {

      if ( a & 1 ) {

         if ( b >= m - r ) r -= m;
         r += b;
      }
      a >>= 1;

      t = b;
      if ( b >= m - t ) t -= m;
      b += t;
   }
   return r;
}

// 64-bit computation of a * b mod 2^64
uint64_t mul64( uint64_t a, uint64_t b ) {

   uint64_t r = 0;
   uint64_t t;

   while ( a != 0 ) {

      if ( a & 1 ) r += b;
      a >>= 1;
      t = b;
      b += t;
   }
   return r;
}

// 64-bit computation of a^n mod m
uint64_t pow_mod64( uint64_t a, uintmax_t n, uint64_t m ) {

   if ( n == 0 ) return 1;
   if ( n == 1 ) return a %= m;

   uint64_t r = 1;
   uint64_t t = a;

   for (;;) {

      if ( n & 1 ) r = mul_mod64( r, t, m );
      n >>= 1;
      if ( n == 0 ) break;
      t = mul_mod64( t, t, m );
   }
   return r;
}

// 64-bit computation of a^n mod m, where n = 2^e + c
uint64_t pow_mod64( uint64_t a, uintmax_t e, uintmax_t c, uint64_t m ) {

   if ( e == 0 ) return pow_mod64( a, c + 1, m );
   uint64_t t = a;
   for ( uint64_t i = 0; i < e; ++i ) t = mul_mod64( t, t, m );
   return mul_mod64( pow_mod64( a, c, m ), t, m );
}

// a^n mod 2^64
uint64_t pow64( uint64_t a, uintmax_t n ) {

   uint64_t r = 1;
   uint64_t t = a;

   for (;;) {

      if ( n & 1 ) r *= t;
      n >>= 1;
      if ( n == 0 ) break;
      t *= t;
   }
   return r;
}

// a^n mod 2^64, where n = 2^e + c
uint64_t pow64( uint64_t a, uint64_t e, uint64_t c ) {

   if ( e == 0 ) return pow64( a, c + 1 );
   uint64_t t = a;
   for ( uint64_t i = 0; i < e; ++i ) t = mul64( t, t );
   return mul64( pow64( a, c ), t );
}

// 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
uint64_t gs_mod64( uint64_t a, uintmax_t n, uint64_t m ) {

   if ( n == 0 ) return 0;

   uint64_t t = a % m;
   uint64_t p = 1;
   uint64_t r = 0;

   while ( n > 1 ) {

      if ( n & 1 ) r = add_mod64( r, mul_mod64( p, pow_mod64( t, n - 1, m ), m ), m );
      p = mul_mod64( p, add_mod64( 1, t, m ), m );
      t = mul_mod64( t, t, m );
      n >>= 1;
   }
   r = add_mod64( r, p, m );
   return r;
}

// 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
uint64_t gs_mod64( uint64_t a, uint32_t e, uint32_t c, uint64_t m ) {

   if ( e == 0 ) return gs_mod64( a, 1 + c, m );

   uint64_t t = a;
   uint64_t r = 1;

  for ( uint32_t i = 0; i < e; ++i ) {

      r = mul_mod64( r, add_mod64( 1, t, m ), m );
      t = mul_mod64( t, t, m );
   }
   if ( c == 0 ) return r;

   return add_mod64( r, mul_mod64( t, gs_mod64( a, c, m ), m ), m );
}

// 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^64
uint64_t gs64( uint64_t a, uintmax_t n ) {

   if ( n == 0 ) return 0;
   if ( n == 1 ) return 1;

   uint64_t t = a;
   uint64_t p = 1;
   uint64_t r = 0;

   while ( n > 1 ) {

      if ( n & 1 ) r += mul64( p, pow64( t, n - 1 ) );
      p = mul64( p, 1 + t );
      t = mul64( t, t );
      n >>= 1;
   }
   r += p;
   return r;
}

// 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^64, where n = 2^e + c
uint64_t gs64( uint64_t a, uint64_t e, uint64_t c ) {

   if ( e == 0 ) return gs64( a, 1 + c );

   uint64_t t = a;
   uint64_t r = 1;

  for ( uint32_t i = 0; i < e; ++i ) {

      r = mul64( r, add64( 1, t ) );
      t = mul64( t, t );
   }
   if ( c == 0 ) return r;

   return add64( r, mul64( t, gs64( a, c ) ) );
}

#endif // UINT64_C

// compute a + b mod m, where a, b and m must be < 2^35
double add_mod( double a, double b, double m ) {

   assert( a < TWO35 && b < TWO35 && m < TWO35 );
   double v = a + b;
   uintmax_t a1;

   if ( v >= TWO53 || v <= -TWO53 ) {
      a1 = static_cast<uintmax_t>( a / TWO17 );
      a -= a1 * TWO17;
      v  = a1;
      a1 = static_cast<uintmax_t>( v / m );
      v -= a1 * m;
      v = v * TWO17 + a + b;
   }

   a1 = static_cast<uintmax_t>( v / m );
   if ( ( v -= a1 * m ) < 0. ) return v += m;
   else                        return v;
}

// a * b mod m, where a, b, and m must be < 2^35
double mul_mod( double a, double b, double m ) {

   assert( a < TWO35 && b < TWO35 && m < TWO35 );
   double v = a * b;
   uintmax_t a1;

   if ( v >= TWO53 || v <= -TWO53 ) {
      a1 = static_cast<uintmax_t>( a / TWO17 );
      a -= a1 * TWO17;
      v  = a1 * b;
      a1 = static_cast<uintmax_t>( v / m );
      v -= a1 * m;
      v = v * TWO17 + a * b;
   }

   a1 = static_cast<uintmax_t>( v / m );
   if ( ( v -= a1 * m ) < 0. ) return v += m;
   else                        return v;
}

// compute a^n mod m
double pow_mod( double a, uintmax_t n, double m ) {

   if ( n == 0 ) return 1.;
   if ( n == 1 ) return a;

   double r = 1.;
   double t = a;

   for (;;) {

      if ( n & 1 ) r = mul_mod( r, t, m );
      n >>= 1;
      if ( n == 0 ) break;
      t = mul_mod( t, t, m );
   }
   return r;
}

// compute a^n mod m, where n = 2^e + c
uint64_t pow_mod( double a, uint32_t e, uintmax_t c, double m ) {

   if ( e == 0 ) return pow_mod( a, c + 1, m );
   double t = a;
   for ( uint32_t i = 0; i < e; ++i ) t = mul_mod( t, t, m );
   return mul_mod( pow_mod( a, c, m ), t, m );
}

// sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
double gs_mod( double a, uintmax_t n, double m ) {

   if ( n == 0 ) return 0.;
   if ( n == 1 ) return 1.;

   double t = a;
   double p = 1.;
   double r = 0.;

   while ( n > 1 ) {

      if ( n & 1 ) r = add_mod( r, mul_mod( p, pow_mod( t, n - 1, m ), m ), m );
      p = mul_mod( p, 1. + t, m );
      t = mul_mod( t, t, m );
      n >>= 1;
   }
   r = add_mod( r, p, m );
   return r;
}

// sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
double gs_mod( double a, uint32_t e, uint32_t c, double m ) {

   if ( e == 0 ) return gs_mod( a, 1 + c, m );

   double t = a;
   double r = 1;

   for ( uint32_t i = 0; i < e; ++i ) {

      r = mul_mod( r, add_mod( 1, t, m ), m );
      t = mul_mod( t, t, m );
   }
   if ( c == 0 ) return r;

   return add_mod( r, mul_mod( t, gs_mod( a, c, m ), m ), m );
}
```

Listing D-2. mod_math.cpp

```
// mod_math.cpp: modular math for both 32-bit and 64-bit exponentiation and multiply

#include "mod_math.h"
#include <iostream>
#include <bitset>
#include <cassert>
using namespace std;

static const uint32_t _LC_MULT  = 69069ul;
static const uint32_t _LC_CONST = 12345ul;
static const uint32_t _LC_MULT_INV    = 2783094533ul;//0xa5e2a705

static const uint64_t LC_MULT    = 1490024343005336237ULL;
static const uint64_t LC_CONST   = 123456789ULL;
static const uint64_t A_INV      = 14241175500494512421ULL;

// jump back
uint32_t jump_back( uint32_t s1, uintmax_t n, double m ) {

   double a = mul_mod( pow_mod( _LC_MULT_INV, n, m ), add_mod(s1, -_LC_CONST, m ), m );
   double b = mul_mod( -_LC_CONST,    gs_mod( _LC_MULT_INV, n, m ), m );
   double c = add_mod( a, b, m );
   return static_cast<uint32_t>( add_mod( _LC_CONST, c, m ) );
}

// jump back 64-bit
uint64_t jump_back64( uint64_t s1, uintmax_t n ) {

   uint64_t p = mul64( pow64( A_INV, n ), s1 - LC_CONST );
   uint64_t q = mul64( -LC_CONST,    gs64( A_INV, n ) );
   s1 = p + q + LC_CONST;
   return s1;
}

// jump ahead n numbers
uint32_t jump_ahead( uint32_t s1, uintmax_t n, double m ) {

   double a = mul_mod( pow_mod( _LC_MULT, n, m ), s1, m );
   double b = mul_mod( _LC_CONST,    gs_mod( _LC_MULT, n, m ), m );
   return static_cast<uint32_t>( add_mod( a, b, m ) );
}

// jump ahead n numbers, where n = 2^e + c
uint32_t jump_ahead( uint32_t s1, uint32_t e, uint32_t c, double m ) {

   double a = mul_mod( pow_mod( _LC_MULT, e, c, m ), s1, m );
   double b = mul_mod( _LC_CONST,    gs_mod( _LC_MULT, e, c, m ), m );
   return static_cast<uint32_t>( add_mod( a, b, m ) );
}

int main( void ) {

// test cases from Mathematica:
   assert( jump_ahead( 123456789, 1024, TWO32 ) == 3928303893 );
   assert( jump_ahead( 123456789, 10, 0, TWO32 ) == 3928303893 );
   assert( jump_ahead( 123456789, 1000000, TWO32 ) == 410693845 );
   assert( jump_ahead( 123456789, 19, 475712, TWO32 ) == 410693845 );
   assert( jump_ahead( 123456789, 1000000000, TWO32 ) == 4013060885 );
   assert( jump_ahead( 123456789, 29, 463129088, TWO32 ) == 4013060885 );
   assert( jump_ahead( 123456789, 1073741824, TWO32 ) == 3344682261 );
   assert( jump_ahead( 123456789, 30, 0, TWO32 ) == 3344682261 );
   assert( jump_ahead( 0x7db7b9e0, 1024, TWO32 ) == 3683370464 );
   assert( jump_ahead( 0x7db7b9e0, 10, 0, TWO32 ) == 3683370464 );
   assert( jump_ahead( 0x7db7b9e0, 1000000, TWO32 ) == 3408994464 );
   assert( jump_ahead( 0x7db7b9e0, 19, 475712, TWO32 ) == 3408994464 );
   assert( jump_ahead( 0x7db7b9e0, 4294967295, TWO32 ) == 1473225027 );
   assert( jump_ahead( 0x7db7b9e0, 31, 2147483647, TWO32 ) == 1473225027 );
   assert( jump_ahead( 0x7db7b9e0, 4294967296ULL, TWO32 ) == 2109192672 );
   assert( jump_ahead( 0x7db7b9e0, 32, 0, TWO32 ) == 2109192672 );
```

```
    assert( jump_ahead( 0x7db7b9e0, 1099511627776ULL, TWO32 ) == 2109192672 );
    assert( jump_ahead( 0x7db7b9e0, 40, 0, TWO32 ) == 2109192672 );

    cout << "Test jump_ahead two different ways..." << endl;
    uint32_t s0 = 0xcafe1234;
    uint32_t s1 = s0;   // initialize
    uint32_t N = 1050;
    uint32_t e = 10, c = 26;

    for ( uint32_t i = 1; i <= N; i++ ) s1 = _LC_MULT * s1 + _LC_CONST;

    assert( jump_ahead( s0, N, TWO32 ) == s1 );
    assert( jump_ahead( s0, e, c, TWO32 ) == s1 );
    cout << "Passed jump_ahead." << endl;
    cout << "Test jump_back two different ways..." << endl;
    assert( jump_back( s1, N, TWO32 ) == s0 );
    //assert( jump_back( s1, e, c, TWO32 ) == s0 );
    cout << "Passed jump_back." << endl;

    cout << dec;

    assert( gs_mod64( 123456789, 0, 4294967296 ) == 0 );
    assert( gs_mod64( 123456789, 1, 4294967296 ) == 1 );
    assert( gs_mod64( 123456789, 10, 4294967296 )== 1382346382 );
    assert( gs_mod64( 123456789, 1024, 4294967296 ) == 3101645824 );
    assert( gs_mod64( 123456789, 1000000, 4294967296 ) == 2009531328 );

    assert( gs64( 1490024343005336237, 0 ) == 0 );
    assert( gs64( 1490024343005336237, 1 ) == 1 );
    assert( gs64( 1490024343005336237, 10 ) == 7987679512244350278 );
    assert( gs64( 1490024343005336237, 1024 ) == 9396580604419943424ULL );
    assert( gs64( 1490024343005336237, 12345 ) == 2047449762047247049 );

    // For a = 69069 and m = 2^32, a^(-1) is given by pow_mod64( 69069, 2147483647, 4294967296 )
    assert( mul_mod64( 69069, pow_mod64( 69069, 2147483647, 4294967296 ), 4294967296 ) == 1 );
    assert( pow_mod64( 69069, 2147483648, 4294967296 ) == 1 );

    // For a = 314527869 and m = 2^32, a^(-1) is given by pow_mod64( 314527869, 2147483647, 4294967296 )
    assert( mul_mod64( 314527869, pow_mod64( 314527869, 2147483647, 4294967296 ), 4294967296 ) == 1 );
    assert( pow_mod64( 314527869, 2147483648, 4294967296 ) == 1 );

    // For a = 1490024343005336237 and m = 2^64, a^(-1) is given by pow64( 1490024343005336237, 9223372036854775807 )
    assert( mul64( 1490024343005336237, pow64( 1490024343005336237, 9223372036854775807 ) ) == 1 );
    assert( pow64( 1490024343005336237, 9223372036854775808ULL ) == 1 );

    // For a = 698769069 and m = 3001190298811367423, a^(-1) is given by pow_mod64( 698769069, 3001190298811367421, 3001190298811367423 )
    assert( mul_mod64( 698769069, pow_mod64( 698769069, 3001190298811367421, 3001190298811367423 ), 3001190298811367423 ) == 1 );
    assert( pow_mod64( 698769069, 3001190298811367422, 3001190298811367423 ) == 1 );

    // For a = 4294584393 and m = 18445099517847011327, a^(-1) is given by pow_mod64( 4294584393, 18445099517847011325ULL, 18445099517847011327
          ULL )
    assert( mul_mod64( 4294584393, pow_mod64( 4294584393, 18445099517847011325ULL, 18445099517847011327ULL ), 18445099517847011327ULL ) == 1 );
    assert( pow_mod64( 4294584393, 18445099517847011326ULL, 18445099517847011327ULL ) == 1 );

    // For a = 4246477509 and m = 18445099517847011327, a^(-1) is given by pow_mod64( 4246477509, 18445099517847011325ULL, 18445099517847011327
          ULL )
    assert( mul_mod64( 4246477509, pow_mod64( 4246477509, 18445099517847011325ULL, 18445099517847011327ULL ), 18445099517847011327ULL ) == 1 );
    assert( pow_mod64( 4246477509, 18445099517847011326ULL, 18445099517847011327ULL ) == 1 );

    cout << dec;
    cout << "Start tests..." << endl;
    assert( add_mod32( 123456789, 987654321, 919 ) == 593 );
    assert( add_mod32( 123456789, 987654321, 4294967295 ) == 1111111110 );
    assert( add_mod64( 8446744073709551615ULL, 4467440737095516l5ULL, 18446744073709551ULL ) == 2157503891099648ULL );
    assert( add_mod64( 8446744073709551615ULL, 4467440737095516l5ULL, 987654321 ) == 147440801 );
    assert( add32( 4294967295UL, 1 ) == 0 );
    assert( add32( 4294967295UL, 4294967295UL ) == 4294967294 );
    assert( add64( 18446744073709551615ULL, 1 ) == 0 );
    assert( add64( 18446744073709551615ULL, 18446744073709551615ULL ) == 18446744073709551614ULL );

    assert( mul_mod32( 123456789, 987654321, 919 ) == 379 );
    assert( mul_mod32( 123456789, 987654321, 4294967295 ) == 4256203929 );
    assert( mul_mod64( 8446744073709551615ULL, 446744073709551615ULL, 18446744073709551ULL ) == 714732902007009ULL );
    assert( mul_mod64( 8446744073709551615ULL, 446744073709551615ULL, 987654321 ) == 906633840 );
    assert( mul32( 4294967295UL, 4294967295UL ) == 1 );
    assert( mul32( 123456789UL, 987654321UL ) == 4227814277 );
    assert( mul64( 18446744073709551615ULL, 18446744073709551615ULL ) == 1 );
    assert( mul64( 18446744073709551615ULL, 8446744073709551615ULL ) == 10000000000000000001ULL );

    assert( pow_mod32( 123456789, 0, 123456 ) == 1 );
    assert( pow_mod32( 123456789, 1, 123456 ) == 789 );
    assert( pow_mod32( 123456789, 10, 123456 ) == 54681 );
    assert( pow_mod32( 123456789, 100, 123456 ) == 30705 );
    assert( pow_mod32( 123456789, 1000, 123456 ) == 18273 );
    assert( pow_mod32( 123456789, 4294967296ULL, 123456 ) == 513 );
    assert( pow_mod32( 123456789, 1, 987654321 ) == 123456789 );
    assert( pow_mod32( 123456789, 0, 0, 987654321 ) == 123456789 );
    assert( pow_mod32( 123456789, 2, 987654321 ) == 478395063 );
    assert( pow_mod32( 123456789, 0, 1, 987654321 ) == 478395063 );

    assert( pow_mod64( 446744073709551616, 1048576, 123456 ) == 34624 );
    assert( pow_mod64( 446744073709551616, 20, 0, 123456 ) == 34624 );
    assert( pow_mod64( 123456789, 1, 987654321 ) == 123456789 );
    assert( pow_mod64( 123456789, 0, 0, 987654321 ) == 123456789 );
    assert( pow_mod64( 123456789, 2, 987654321 ) == 478395063 );
    assert( pow_mod64( 123456789, 0, 1, 987654321 ) == 478395063 );

    assert( pow32( 123456789, 0 ) == 1 );
    assert( pow32( 123456789, 1 ) == 123456789 );
    assert( pow32( 123456789, 0, 0 ) == 123456789 );
    assert( pow32( 123456789, 100 ) == 3584311345 );
    assert( pow32( 123456789, 6, 36 ) == 3584311345 );
    assert( pow32( 987654321, 317 ) == 1333802993 );
    assert( pow32( 123456789, 1048576 ) == 2092957697 );
    assert( pow32( 123456789, 20, 0 ) == 2092957697 );
    assert( pow32( 123456789, 19, 524288 ) == 2092957697 );

```

```
   assert( pow64( 123456789, 1048576 ) == 16544794250596843521ULL );
   assert( pow64( 123456789, 20, 0 ) == 16544794250596843521ULL );
   assert( pow64( 123456789, 19, 524288 ) == 16544794250596843521ULL );

   assert( gs_mod32( 123456789, 1000, 12345 ) == 4060 );
   assert( gs32( 123456789, 1000 ) == 3030896216 );

   assert( gs_mod64( 446744073709551616, 512, 987654321 ) == 852835532 );
   assert( gs_mod64( 446744073709551616, 9, 0, 987654321 ) == 852835532 );

   assert( gs64( 446744073709551616, 512 ) == 3452140635857354753ULL );
   assert( gs64( 446744073709551616, 9, 0 ) == 3452140635857354753ULL );

   assert( gs_mod64( 123456789, 64, 12345 ) == 3370 );
   assert( gs_mod64( 123456789, 6, 0, 12345 ) == 3370 );
   assert( gs_mod64( 123456789, 0, 63, 12345 ) == 3370 );

   assert( gs_mod64( 123456789, 67, 12345 ) == 5446 );
   assert( gs_mod64( 123456789, 6, 3, 12345 ) == 5446 );
   assert( gs_mod64( 123456789, 0, 66, 12345 ) == 5446 );

   assert( gs_mod64( 446744073709551616, 1024, 987654321 ) == 608654230 );
   assert( gs_mod64( 446744073709551616, 10, 0, 987654321 ) == 608654230 );
   assert( gs_mod64( 446744073709551616, 0, 1023, 987654321 ) == 608654230 );
   assert( gs_mod64( 446744073709551616, 9, 512, 987654321 ) == 608654230 );
   assert( gs_mod64( 446744073709551616, 8, 768, 987654321 ) == 608654230 );

   assert( gs64( 446744073709551616, 1024 ) == 3452140635857354753ULL );
   assert( gs64( 446744073709551616, 10, 0 ) == 3452140635857354753ULL );
   assert( gs64( 446744073709551616, 0, 1023 ) == 3452140635857354753ULL );
   assert( gs64( 446744073709551616, 9, 512 ) == 3452140635857354753ULL );
   assert( gs64( 446744073709551616, 8, 768 ) == 3452140635857354753ULL );

   assert( gs_mod32( 123456789, 1024, 12345 ) == 7165 );
   assert( gs_mod32( 123456789, 10, 0, 12345 ) == 7165 );
   assert( gs_mod32( 123456789, 0, 1023, 12345 ) == 7165 );
   assert( gs_mod32( 123456789, 9, 512, 12345 ) == 7165 );
   assert( gs_mod32( 123456789, 8, 768, 12345 ) == 7165 );

   assert( gs32( 123456789, 1024 ) == 3101645824 );
   assert( gs32( 123456789, 10, 0 ) == 3101645824 );
   assert( gs32( 123456789, 0, 1023 ) == 3101645824 );
   assert( gs32( 123456789, 9, 512 ) == 3101645824 );
   assert( gs32( 123456789, 8, 768 ) == 3101645824 );

   assert( gs_mod( 123456789, 64, 12345 ) == gs_mod64( 123456789, 64, 12345 ) );
   assert( gs_mod( 123456789, 6, 0, 12345 ) == gs_mod64( 123456789, 6, 0, 12345 ) );
   assert( gs_mod( 123456789, 0, 63, 12345 ) == gs_mod( 123456789, 0, 63, 12345 ) );

   assert( pow_mod( 123456789, 1048576, 123456 ) == pow_mod64( 123456789, 1048576, 123456 ) );
   assert( pow_mod( 123456789, 20, 0, 123456 ) == pow_mod64( 123456789, 20, 0, 123456 ) );

   cout << "All tests passed." << endl;

   return 0;
}
```

Listing D-3. Bitmatrix.h

```
// Bitmatrix.h: template class for 32 x 32 or 64 x 64 matrices using mod 2 arithmetic
// R. Saucier, August 2016

#ifndef BITMATRIX_H
#define BITMATRIX_H

#include <cstdint>   // for uint32_t and uint64_t
#include <climits>   // for CHAR_BIT, the number of bits per byte

typedef struct { uint32_t matrix[32]; } bitmatrix32_t;
typedef struct { uint64_t matrix[64]; } bitmatrix64_t;

template <class T>
class Bitmatrix {

public:

   static const unsigned int N_BITS = CHAR_BIT * sizeof( T );   // number of bits

public:

   Bitmatrix( void ) {   // default constructor
   }

   Bitmatrix( const bitmatrix32_t& A ) {   // constructor from array of 32-bit constants

      for ( T i = 0; i < N_BITS; i++ ) _matrix[i] = A.matrix[i];
   }

   Bitmatrix( const bitmatrix64_t& A ) {   // constructor from array of 64-bit constants

      for ( T i = 0; i < N_BITS; i++ ) _matrix[i] = A.matrix[i];
   }

  ~Bitmatrix( void ) {   // default destructor
   }

   Bitmatrix( const Bitmatrix& A ) {   // copy constructor

      for ( T i = 0; i < N_BITS; i++ ) _matrix[i] = A._matrix[i];
   }

   Bitmatrix& operator=( const Bitmatrix& A ) {   // assignment operator

      if ( this != &A ) for ( T i = 0; i < N_BITS; i++ ) _matrix[i] = A._matrix[i];
      return *this;
   }

   void identity( Bitmatrix& A ) {   // create an identity matrix

      T v = T(1);
      for ( T i = 0; i < N_BITS; i++, v <<= 1 ) A._matrix[i] = v;
   }

   T matrix( T i ) {   // return the ith vector of the bitmatrix

      return _matrix[i];
   }

   // overloaded operators

   friend T operator*( const Bitmatrix<T>& A, T v ) {   // matrix multiplication of a vector

      T r = T(0);
      T b = T(1);
      for ( T i = 0; i < N_BITS; i++, v >>= 1 ) if ( v & b ) r ^= A._matrix[i];
      return r;
   }

   friend Bitmatrix operator*( Bitmatrix<T>& A, const Bitmatrix<T>& B ) {   // multiplication of two Bitmatrices

      Bitmatrix<T> C;

      for ( T i = 0; i < N_BITS; i++ ) C._matrix[i] = A * B._matrix[i];
      return C;
   }

   Bitmatrix& operator*=( const Bitmatrix<T>& A ) {   // multiplication assignment of two Bitmatrices

      return *this = *this * A;
   }

   Bitmatrix operator^( T n ) {   // return A^n, Bitmatrix A to the power n

      Bitmatrix<T> B, A = *this;

      identity( B );
      T b = T(1);

      while ( n > 0 ) {   // Knuth's "exponentiation by squaring" algorithm

         if ( n & b ) B *= A;
         A *= A;
         n >>= 1;
      }
      return B;
   }

   friend Bitmatrix pow( Bitmatrix<T>& A, T e, T c ) {   // return A^n, Bitmatrix A to the power n, where n = 2^e + c

      Bitmatrix<T> B;//, A = *this;
      if ( e > 0 ) {
         B = A;
         for ( T i = 0; i < e; i++ ) B *= B;
      }
      A = A^c;
      if ( e ) A *= B;
      return A;
   }

private:

   T _matrix[N_BITS];
};
   // declaration of friends
   //void identity( Bitmatrix<uint32_t>& A );
   uint32_t operator*( const Bitmatrix<uint32_t>& A, uint32_t v );
   Bitmatrix<uint32_t> operator*( Bitmatrix<uint32_t>& A, const Bitmatrix<uint32_t>& B );
   Bitmatrix<uint32_t> pow( Bitmatrix<uint32_t>& A, uint32_t e, uint32_t c );

   //void identity( Bitmatrix<uint64_t>& A );
   uint64_t operator*( const Bitmatrix<uint64_t>& A, uint32_t v );
   Bitmatrix<uint64_t> operator*( Bitmatrix<uint64_t>& A, const Bitmatrix<uint64_t>& B );


#endif // BITMATRIX_H
```

Listing D-4. lfsr88.h

```
// lfsr88.h: 32-bit Random number generator U[0,1): lfsr88
// Cycle length is (2^31 - 1)(2^29 - 1)(2^28 - 1) = 309485007947847626691444735 or approximately 2^88
// Author: Pierre L'Ecuyer,
// Source:  http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tausme2.ps
// R. Saucier, December 2016

#ifndef LFSR88_H
#define LFSR88_H

namespace LFSR88 {

static const uint32_t N_SEEDS = 3;
static const bitmatrix32_t MATRIX[N_SEEDS] = {
   {
      {
         0x00000000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040001, 0x00080002,
         0x00100004, 0x00200008, 0x00400010, 0x00800020, 0x01000040, 0x02000080, 0x04000100, 0x08000200,
         0x10000400, 0x20000800, 0x40001000, 0x80000001, 0x00000002, 0x00000004, 0x00000008, 0x00000010,
         0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800, 0x00001000
      }
   },
   {
    {
      0x00000000, 0x00000000, 0x00000000, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800,
      0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000, 0x00080000,
      0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x04000000, 0x08000001,
      0x10000002, 0x20000005, 0x4000000A, 0x80000014, 0x00000028, 0x00000050, 0x00000020, 0x00000040
    }
  },
  {
    {
      0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00200000, 0x00400000, 0x00800000, 0x01000000,
      0x02000001, 0x04000002, 0x08000004, 0x10000009, 0x20000012, 0x40000024, 0x80000048, 0x00000090,
      0x00000120, 0x00000240, 0x00000480, 0x00000900, 0x00001200, 0x00002400, 0x00004800, 0x00009000,
      0x00012000, 0x00024000, 0x00048000, 0x00090000, 0x00120000, 0x00040000, 0x00080000, 0x00100000
    }
  }
};
static const bitmatrix32_t MATRIX_INV[N_SEEDS] = {
  {
    {
      0x00000000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x04000000,
      0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000001, 0x00000002, 0x00000004, 0x00000008,
      0x00000010, 0x00000020, 0x00000040, 0x00100080, 0x00200100, 0x00400200, 0x00800400, 0x01000800,
      0x02001000, 0x04002000, 0x08004000, 0x10008000, 0x20010000, 0x40020000, 0x80040000, 0x00080000
    }
  },
  {
    {
      0x00000000, 0x00000000, 0x00000000, 0x50000000, 0xa0000001, 0x40000002, 0x80000004, 0x00000008,
      0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800,
      0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000, 0x00080000,
      0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x54000000, 0xa8000000
    }
  },
  {
    {
      0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x49248000, 0x92490000, 0x24920000, 0x49240000,
      0x92480000, 0x24900000, 0x49200000, 0x92400000, 0x24800000, 0x49000000, 0x92000000, 0x24000000,
      0x48000000, 0x90000001, 0x20000002, 0x40000004, 0x80000008, 0x00000010, 0x00000020, 0x00000040,
      0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800, 0x49249000, 0x92492000, 0x24924000
    }
  }
};
static const uint32_t C0 = 0xfffffffful;   // 4294967295ul
static const uint32_t C1 = 0xfffffffeul;   // 4294967294ul
static const uint32_t C2 = 0xfffffff8ul;   // 4294967288ul
static const uint32_t C3 = 0xfffffff0ul;   // 4294967280ul

class lfsr88 : public Generator<uint32_t> {

public:
  lfsr88 ( void ) { // default constructor
  }

  lfsr88 ( std::vector<uint32_t> seed ) { // constructor from vector seed

    setState( seed );
  }

  virtual ~lfsr88() { // default destructor

    std::cout << "deleting lfsr88" << std::endl;
  }

  virtual void setState( std::vector<uint32_t> seed ) { // set the seeds

    assert( seed.size() >= N_SEEDS );

    // VERY IMPORTANT: The initial seeds _s[0], _s[1], _s[2] MUST be larger than 1, 7, and 15 respectively
    _s[0] = seed[0]; if ( _s[0] <  2 ) _s[0] +=  2;
    _s[1] = seed[1]; if ( _s[1] <  8 ) _s[1] +=  8;
    _s[2] = seed[2]; if ( _s[2] < 16 ) _s[2] += 16;
  }

  virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector

    assert( seed.size() >= N_SEEDS );
    for ( size_t i = 0; i < N_SEEDS; i++ ) seed[i] = _s[i];
  }

  virtual void jump_ahead( uintmax_t n ) {   // jump ahead the next n random numbers

    for ( size_t i = 0; i < N_SEEDS; i++ ) {

      Bitmatrix<uint32_t> A( MATRIX[i] );
      _s[i] = ( A^n ) * _s[i];
    }
  }

  virtual void jump_ahead( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

    if ( e == 0 && c == 0 ) return jump_ahead( 1 );

    Bitmatrix<uint32_t> A, B;

    for ( size_t i = 0; i < N_SEEDS; i++ ) {

      if ( e ) {
        B = MATRIX[i];
        for ( uintmax_t j = 0; j < e; j++ ) B *= B;
      }
      A = MATRIX[i];
      A = A^c;
      if ( e ) A *= B;
      _s[i] = A * _s[i];
    }
  }

```

```
   virtual void jump_back( uintmax_t n ) {   // jump ahead the next n random numbers

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         Bitmatrix<uint32_t> A( MATRIX_INV[i] );
         _s[i] = ( A^n ) * _s[i];
      }
   }

   virtual void jump_back( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_back( 1 );

      Bitmatrix<uint32_t> A, B;

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         if ( e ) {
            B = MATRIX_INV[i];
            for ( uintmax_t j = 0; j < e; j++ ) B *= B;
         }
         A = MATRIX_INV[i];
         A = A^c;
         if ( e ) A *= B;
         _s[i] = A * _s[i];
      }
   }

   virtual void jump_cycle( void ) {

      const uint32_t A = 31, B = 29, C = 28;
      jump_ahead( A + B + C, 0 );
      jump_back( A + B, 0 );
      jump_back( A + C, 0 );
      jump_back( B + C, 0 );
      jump_ahead( A, 0 );
      jump_ahead( B, 0 );
      jump_ahead( C, 0 );
      jump_back( 1 );
   }

   virtual uint32_t rng32( void ) {   // returns 32-bit integer

      _s[0] = ( ( _s[0] & C1 ) << 12 ) ^ ( ( ( _s[0] << 13 ) ^ _s[0] ) >> 19 ) ;
      _s[1] = ( ( _s[1] & C2 ) <<  4 ) ^ ( ( ( _s[1] <<  2 ) ^ _s[1] ) >> 25 ) ;
      _s[2] = ( ( _s[2] & C3 ) << 17 ) ^ ( ( ( _s[2] <<  3 ) ^ _s[2] ) >> 11 ) ;

      return ( _s[0] ^ _s[1] ^ _s[2] ) & C0;
   }

   virtual uint64_t rng64( void ) {   // returns 64-bit integer

      uint64_t low  = rng32();
      uint64_t high = rng32();
      return low | ( high << 32 );
   }

   virtual double rng32_01( void ) {   // returns a double in [0,1)

      return double( rng32() ) * TWO32_INV;
   }

   virtual long double rng64_01( void ) {   // returns a long double in [0,1)

      return double( rng64() ) * TWO64_INV;
   }

private:

   uint32_t _s[ N_SEEDS ];

}; // end lfsr88 class
} // end namespace LFSR88

#endif // LFSR88_H
```

Listing D-5. lfsr113.h

```
// 32-bit Random number generator U[0,1): lfsr113
// Period is (2^31 - 1)(2^29 - 1)(2^28 - 1)(2^25 - 1) = 10384593344720504788331840650870785 or approximately 2^113
// Author: Pierre L'Ecuyer,
// Source: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tausme2.ps
// R. Saucier, December 2016

#ifndef LFSR113_H
#define LFSR113_H

namespace LFSR113 {

static const uint32_t N_SEEDS = 4;
static const bitmatrix32_t MATRIX[N_SEEDS] = {
   {
      {
         0x00000000, 0x00080000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000001,
         0x04000002, 0x08000004, 0x10000008, 0x20000010, 0x40000020, 0x80000041, 0x00000082, 0x00000104,
         0x00000208, 0x00000410, 0x00000820, 0x00001040, 0x00002080, 0x00004100, 0x00008200, 0x00010400,
         0x00020800, 0x00041000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000
      }
   },
   {
      {
         0x00000000, 0x00000000, 0x00000000, 0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200,
         0x00000400, 0x00000800, 0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000,
         0x00040000, 0x00080000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000,
         0x04000000, 0x08000001, 0x10000002, 0x20000005, 0x4000000a, 0x80000014, 0x00000008, 0x00000010
      }
   },
   {
      {
         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000800, 0x00001000, 0x00002000, 0x00004000,
         0x00008001, 0x00010002, 0x00020004, 0x00040008, 0x00080010, 0x00100020, 0x00200040, 0x00400080,
         0x00800100, 0x01000200, 0x02000400, 0x04000000, 0x08000000, 0x10000001, 0x20000002, 0x40000004,
         0x80000008, 0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400
      }
   },
   {
      {
         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00100000,
         0x00200000, 0x00400001, 0x00800002, 0x01000004, 0x02000009, 0x04000012, 0x08000024, 0x10000048,
         0x20000090, 0x40000120, 0x80000240, 0x00000480, 0x00000900, 0x00001200, 0x00002400, 0x00004800,
         0x00009000, 0x00012000, 0x00024000, 0x00048000, 0x00090000, 0x00020000, 0x00040000, 0x00080000
      }
   }
};
static const bitmatrix32_t MATRIX_INV[N_SEEDS] = {
   {
      {
         0x00000000, 0x04104000, 0x08208000, 0x10410000, 0x20820000, 0x41040000, 0x82080000, 0x04100000,
         0x08200000, 0x10400000, 0x20800000, 0x41000000, 0x82000000, 0x04000000, 0x08000000, 0x10000000,
         0x20000000, 0x40000000, 0x80000001, 0x00000002, 0x00000004, 0x00000008, 0x00000010, 0x00000020,
         0x00000040, 0x00000080, 0x04104100, 0x08208200, 0x10410400, 0x20820800, 0x41041000, 0x82082000
      }
   },
   {
      {
         0x00000000, 0x00000000, 0x00000000, 0x40000002, 0x80000004, 0x00000008, 0x00000010, 0x00000020,
         0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800, 0x00001000, 0x00002000,
         0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000, 0x00080000, 0x00100000, 0x00200000,
         0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x04000000, 0x08000001, 0x50000000, 0xa0000001
      }
   },
   {
      {
         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x02000000, 0x04000000, 0x08000000, 0x10000001,
         0x20000002, 0x40000004, 0x80000008, 0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000100,
         0x00000200, 0x00000400, 0x00000800, 0x02001000, 0x04002000, 0x08004000, 0x10008000, 0x20010000,
         0x40020000, 0x80040000, 0x00080000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000
      }
   },
   {
      {
         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x92480000,
         0x24900000, 0x49200000, 0x92400000, 0x24800000, 0x49000000, 0x92000001, 0x24000002, 0x48000004,
         0x90000008, 0x20000010, 0x40000020, 0x80000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400,
         0x00000800, 0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x92490000, 0x24920000, 0x49240000
      }
   }
};
static const uint32_t C0 = 0xfffffffful;   // 4294967295ul
static const uint32_t C1 = 0xfffffffeul;   // 4294967294ul
static const uint32_t C2 = 0xfffffff8ul;   // 4294967288ul
static const uint32_t C3 = 0xfffffff0ul;   // 4294967280ul
static const uint32_t C4 = 0xffffff80ul;   // 4294967168ul

class lfsr113 : public Generator<uint32_t> {

public:
   lfsr113 ( void ) { // default constructor
   }

   lfsr113 ( std::vector<uint32_t> seed ) { // constructor from vector seed

      setState( seed );
   }

   virtual ~lfsr113() { // default destructor

      std::cout << "deleting lfsr113" << std::endl;
   }

   virtual void setState( std::vector<uint32_t> seed ) { // set the seeds

      assert( seed.size() >= N_SEEDS );

      // VERY IMPORTANT: The initial seeds _s1, _s2, _s3 MUST be larger than 1, 7, 15, and 127 respectively
      _s[0] = seed[0]; if ( _s[0] <   2 ) _s[0] +=   2;
      _s[1] = seed[1]; if ( _s[1] <   8 ) _s[1] +=   8;
      _s[2] = seed[2]; if ( _s[2] <  16 ) _s[2] +=  16;
      _s[3] = seed[3]; if ( _s[3] < 128 ) _s[3] += 128;
   }

   virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector

      assert( seed.size() >= N_SEEDS );
      for ( size_t i = 0; i < N_SEEDS; i++ ) seed[i] = _s[i];
   }

   virtual void jump_ahead( uintmax_t n ) { // jumps ahead the next n random numbers

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         Bitmatrix<uint32_t> A( MATRIX[i] );
         _s[i] = ( A^n ) * _s[i];
      }
   }

   virtual void jump_ahead( uintmax_t e, uintmax_t c ) {   // jumps ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) jump_ahead( 1 );

      Bitmatrix<uint32_t> A, B;

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         if ( e ) {
            B = MATRIX[i];
            for ( uintmax_t j = 0; j < e; j++ ) B *= B;
         }
         A = MATRIX[i];
         A = A^c;
         if ( e ) A *= B;
         _s[i] = A * _s[i];
      }
   }

   virtual void jump_back( uintmax_t n ) { // jumps ahead the next n random numbers

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         Bitmatrix<uint32_t> A( MATRIX_INV[i] );
         _s[i] = ( A^n ) * _s[i];
      }
   }

   virtual void jump_back( uintmax_t e, uintmax_t c ) {   // jumps ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_back( 1 );

      Bitmatrix<uint32_t> A, B;

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         if ( e ) {
            B = MATRIX_INV[i];
            for ( uintmax_t j = 0; j < e; j++ ) B *= B;
         }
         A = MATRIX_INV[i];
         A = A^c;
         if ( e ) A *= B;
         _s[i] = A * _s[i];
      }
   }

   virtual void jump_cycle( void ) { // jump ahead an entire cycle of lfsr113

      const uint32_t A = 31, B = 29, C = 28, D = 25;
      jump_ahead( A + B + C + D, 0 );
      jump_back( A + B + C, 0 );
      jump_back( A + B + D, 0 );
      jump_back( A + C + D, 0 );
      jump_back( B + C + D, 0 );
      jump_ahead( A + B, 0 );
      jump_ahead( A + C, 0 );
      jump_ahead( A + D, 0 );
      jump_ahead( B + C, 0 );
      jump_ahead( B + D, 0 );
      jump_ahead( C + D, 0 );
      jump_back( A, 0 );
      jump_back( B, 0 );
      jump_back( C, 0 );
      jump_back( D, 0 );
      jump_ahead( 1 );
   }

   virtual uint32_t rng32( void ) {   // returns the next number (a 32-bit unsigned int)

      _s[0] = ( ( _s[0] & C1 ) << 18 ) ^ ( ( ( _s[0] <<  6 ) ^ _s[0] ) >> 13 );
      _s[1] = ( ( _s[1] & C2 ) <<  2 ) ^ ( ( ( _s[1] <<  2 ) ^ _s[1] ) >> 27 );
      _s[2] = ( ( _s[2] & C3 ) <<  7 ) ^ ( ( ( _s[2] << 13 ) ^ _s[2] ) >> 21 );
      _s[3] = ( ( _s[3] & C4 ) << 13 ) ^ ( ( ( _s[3] <<  3 ) ^ _s[3] ) >> 12 );

      return ( _s[0] ^ _s[1] ^ _s[2] ^ _s[3] ) & C0;
   }

   virtual uint64_t rng64( void ) {   // returns 64-bit integer

      uint64_t low  = rng32();
      uint64_t high = rng32();
      return low | ( high << 32 );
   }

   virtual double rng32_01( void ) {   // returns a double int the half-open interval [0,1)

      return double( rng32() ) * TWO32_INV;
   }

   virtual long double rng64_01( void ) {   // returns a long double in [0,1)

      return double( rng64() ) * TWO64_INV;
   }

private:

   uint32_t _s[ N_SEEDS ];

}; // end lfsr113 class
} // end namespace LFSR113

#endif // LFSR113_H
```

Listing D-6. kiss.h

```
// kiss.h: Marsaglia's Keep It Simple Stupid RNG,
// which consists of a combination of linear congruential, 3-shift-register, and multiply with carry.
// Period is (2^32)(2^32-1)(698769069(2^31)-1) = 27681094672891588090390813844460011520, approximately 2^124.
// R. Saucier, December 2016

```

```
#ifndef KISS_H
#define KISS_H
#include <bitset>

namespace KISS {

   static const bitmatrix32_t MATRIX = {
      {
         0x00042021, 0x00084042, 0x00108084, 0x00210108, 0x00420231, 0x00840462, 0x010808C4, 0x02101188,
         0x04202310, 0x08404620, 0x10808C40, 0x21011880, 0x42023100, 0x84046200, 0x0808C400, 0x10118800,
         0x20231000, 0x40462021, 0x808C4042, 0x01080084, 0x02100108, 0x04200210, 0x08400420, 0x10800840,
         0x21001080, 0x42002100, 0x84004200, 0x08008400, 0x10010800, 0x20021000, 0x40042000, 0x80084000
      }
   };
   static const bitmatrix32_t MATRIX_INV = {
      {
         0xf2b58529, 0xe56b0a52, 0xded6b4a5, 0xbdad694a, 0x7b5ad294, 0xf6b5a528, 0xed6b4a50, 0xced634a1,
         0x9dac6942, 0x3b58d284, 0x76b1a508, 0xed634a10, 0xcec63421, 0x9d8c6842, 0x3b18d084, 0x7631a108,
         0xec634210, 0xccc62421, 0x998c4842, 0x33189084, 0x66312108, 0xcc624210, 0x88c40420, 0x11880840,
         0x23101080, 0x46202100, 0x8c404200, 0x08800400, 0x11000800, 0x22001000, 0x44002000, 0x88004000
      }
   };
   static const uint32_t LC_MULT         = 0x00010dcd;           // 69069UL
   static const uint32_t LC_CONST        = 0x00003039;           // 12345UL
   static const uint32_t LC_MULT_INV     = 0xa5e2a705;           // 2783094533UL
   static const uint64_t MWC_MULT        = 0x0000000029a65ead;   // 698769069ull;
   static const uint64_t MWC_MOD         = 0x29a65eacffffffff;   // 3001190298811367423ULL
   static const uint64_t MWC_MULT_INV    = 0x0000000100000000;   // 4294967296ULL;
   static const uint32_t N_SEEDS         = 4;                    // requires four 32-bit seeds

class kiss : public Generator<uint32_t> {

public:
   kiss( void ) { // default constructor
   }

   kiss( std::vector<uint32_t> seed ) { // constructor from seed vector

      setState( seed );
   }

   virtual ~kiss() {   // default destructor

      std::cout << "deleting kiss" << std::endl;
   }

   virtual void setState( std::vector<uint32_t> seed ) { // set the seeds

      assert( seed.size() >= N_SEEDS );
      _s1 = seed[0];
      _s2 = seed[1];
      _s3 = seed[2];
      _s4 = seed[3];
   }

   virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector

      assert( seed.size() >= N_SEEDS );
      seed[0] = _s1;
      seed[1] = _s2;
      seed[2] = _s3;
      seed[3] = _s4;
   }

   virtual void jump_ahead( uintmax_t n ) { // jumps ahead the next n random numbers

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT, n, M ), _s1, M );
      uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, n, M ), M );
      _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT, n, M ), _s1, M );
      double q = mul_mod( LC_CONST, gs_mod( LC_MULT, n, M ), M );
      _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );

#endif // UINT64_C

      Bitmatrix<uint32_t> A, B( MATRIX );
      A = B^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT, n, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_ahead( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_ahead( 1 );

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT, e, c, M ), _s1, M );
      uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, e, c, M ), M );
      _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT, e, c, M ), _s1, M );
      double q = mul_mod( LC_CONST, gs_mod( LC_MULT, e, c, M ), M );
      _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );

#endif // UINT64_C
```

```

      Bitmatrix<uint32_t> A, B;

      if ( e ) {
         B = MATRIX;
         for ( size_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT, e, c, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_back( uintmax_t n ) { // jump back n

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT_INV, n, M ), add_mod64( _s1, -LC_CONST, M ), M );
      uint64_t q = mul_mod64( -LC_CONST, gs_mod64( LC_MULT_INV, n, M ), M );
      uint64_t r = add_mod64( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT_INV, n, M ), add_mod( _s1, -LC_CONST, M ), M );
      double q = mul_mod( -LC_CONST, gs_mod( LC_MULT_INV, n, M ), M );
      double r = add_mod( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );

#endif // UINT64_C

      Bitmatrix<uint32_t> A( MATRIX_INV );
      A = A^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT_INV, n, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_back( uintmax_t e, uintmax_t c ) {   // jump back the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_back( 1 );

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT_INV, e, c, M ), add_mod64( _s1, -LC_CONST, M ), M );
      uint64_t q = mul_mod64( -LC_CONST, gs_mod64( LC_MULT_INV, e, c, M ), M );
      uint64_t r = add_mod64( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod64( LC_CONST, r, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT_INV, e, c, M ), add_mod( _s1, -LC_CONST, M ), M );
      double q = mul_mod( -LC_CONST, gs_mod( LC_MULT_INV, e, c, M ), M );
      double r = add_mod( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );

#endif // UINT64_C

      Bitmatrix<uint32_t> A, B;

      if ( e ) {
         B = MATRIX_INV;
         for ( size_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX_INV;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT_INV, e, c, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_cycle( void ) { // jump ahead a full cycle of kiss

      std::bitset<125> p( std::string( "10100110100110010111101010110011010110010110011010" ) +
                          std::string( "00010101000100000000000000000000000000000010000000" ) +
                          std::string( "0000000000000000000000000" ) );
      for ( size_t i = 0; i < p.size(); ++i ) if ( p.test(i) ) jump_ahead( i, 0 );
   }

   virtual uint32_t rng32( void ) { // returns the next random number (as a 32-bit unsigned int)

      _s1 = LC_MULT * _s1 + LC_CONST;

      _s2 ^= ( _s2 << 13 ), _s2 ^= ( _s2 >> 17 ), _s2 ^= ( _s2 << 5 );

      uint64_t a = MWC_MULT * _s3 + _s4;
      _s4 = ( a >> 32u );
      _s3 = uint32_t( a );

      return _s1 + _s2 + _s3;
   }

   virtual uint64_t rng64( void ) {   // returns 64-bit integer

      uint64_t low  = rng32();
      uint64_t high = rng32();
      return low | ( high << 32 );
   }

   virtual double rng32_01( void ) { // returns a random number in the half-open interval [0,1)

      return double( rng32() ) * TWO32_INV;
   }

   virtual long double rng64_01( void ) {   // returns a long double in [0,1)

      return double( rng64() ) * TWO64_INV;
   }

private:

   uint32_t _s1, _s2, _s3, _s4;

}; // end kiss class
} // end namespace KISS

#endif // KISS_H
```

Listing D-7. jkiss.h

```
// jkiss.h: Based upon Marsaglia’s Keep It Simple Stupid RNG
// Ref: Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications
//      David Jones, UCL Bioinformatics Group (d.jones@cs.ucl.ac.uk), May 7, 2010
// Period is (2^32)(2^32-1)(4294584393(2^31)-1) = 170126015070303082434102628274311004l6 or approximately 2^127
// Period of MWC is 4294584393(2^31)-1 = 9222549758923505663
// R. Saucier, December 2016

#ifndef JKISS_H
#define JKISS_H

namespace JKISS {

   static const bitmatrix32_t MATRIX = { // 32x32 bitmatrix
      {
         0x08400021, 0x10800042, 0x21400085, 0x4280010a, 0x85000214, 0x0a000428, 0x14000850, 0x284010a1,
         0x50802142, 0xa1004284, 0x42008508, 0x84010a10, 0x08021420, 0x10042840, 0x20085080, 0x4010a100,
         0x80214200, 0x00428400, 0x00850800, 0x010a1000, 0x02142000, 0x04284000, 0x08508000, 0x10a10000,
         0x21420000, 0x42840000, 0x85080000, 0x08100000, 0x10200000, 0x20400000, 0x40800000, 0x81000000
      }
   };
   // A_INV = A + A^7 + A^9 + A^10 + A^11 + A^13 + A^19 + A^20 + A^21 + A^22 + A^23 + A^31, where A = MATRIX
   static const bitmatrix32_t MATRIX_INV = {
      {
         0x9ce52d63, 0x39ca5ac6, 0x7394b58c, 0xe7296b18, 0xce52d630, 0x9ca5ac60, 0x7b5bdce1, 0xb4a73de3,
         0x694e7bc6, 0xd29cf78c, 0x5294a508, 0xa5294a10, 0x4a529420, 0x94a52840, 0x6b5ad4a1, 0xd6b5a942,
         0xad6b5284, 0x5ad6a508, 0xb5ad4a10, 0x6b5a9420, 0xd6b52840, 0xef7ad4a1, 0xdef5a942, 0xbdeb5284,
         0x7bd6a508, 0xf7ad4a10, 0xef5a9420, 0xdeb52840, 0xff7ad4a1, 0xfef5a942, 0xfdeb5284, 0xfbd6a508
      }
   };
   static const uint32_t LC_MULT      = 0x12bf507dul;            // 314527869ul;
   static const uint32_t LC_CONST     = 0x0012d687ul;            // 1234567ul;
   static const uint32_t LC_MULT_INV  = 0x6200a8d5ul;            // 1644210389ul;
   static const uint64_t MWC_MULT     = 0x00000000fffa2849ull;   // 4294584393ull;
   static const uint64_t MWC_MOD      = 0xfffa2848ffffffffull;   // 18445099517847011327ull;
   static const uint64_t MWC_MULT_INV = 0x0000000100000000ull;   // 4294967296ull;
   static const uint64_t LC_PERIOD    = 0x0000000100000000ull;   // 4294967296ull;
   static const uint64_t SR_PERIOD    = 0x00000000ffffffffull;   // 4294967295ull;
   static const uint64_t MWC_PERIOD   = 0x7ffd14247fffffffull;   // 9222549758923505663ull;
   static const uint32_t N_SEEDS      = 4;                       // requires four 32-bit words

class jkiss : public Generator<uint32_t> {

public:
   jkiss( void ) { // default constructor
   }

   jkiss( std::vector<uint32_t> seed ) { // constructor from seed vector

      setState( seed );
   }

   virtual ~jkiss() { // default destructor

      std::cout << "deleting jkiss" << std::endl;
   }

   virtual void setState( std::vector<uint32_t> seed ) { // set the seeds

      assert( seed.size() >= N_SEEDS );
      _s1 = seed[0];
      _s2 = seed[1];
      _s3 = seed[2];
      _s4 = seed[3];
   }

   virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector

      assert( seed.size() >= N_SEEDS );
      seed[0] = _s1;
      seed[1] = _s2;
      seed[2] = _s3;
      seed[3] = _s4;
   }

   virtual void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT, n, M ), _s1, M );
      uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, n, M ), M );
      _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT, n, M ), _s1, M );
      double q = mul_mod( LC_CONST, gs_mod( LC_MULT, n, M ), M );
      _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );

#endif // UINT64_C

      Bitmatrix<uint32_t> A, B( MATRIX );
      A = B^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT, n, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_ahead( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_ahead( 1 );

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT, e, c, M ), _s1, M );
      uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, e, c, M ), M );
      _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT, e, c, M ), _s1, M );
      double q = mul_mod( LC_CONST, gs_mod( LC_MULT, e, c, M ), M );
      _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );

#endif // UINT64_C

      Bitmatrix<uint32_t> A, B;

      if ( e ) {
         B = MATRIX;
         for ( size_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT, e, c, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_back( uintmax_t n ) { // jump back the next n random numbers

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT_INV, n, M ), add_mod64( _s1, -LC_CONST, M ), M );
      uint64_t q = mul_mod64( -LC_CONST, gs_mod64( LC_MULT_INV, n, M ), M );
      uint64_t r = add_mod64( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT_INV, n, M ), add_mod( _s1, -LC_CONST, M ), M );
      double q = mul_mod( -LC_CONST, gs_mod( LC_MULT_INV, n, M ), M );
      double r = add_mod( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );

#endif // UINT64_C

      Bitmatrix<uint32_t> A( MATRIX_INV );
      A = A^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT_INV, n, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_back( uintmax_t e, uintmax_t c ) {   // jump back the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_back( 1 );

#ifdef UINT64_C // use the native 64-bit capability

      uint64_t p = mul_mod64( pow_mod64( LC_MULT_INV, e, c, M ), add_mod64( _s1, -LC_CONST, M ), M );
      uint64_t q = mul_mod64( -LC_CONST, gs_mod64( LC_MULT_INV, e, c, M ), M );
      uint64_t r = add_mod64( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod64( LC_CONST, r, M ) );

#else // native 64-bit not available, so use double instead

      double p = mul_mod( pow_mod( LC_MULT_INV, e, c, M ), add_mod( _s1, -LC_CONST, M ), M );
      double q = mul_mod( -LC_CONST, gs_mod( LC_MULT_INV, e, c, M ), M );
      double r = add_mod( p, q, M );
      _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );

#endif // UINT64_C

      Bitmatrix<uint32_t> A, B;

      if ( e ) {
         B = MATRIX_INV;
         for ( size_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX_INV;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT_INV, e, c, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_cycle( void ) { // jump ahead a full cycle of jkiss

      std::bitset<127> p( std::string( "11111111111110100010100001001000000000000000010111" ) +
                          std::string( "01011110110101000000000000000000000000000000100000" ) +
                          std::string( "000000000000000000000000000" ) );
      for ( size_t i = 0; i < p.size(); ++i ) if ( p.test(i) ) jump_ahead( i, 0 );
   }

   virtual uint32_t rng32( void ) { // returns the next random number (as a 32-bit unsigned int)

      _s1 = LC_MULT * _s1 + LC_CONST;

      _s2 ^= ( _s2 << 5 ), _s2 ^= ( _s2 >> 7 ), _s2 ^= ( _s2 << 22 );

      uint64_t a = MWC_MULT * _s3 + _s4;
      _s4 = ( a >> 32u );
      _s3 = uint32_t( a );

      return _s1 + _s2 + _s3;
   }

   virtual uint64_t rng64( void ) {   // returns 64-bit unsigned integer

      uint64_t low  = rng32();
      uint64_t high = rng32();
      return low | ( high << 32 );
   }

   virtual double rng32_01( void ) { // returns a double in the half-open interval [0,1)

      return double( rng32() ) * TWO32_INV;
   }

   virtual long double rng64_01( void ) {   // returns a long double in [0,1)

      return double( rng64() ) * TWO64_INV;
   }

private:

   uint32_t _s1, _s2, _s3, _s4;

}; // end jkiss class
} // end namespace JKISS

#endif // JKISS_H
```

Listing D-8. lfsr258.h

```
// lfsr258.h: L'Ecuyer's 64-bit Linear Feedback Shift Register RNG
// Period is (2^63 - 1)(2^55 - 1)(2^52 - 1)(2^47 - 1)(2^41 - 1) =
// 463168356949050750352076184268918090343706927944462529355293134289296410279935 or approximately 2^258
// Author: Pierre L'Ecuyer,
// Source: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tausme2.ps
// R. Saucier, December 2016

#ifndef LFSR258_H
#define LFSR258_H

namespace LFSR258 {

static const uint64_t N_SEEDS = 5;
static const bitmatrix64_t MATRIX[N_SEEDS] = {
   {
      {
         0x0000000000000000, 0x0000000000000800, 0x0000000000001000, 0x0000000000002000, 0x0000000000004000, 0x0000000000008000, 0x0000000000010000, 0x0000000000020000,
         0x0000000000040000, 0x0000000000080000, 0x0000000000100000, 0x0000000000200000, 0x0000000000400000, 0x0000000000800000, 0x0000000001000000, 0x0000000002000000,
         0x0000000004000000, 0x0000000008000000, 0x0000000010000000, 0x0000000020000000, 0x0000000040000000, 0x0000000080000000, 0x0000000100000000, 0x0000000200000000,
         0x0000000400000000, 0x0000000800000000, 0x0000001000000000, 0x0000002000000000, 0x0000004000000000, 0x0000008000000000, 0x0000010000000000, 0x0000020000000000,
         0x0000040000000000, 0x0000080000000000, 0x0000100000000000, 0x0000200000000000, 0x0000400000000000, 0x0000800000000000, 0x0001000000000000, 0x0002000000000000,
         0x0004000000000000, 0x0008000000000000, 0x0010000000000000, 0x0020000000000000, 0x0040000000000000, 0x0080000000000000, 0x0100000000000000, 0x0200000000000000,
         0x0400000000000000, 0x0800000000000000, 0x1000000000000000, 0x2000000000000000, 0x4000000000000001, 0x8000000000000003, 0x0000000000000006, 0x000000000000000c,
         0x0000000000000018, 0x0000000000000030, 0x0000000000000060, 0x00000000000000c0, 0x0000000000000180, 0x0000000000000300, 0x0000000000000600, 0x0000000000000400
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000004000, 0x0000000000008000, 0x0000000000010000, 0x0000000000020000, 0x0000000000040000, 0x0000000000080000, 0x0000000000100000,
         0x0000000000200000, 0x0000000000400000, 0x0000000000800000, 0x0000000001000000, 0x0000000002000000, 0x0000000004000000, 0x0000000008000000, 0x0000000010000000,
         0x0000000020000000, 0x0000000040000000, 0x0000000080000001, 0x0000000100000002, 0x0000000200000004, 0x0000000400000008, 0x0000000800000010, 0x0000001000000020,
         0x0000002000000040, 0x0000004000000080, 0x0000008000000100, 0x0000010000000200, 0x0000020000000400, 0x0000040000000800, 0x0000080000001000, 0x0000100000002000,
         0x0000200000000000, 0x0000400000000000, 0x0000800000000000, 0x0001000000000000, 0x0002000000000000, 0x0004000000000000, 0x0008000000000000, 0x0010000000000000,
         0x0020000000000000, 0x0040000000000000, 0x0080000000000001, 0x0100000000000002, 0x0200000000000004, 0x0400000000000008, 0x0800000000000010, 0x1000000000000020,
         0x2000000000000040, 0x4000000000000080, 0x8000000000000100, 0x0000000000000200, 0x0000000000000400, 0x0000000000000800, 0x0000000000001000, 0x0000000000002000
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000020000000000, 0x0000040000000000, 0x0000080000000000, 0x0000100000000000,
         0x0000200000000000, 0x0000400000000000, 0x0000800000000000, 0x0001000000000000, 0x0002000000000001, 0x0004000000000002, 0x0008000000000004, 0x0010000000000009,
         0x0020000000000012, 0x0040000000000024, 0x0080000000000048, 0x0100000000000090, 0x0200000000000120, 0x0400000000000240, 0x0800000000000480, 0x1000000000000900,
         0x2000000000001200, 0x4000000000002400, 0x8000000000004800, 0x0000000000009000, 0x0000000000012000, 0x0000000000024000, 0x0000000000048000, 0x0000000000090000,
         0x0000000000120000, 0x0000000000240000, 0x0000000000480000, 0x0000000000900000, 0x0000000001200000, 0x0000000002400000, 0x0000000004800000, 0x0000000009000000,
         0x0000000012000000, 0x0000000024000000, 0x0000000048000000, 0x0000000090000000, 0x0000000120000000, 0x0000000240000000, 0x0000000480000000, 0x0000000900000000,
         0x0000001200000000, 0x0000002400000000, 0x0000004800000000, 0x0000009000000000, 0x0000012000000000, 0x0000004000000000, 0x0000008000000000, 0x0000010000000000
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000010000000000, 0x0000020000000000, 0x0000040000000001, 0x0000080000000002, 0x0000100000000004, 0x0000200000000008, 0x0000400000000010,
         0x0000800000000021, 0x0001000000000042, 0x0002000000000084, 0x0004000000000108, 0x0008000000000210, 0x0010000000000420, 0x0020000000000840, 0x0040000000001080,
         0x0080000000002100, 0x0100000000004200, 0x0200000000008400, 0x0400000000010800, 0x0800000000021000, 0x1000000000042000, 0x2000000000084000, 0x4000000000108000,
         0x8000000000210000, 0x0000000000420000, 0x0000000000840000, 0x0000000001080000, 0x0000000002100000, 0x0000000004200000, 0x0000000008400000, 0x0000000010800000,
         0x0000000021000000, 0x0000000042000000, 0x0000000084000000, 0x0000000108000000, 0x0000000210000000, 0x0000000420000000, 0x0000000840000000, 0x0000001080000000,
         0x0000002100000000, 0x0000004200000000, 0x0000008400000000, 0x0000000800000000, 0x0000001000000000, 0x0000002000000000, 0x0000004000000000, 0x0000008000000000
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000080000000,
         0x0000000100000000, 0x0000000200000000, 0x0000000400000000, 0x0000000800000000, 0x0000001000000000, 0x0000002000000000, 0x0000004000000001, 0x0000008000000002,
         0x0000010000000004, 0x0000020000000009, 0x0000040000000012, 0x0000080000000024, 0x0000100000000048, 0x0000200000000090, 0x0000400000000120, 0x0000800000000240,
         0x0001000000000480, 0x0002000000000900, 0x0004000000001200, 0x0008000000002400, 0x0010000000004800, 0x0020000000009000, 0x0040000000012000, 0x0080000000024000,
         0x0100000000048000, 0x0200000000090000, 0x0400000000120000, 0x0800000000240000, 0x1000000000480000, 0x2000000000900000, 0x4000000001200000, 0x8000000002400000,
         0x0000000004800000, 0x0000000009000000, 0x0000000012000000, 0x0000000024000000, 0x0000000048000000, 0x0000000010000000, 0x0000000020000000, 0x0000000040000000
      }
   }
};
```

```
static const bitmatrix64_t MATRIX_INV[N_SEEDS] = {
   {
      {
         0x0000000000000001, 0xffc0000000000000, 0xff80000000000000, 0xff00000000000000, 0xfe00000000000000, 0xfc00000000000000, 0xf800000000000000, 0xf000000000000000,
         0xe000000000000000, 0xc000000000000000, 0x8000000000000001, 0x0000000000000002, 0x0000000000000004, 0x0000000000000008, 0x0000000000000010, 0x0000000000000020,
         0x0000000000000040, 0x0000000000000080, 0x0000000000000100, 0x0000000000000200, 0x0000000000000400, 0x0000000000000800, 0x0000000000001000, 0x0000000000002000,
         0x0000000000004000, 0x0000000000008000, 0x0000000000010000, 0x0000000000020000, 0x0000000000040000, 0x0000000000080000, 0x0000000000100000, 0x0000000000200000,
         0x0000000000400000, 0x0000000000800000, 0x0000000001000000, 0x0000000002000000, 0x0000000004000000, 0x0000000008000000, 0x0000000010000000, 0x0000000020000000,
         0x0000000040000000, 0x0000000080000000, 0x0000000100000000, 0x0000000200000000, 0x0000000400000000, 0x0000000800000000, 0x0000001000000000, 0x0000002000000000,
         0x0000004000000000, 0x0000008000000000, 0x0000010000000000, 0x0000020000000000, 0x0000040000000000, 0x0000080000000000, 0x0000100000000000, 0x0000200000000000,
         0x0000400000000000, 0x0000800000000000, 0x0001000000000000, 0x0002000000000000, 0x0004000000000000, 0x0008000000000000, 0x0010000000000001, 0xffe0000000000001
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0800000000000010, 0x1000000000000020, 0x2000000000000040, 0x4000000000000080, 0x8000000000000100, 0x0000000000000200, 0x0000000000000400,
         0x0000000000000800, 0x0000000000001000, 0x0000000000002000, 0x0000000000004000, 0x0000000000008000, 0x0000000000010000, 0x0000000000020000, 0x0000000000040000,
         0x0000000000080000, 0x0000000000100000, 0x0000000000200000, 0x0000000000400000, 0x0000000000800000, 0x0000000001000000, 0x0000000002000000, 0x0000000004000000,
         0x0000000008000000, 0x0000000010000000, 0x0000000020000000, 0x0000000040000000, 0x0000000080000001, 0x0000000100000002, 0x0000000200000004, 0x0000000400000008,
         0x0800000800000000, 0x1000001000000000, 0x2000002000000000, 0x4000004000000000, 0x8000008000000000, 0x0000010000000000, 0x0000020000000000, 0x0000040000000000,
         0x0000080000000000, 0x0000100000000000, 0x0000200000000000, 0x0000400000000000, 0x0000800000000000, 0x0001000000000000, 0x0002000000000000, 0x0004000000000000,
         0x0008000000000000, 0x0010000000000000, 0x0020000000000000, 0x0040000000000000, 0x0080000000000001, 0x0100000000000002, 0x0200000000000004, 0x0400000000000008
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x4924924800000000, 0x9249249000000000, 0x2492492000000000, 0x4924924000000000,
         0x9249248000000000, 0x2492490000000000, 0x4924920000000000, 0x9249240000000000, 0x2492480000000000, 0x4924900000000000, 0x9249200000000000, 0x2492400000000000,
         0x4924800000000000, 0x9249000000000000, 0x2492000000000000, 0x4924000000000000, 0x9248000000000000, 0x2490000000000001, 0x4920000000000002, 0x9240000000000004,
         0x2480000000000008, 0x4900000000000010, 0x9200000000000020, 0x2400000000000040, 0x4800000000000080, 0x9000000000000100, 0x2000000000000200, 0x4000000000000400,
         0x8000000000000800, 0x0000000000001000, 0x0000000000002000, 0x0000000000004000, 0x0000000000008000, 0x0000000000010000, 0x0000000000020000, 0x0000000000040000,
         0x0000000000080000, 0x0000000000100000, 0x0000000000200000, 0x0000000000400000, 0x0000000000800000, 0x0000000001000000, 0x0000000002000000, 0x0000000004000000,
         0x0000000008000000, 0x0000000010000000, 0x0000000020000000, 0x0000000040000000, 0x0000000080000000, 0x4924924900000000, 0x9249249200000000, 0x2492492400000000
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x2108420000000000, 0x4210840000000000, 0x8421080000000000, 0x0842100000000000, 0x1084200000000000, 0x2108400000000000, 0x4210800000000001,
         0x8421000000000002, 0x0842000000000004, 0x1084000000000008, 0x2108000000000010, 0x4210000000000020, 0x8420000000000040, 0x0840000000000080, 0x1080000000000100,
         0x2100000000000200, 0x4200000000000400, 0x8400000000000800, 0x0800000000001000, 0x1000000000002000, 0x2000000000004000, 0x4000000000008000, 0x8000000000010000,
         0x0000000000020000, 0x0000000000040000, 0x0000000000080000, 0x0000000000100000, 0x0000000000200000, 0x0000000000400000, 0x0000000000800000, 0x0000000001000000,
         0x0000000002000000, 0x0000000004000000, 0x0000000008000000, 0x0000000010000000, 0x0000000020000000, 0x0000000040000000, 0x0000000080000000, 0x0000000100000000,
         0x0000000200000000, 0x0000000400000000, 0x0000000800000000, 0x2108421000000000, 0x4210842000000000, 0x8421084000000000, 0x0842108000000000, 0x1084210000000000
      }
   },
   {
      {
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x4900000000008000,
         0x9200000000010000, 0x2400000000020000, 0x4800000000040000, 0x9000000000080000, 0x2000000000100000, 0x4000000000200000, 0x8000000000400000, 0x0000000000800000,
         0x0000000001000000, 0x0000000002000000, 0x0000000004000000, 0x0000000008000000, 0x0000000010000000, 0x0000000020000000, 0x0000000040000000, 0x0000000080000000,
         0x0000000100000000, 0x0000000200000000, 0x0000000400000000, 0x0000000800000000, 0x0000001000000000, 0x0000002000000000, 0x0000004000000001, 0x0000008000000002,
         0x0000010000000004, 0x0000020000000009, 0x0000040000000012, 0x0000080000000024, 0x0000100000000048, 0x0000200000000090, 0x0000400000000120, 0x0000800000000240,
         0x0001000000000480, 0x0002000000000900, 0x0004000000001200, 0x0008000000002400, 0x0010000000004800, 0x4920000000001000, 0x9240000000002000, 0x2480000000004000
      }
   }
};
static const uint64_t C0 = 0xffffffffffffffffull;   // 18446744073709551615ull
static const uint64_t C1 = 0xfffffffffffffffeull;   // 18446744073709551614ull
static const uint64_t C2 = 0xfffffffffffffe00ull;   // 18446744073709551104ull
static const uint64_t C3 = 0xfffffffffffff000ull;   // 18446744073709547520ull
static const uint64_t C4 = 0xfffffffffffe0000ull;   // 18446744073709420544ull
static const uint64_t C5 = 0xffffffffff800000ull;   // 18446744073701163008ull

class lfsr258 : public Generator<uint64_t> {

public:
   lfsr258( void ) { // default constructor
   }

   lfsr258( std::vector<uint64_t> seed ) { // constructor from vector seed

```

```
      setState( seed );
   }

   virtual ~lfsr258() { // default destructor

      std::cout << "deleting lfsr258" << std::endl;
   }

   virtual void setState( std::vector<uint64_t> seed ) { // set the seeds

      assert( seed.size() >= N_SEEDS );

      // VERY IMPORTANT: The initial seeds _s1, _s2, _s3 MUST be larger than 1, 511, 4095, 131071, and 8388607 respectively
      _s[0] = seed[0]; if ( _s[0] <       2 ) _s[0] +=       2;
      _s[1] = seed[1]; if ( _s[1] <     512 ) _s[1] +=     512;
      _s[2] = seed[2]; if ( _s[2] <    4096 ) _s[2] +=    4096;
      _s[3] = seed[3]; if ( _s[3] <  131072 ) _s[3] +=  131072;
      _s[4] = seed[4]; if ( _s[4] < 8388608 ) _s[4] += 8388608;
   }

   virtual void getState( std::vector<uint64_t>& seed ) { // get the seed vector

      assert( seed.size() >= N_SEEDS );
      for ( size_t i = 0; i < N_SEEDS; i++ ) seed[i] = _s[i];
   }

   virtual void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         Bitmatrix<uint64_t> A( MATRIX[i] );
         _s[i] = ( A^n ) * _s[i];
      }
   }

   virtual void jump_ahead( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_ahead( 1 );

      Bitmatrix<uint64_t> A, B;

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         if ( e ) {
            B = MATRIX[i];
            for ( uintmax_t j = 0; j < e; j++ ) B *= B;
         }
         A = MATRIX[i];
         A = A^c;
         if ( e ) A *= B;
         _s[i] = A * _s[i];
      }
   }

   virtual void jump_back( uintmax_t n ) { // jump ahead the next n random numbers

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         Bitmatrix<uint64_t> A( MATRIX_INV[i] );
         _s[i] = ( A^n ) * _s[i];
      }
   }

   virtual void jump_back( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_back( 1 );

      Bitmatrix<uint64_t> A, B;

      for ( size_t i = 0; i < N_SEEDS; i++ ) {

         if ( e ) {
            B = MATRIX_INV[i];
            for ( uintmax_t j = 0; j < e; j++ ) B *= B;
         }
         A = MATRIX_INV[i];
         A = A^c;
         if ( e ) A *= B;
         _s[i] = A * _s[i];
      }
   }

   virtual void jump_cycle( void ) { // jump ahead an entire cycle of lfsr258

      const uint32_t A = 63, B = 55, C = 52, D = 47, E = 41;
      jump_ahead( A + B + C + D + E, 0 );
      jump_back( B + C + D + E, 0 );
      jump_back( A + C + D + E, 0 );
      jump_back( A + B + D + E, 0 );
      jump_back( A + B + C + E, 0 );
      jump_back( A + B + C + D, 0 );
      jump_ahead( C + D + E, 0 );
      jump_ahead( B + D + E, 0 );
      jump_ahead( B + C + E, 0 );
      jump_ahead( B + C + D, 0 );
      jump_ahead( A + D + E, 0 );
      jump_ahead( A + C + E, 0 );
      jump_ahead( A + C + D, 0 );
      jump_ahead( A + B + E, 0 );
      jump_ahead( A + B + D, 0 );
      jump_ahead( A + B + C, 0 );
      jump_back( A + B, 0 );
      jump_back( A + C, 0 );
      jump_back( A + D, 0 );
      jump_back( A + E, 0 );
      jump_back( B + C, 0 );
      jump_back( B + D, 0 );
      jump_back( B + E, 0 );
      jump_back( C + D, 0 );
      jump_back( C + E, 0 );
      jump_back( D + E, 0 );
      jump_ahead( A, 0 );
      jump_ahead( B, 0 );
      jump_ahead( C, 0 );
      jump_ahead( D, 0 );
      jump_ahead( E, 0 );
      jump_back( 1 );
   }

   uint32_t rng32( void ) { // returns the next random number as a 32-bit integer

      return uint32_t( rng64() );
   }

   uint64_t rng64( void ) { // returns the next random number as a 64-bit integer

      _s[0] = ( ( _s[0] & C1 ) << 10 ) ^ ( ( ( _s[0] <<  1 ) ^ _s[0] ) >> 53 );
      _s[1] = ( ( _s[1] & C2 ) <<  5 ) ^ ( ( ( _s[1] << 24 ) ^ _s[1] ) >> 50 );
      _s[2] = ( ( _s[2] & C3 ) << 29 ) ^ ( ( ( _s[2] <<  3 ) ^ _s[2] ) >> 23 );
      _s[3] = ( ( _s[3] & C4 ) << 23 ) ^ ( ( ( _s[3] <<  5 ) ^ _s[3] ) >> 24 );
      _s[4] = ( ( _s[4] & C5 ) <<  8 ) ^ ( ( ( _s[4] <<  3 ) ^ _s[4] ) >> 33 );

      return ( _s[0] ^ _s[1] ^ _s[2] ^ _s[3] ^ _s[4] ) & C0;
   }

   double rng32_01( void ) { // returns a random number in the half-open interval [0,1)

      return rng32() * TWO32_INV;
   }

   long double rng64_01( void ) { // returns a random number in the half-open interval [0,1)

      return rng64() * TWO64_INV;
   }

private:

   uint64_t _s[ N_SEEDS ];

}; // end lfsr258 class
} // end namespace LFSR258

#endif // LFSR258_H
```

Listing D-9. jlkiss.h

```
// jlkiss.h: Based upon Marsaglia's Keep It Simple Stupid RNG
// Ref: Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications
// David Jones, UCL Bioinformatics Group (d.jones@cs.ucl.ac.uk), May 7, 2010
// Cycle length is (2^64)(2^64-1)(4294584393(2^31)-1) = 3138271061012620924047441856806230331094853687768430673920,
// or approximately 2^191
// Period of MWC is 4294584393(2^31)-1 = 9222549758923505663
// R. Saucier, December 2016

#ifndef JLKISS_H
#define JLKISS_H

namespace JLKISS {

   static const bitmatrix64_t MATRIX = {
      {
         0x0008000440200011, 0x0010000880400022, 0x0020001100800044, 0x0040002201000088, 0x0080004402000110, 0x0100008804000220, 0x0200011008000440, 0x0400022010000880,
         0x0800044020001100, 0x1000088040002200, 0x2000110080004400, 0x4000220100008800, 0x8000440200011000, 0x0000880400022000, 0x0001100800044000, 0x0002201000088000,
         0x0004402000110000, 0x0008804040220001, 0x0011008080440002, 0x0022010100880004, 0x0044020201100008, 0x0088040402200010, 0x0110080804400020, 0x0220101008800040,
         0x0440202011000080, 0x0880404022000100, 0x1100808044000200, 0x2201010088000400, 0x4402020110000800, 0x8804040220001000, 0x1008080440002000, 0x2010100880004000,
         0x4020201100008000, 0x8040402200010000, 0x0080804400020000, 0x0101008800040000, 0x0202011000080000, 0x0404022000100000, 0x0808044000200000, 0x1010088000400000,
         0x2020110000800000, 0x4040220001000000, 0x8080440002000000, 0x0100080004000000, 0x0200100008000000, 0x0400200010000000, 0x0800400020000000, 0x1000800040000000,
         0x2001000080000000, 0x4002000100000000, 0x8004000200000000, 0x0008000400000000, 0x0010000800000000, 0x0020001000000000, 0x0040002000000000, 0x0080004000000000,
         0x0100008000000000, 0x0200010000000000, 0x0400020000000000, 0x0800040000000000, 0x1000080000000000, 0x2000100000000000, 0x4000200000000000, 0x8000400000000000
      }
   };
   static const bitmatrix64_t MATRIX_INV = {
      {
         0x90808c0404202201, 0x2101180808404402, 0x4202301010808804, 0x8404602021011008, 0x8880444402220011, 0x1100888804440022, 0x2201111008880044, 0x4402222011100088,
         0x8804444022200110, 0x1008888044400220, 0x2011110088800440, 0x4022220111000880, 0x8044440222001100, 0x0088880444002200, 0x0111100888004400, 0x0222201110008800,
         0x0444402220011000, 0x8888844440222001, 0x1111088880444002, 0x2222111100888004, 0x4444222201110008, 0x0888404402020011, 0x1110808804040022, 0x2221011008080044,
         0x4442022010100088, 0x8884044020200110, 0x1108088040400220, 0x2210110080800440, 0x4420220101000880, 0x8840440202001100, 0x1080880404002200, 0x2101100808004400,
         0x4202201010008800, 0x8404402020011000, 0x8880044400220001, 0x1100088800440002, 0x2200111000880004, 0x4400222001100008, 0x8800444002200010, 0x1000888004400020,
         0x2001110008800040, 0x4002220011000080, 0x8004440022000100, 0x0008880044000200, 0x0011100088000400, 0x0022200110000800, 0x0044400220001000, 0x0088800440002000,
         0x0111000880004000, 0x0222001100008000, 0x0444002200010000, 0x8888044400220001, 0x1110088800440002, 0x2220111000880004, 0x4440222001100008, 0x8880444002200010,
         0x1100888004400020, 0x2201110008800040, 0x4402220011000080, 0x8804440022000100, 0x1008880044000200, 0x2011100088000400, 0x4022200110000800, 0x8044400220001000
      }
   };
   static const uint64_t LC_MULT      = 0x14ada13ed78492adull;   // 1490024343005336237ull;
   static const uint64_t LC_CONST     = 0x00000000075bcd15ull;   // 123456789ull;
   static const uint64_t LC_MULT_INV  = 0xc5a2d1aa2af8a125ull;   // 14241175500494512421ull;
   static const uint64_t MWC_MULT     = 0x00000000fffa2849ull;   // 4294584393ull;
   static const uint64_t MWC_MOD      = 0xfffa2848ffffffffull;   // 18445099517847011327ull;
   static const uint64_t MWC_MULT_INV = 0x0000000100000000ull;   // 4294967296ull;
   static const uint64_t SR_PERIOD    = 0xffffffffffffffffull;   // 18446744073709551615ull;
   static const uint64_t MWC_PERIOD   = 0x7ffd14247fffffffull;   // 9222549758923505663ull;
   static const uint32_t N_SEEDS      = 3;                       // requires three 64-bit words

class jlkiss : public Generator<uint64_t> {

public:
   jlkiss( void ) { // default constructor
   }

   jlkiss( std::vector<uint64_t> seed ) { // constructor from seed vector

      setState( seed );
   }

   virtual ~jlkiss() { // default destructor

      std::cout << "deleting jlkiss" << std::endl;
   }

   virtual void setState( std::vector<uint64_t> seed ) { // set the seeds

      assert( seed.size() >= N_SEEDS );
      _s1 = seed[0];
      _s2 = seed[1];
      _s3 = uint32_t( seed[2] >> 32 );
      _s4 = uint32_t( seed[2] );
   }

   virtual void getState( std::vector<uint64_t>& seed ) { // get the seed vector

      assert( seed.size() >= N_SEEDS );
      seed[0] = _s1;
      seed[1] = _s2;
      seed[2] = ( uint64_t( _s3 ) << 32 ) + _s4;
   }

   void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers

      _s1 = mul64( pow64( LC_MULT, n ), _s1 ) + mul64( LC_CONST, gs64( LC_MULT, n ) );

      Bitmatrix<uint64_t> A, B( MATRIX );
      A = B^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT, n, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   void jump_ahead( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_ahead( 1 );

      _s1 = mul64( pow64( LC_MULT, e, c ), _s1 ) + mul64( LC_CONST, gs64( LC_MULT, e, c ) );

      Bitmatrix<uint64_t> A, B;

      if ( e ) {
         B = MATRIX;
         for ( uint64_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT, e, c, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   void jump_back( uintmax_t n ) { // jump back the next n random numbers

      _s1 = mul64( pow64( LC_MULT_INV, n ), _s1 - LC_CONST ) + LC_CONST - mul64( LC_CONST, gs64( LC_MULT_INV, n ) );

      Bitmatrix<uint64_t> A, B( MATRIX_INV );
      A = B^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT_INV, n, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   void jump_back( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_back( 1 );

      _s1 = mul64( pow64( LC_MULT_INV, e, c ), _s1 - LC_CONST ) + LC_CONST - mul64( LC_CONST, gs64( LC_MULT_INV, e, c ) );

      Bitmatrix<uint64_t> A, B;

      if ( e ) {
         B = MATRIX_INV;
         for ( uint64_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX_INV;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT_INV, e, c, MWC_MOD ), a, MWC_MOD );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );
   }

   virtual void jump_cycle( void ) { // jump ahead a full cycle of jlkiss

      std::bitset<191> p( std::string( "11111111111110100010100001001000111111111111111111" ) +
                          std::string( "11111111111110100000000000010111010111101101110000" ) +
                          std::string( "00000000000000000000000000100000000000000000000000" ) +
                          std::string( "00000000000000000000000000000000000000000" ) );
      for ( size_t i = 0; i < p.size(); ++i ) if ( p.test(i) ) jump_ahead( i, 0 );
   }

   uint32_t rng32( void ) {

      _s1 = LC_MULT * _s1 + LC_CONST;

      _s2 ^= ( _s2 << 21 ), _s2 ^= ( _s2 >> 17 ), _s2 ^= ( _s2 << 30 );

      uint64_t a = MWC_MULT * _s3 + _s4;
      _s4 = ( a >> 32u );
      _s3 = uint32_t( a );

      //return (uint32_t)_s1 + (uint32_t)_s2 + _s3;   // this didn't work
      //return (uint32_t)( _s1 >> 32 ) + (uint32_t)_s2 + _s3;   // this doesn't seem to work
      return _s1 + _s2 + _s3;   // this passes all the tests, small crush, crush, and big crush
   }

   uint64_t rng64( void ) {

      _s1 = LC_MULT * _s1 + LC_CONST;

      _s2 ^= ( _s2 << 21 ), _s2 ^= ( _s2 >> 17 ), _s2 ^= ( _s2 << 30 );

      uint64_t a = MWC_MULT * _s3 + _s4;
      _s4 = ( a >> 32u );
      _s3 = uint32_t( a );

      return _s1 + _s2 + ( (uint64_t)_s4 << 32 ) + _s3;
   }

   double rng32_01( void ) { // returns a random number in the half-open interval [0,1)

      return double( rng32() ) * TWO32_INV;
   }

   long double rng64_01( void ) { // returns a random number in the half-open interval [0,1)

      return ( long double )( rng64() ) * TWO64_INV;
   }

private:

   uint64_t _s1, _s2;
   uint32_t _s3, _s4;

}; // end jlkiss class
} // end namespace JLKISS

#endif // JLKISS_H
```

Listing D-10. jlkiss64.h

```
// jlkiss64.h: Based upon Marsaglia's Keep It Simple Stupid RNG
// Ref: Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications
// David Jones, UCL Bioinformatics Group (d.jones@cs.ucl.ac.uk), May 7, 2010
// Period is (2^64)(2^64-1)(4294584393(2^31)-1)(698769069(2^31)-1) = 4709274331675767436556996170486797220343483431369386641683085078522096517120
// or approximately 2^251
// Requires four 64-bit seeds to initialize.
// R. Saucier, December 2016

#ifndef JLKISS64_H
#define JLKISS64_H

#include <iostream>
#include <iomanip>
using namespace std;
```

```cpp

namespace JLKISS64 {

   static const bitmatrix64_t MATRIX = {
      {
         0x0008000440200011, 0x0010000880400022, 0x0020001100800044, 0x0040002201000088, 0x0080004402000110, 0x0100008804000220, 0x0200011008000440, 0x0400022010000880,
         0x0800044020001100, 0x1000088040002200, 0x2000110080004400, 0x4000220100008800, 0x8000440200011000, 0x0000880400022000, 0x0001100800044000, 0x0002201000088000,
         0x0004402000110000, 0x0008804040220001, 0x0011008080440002, 0x0022010100880004, 0x0044020201100008, 0x0088040402200010, 0x0110080804400020, 0x0220101008800040,
         0x0440202011000080, 0x0880404022000100, 0x1100808044000200, 0x2201010088000400, 0x4402020110000800, 0x8804040220001000, 0x1008080440002000, 0x2010100880004000,
         0x4020201100008000, 0x8040402200010000, 0x0080804400020000, 0x0101008800040000, 0x0202011000080000, 0x0404022000100000, 0x0808044000200000, 0x1010088000400000,
         0x2020110000800000, 0x4040220001000000, 0x8080440002000000, 0x0100080004000000, 0x0200100008000000, 0x0400200010000000, 0x0800400020000000, 0x1000800040000000,
         0x2001000080000000, 0x4002000100000000, 0x8004000200000000, 0x0008000400000000, 0x0010000800000000, 0x0020001000000000, 0x0040002000000000, 0x0080004000000000,
         0x0100008000000000, 0x0200010000000000, 0x0400020000000000, 0x0800040000000000, 0x1000080000000000, 0x2000100000000000, 0x4000200000000000, 0x8000400000000000
      }
   };
   static const bitmatrix64_t MATRIX_INV = {
      {
         0x90808c0404202201, 0x2101180808404402, 0x4202301010808804, 0x8404602021011008, 0x8880444402220011, 0x1100888804440022, 0x2201111008880044, 0x4402222011100088,
         0x8804444022200110, 0x1008888044400220, 0x2011110088800440, 0x4022220111000880, 0x8044440222001100, 0x0088880444002200, 0x0111100888004400, 0x0222201110008800,
         0x0444402220011000, 0x8888844440222001, 0x1111088880444002, 0x2222111100888004, 0x4444222201110008, 0x0888404402020011, 0x1110808804040022, 0x2221011008080044,
         0x4442022010100088, 0x8884044020200110, 0x1108088040400220, 0x2210110080800440, 0x4420220101000880, 0x8840440202001100, 0x1080880404002200, 0x2101100808004400,
         0x4202201010008800, 0x8404402020011000, 0x8880044400220001, 0x1100088800440002, 0x2200111000880004, 0x4400222001100008, 0x8800444002200010, 0x1000888004400020,
         0x2001110008800040, 0x4002220011000080, 0x8004440022000100, 0x0008880044000200, 0x0011100088000400, 0x0022200110000800, 0x0044400220001000, 0x0088800440002000,
         0x0111000880004000, 0x0222001100008000, 0x0444002200010000, 0x8888044400220001, 0x1110088800440002, 0x2220111000880004, 0x4440222001100008, 0x8880444002200010,
         0x1100888004400020, 0x2201110008800040, 0x4402220011000080, 0x8804440022000100, 0x1008880044000200, 0x2011100088000400, 0x4022200110000800, 0x8044400220001000
      }
   };
   static const uint64_t LC_MULT        = 0x14ada13ed78492ad;   // 1490024343005336237ULL;
   static const uint64_t LC_CONST       = 0x00000000075bcd15;   // 123456789ULL;
   static const uint64_t LC_MULT_INV    = 0xc5a2d1aa2af8a125;   // 14241175500494512421ULL;
   static const uint64_t MWC_MULT1      = 0x00000000fffa2849;   // 4294584393ULL;
   static const uint64_t MWC_MOD1       = 0xfffa2848ffffffff;   // 18445099517847011327ULL;
   static const uint64_t MWC_MULT1_INV  = 0x0000000100000000;   // 4294967296ULL;
   static const uint64_t MWC_MULT2      = 0x0000000029a65ead;   // 698769069ULL;
   static const uint64_t MWC_MOD2       = 0x29a65eacffffffff;   // 3001190298811367423ULL;
   static const uint64_t MWC_MULT2_INV  = 0x0000000100000000;   // 4294967296ULL;
   static const uint32_t N_SEEDS        = 4;                    // requires 4 64-bit words

class jlkiss64 : public Generator<uint64_t> {

public:
   jlkiss64( void ) { // default constructor
   }

   jlkiss64( std::vector<uint64_t> seed ) { // constructor from seed vector

      setState( seed );
   }

   virtual ~jlkiss64() { // default destructor

      std::cout << "deleting jlkiss64" << std::endl;
   }

   virtual void setState( std::vector<uint64_t> seed ) { // set the seeds from four 64-bit words

      assert( seed.size() >= N_SEEDS );
      _s1 = seed[0];
      _s2 = seed[1];
      _s3 = uint32_t( seed[2] >> 32 );
      _s4 = uint32_t( seed[2] );
      _s5 = uint32_t( seed[3] >> 32 );
      _s6 = uint32_t( seed[3] );
   }

   virtual void getState( std::vector<uint64_t>& seed ) { // get the seed vector

      assert( seed.size() >= N_SEEDS );
      seed[0] = _s1;
      seed[1] = _s2;
      seed[2] = ( uint64_t( _s3 ) << 32 ) + _s4;
      seed[3] = ( uint64_t( _s5 ) << 32 ) + _s6;
   }

   void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers

      _s1 = mul64( pow64( LC_MULT, n ), _s1 ) + mul64( LC_CONST, gs64( LC_MULT, n ) );
      Bitmatrix<uint64_t> A, B( MATRIX );
      A = B^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT1, n, MWC_MOD1 ), a, MWC_MOD1 );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );

      a = _s5 + ( (uint64_t)_s6 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT2, n, MWC_MOD2 ), a, MWC_MOD2 );
      _s6 = ( uint32_t )( a >> 32u );
      _s5 = ( uint32_t )( a );
   }

   void jump_ahead( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_ahead( 1 );

      _s1 = mul64( pow64( LC_MULT, e, c ), _s1 ) + mul64( LC_CONST, gs64( LC_MULT, e, c ) );

      Bitmatrix<uint64_t> A, B;

      if ( e ) {
         B = MATRIX;
         for ( uint64_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT1, e, c, MWC_MOD1 ), a, MWC_MOD1 );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );

      a = _s5 + ( (uint64_t)_s6 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT2, e, c, MWC_MOD2 ), a, MWC_MOD2 );
      _s6 = ( uint32_t )( a >> 32u );
      _s5 = ( uint32_t )( a );
   }

   void jump_back( uintmax_t n ) { // jump back the next n random numbers

      _s1 = mul64( pow64( LC_MULT_INV, n ), _s1 - LC_CONST ) + LC_CONST - mul64( LC_CONST, gs64( LC_MULT_INV, n ) );

      Bitmatrix<uint64_t> A, B( MATRIX_INV );
      A = B^n;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT1_INV, n, MWC_MOD1 ), a, MWC_MOD1 );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );

      a = _s5 + ( (uint64_t)_s6 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT2_INV, n, MWC_MOD2 ), a, MWC_MOD2 );
      _s6 = ( uint32_t )( a >> 32u );
      _s5 = ( uint32_t )( a );
   }

   void jump_back( uintmax_t e, uintmax_t c ) {   // jump ahead the next n random numbers, where n = 2^e + c

      if ( e == 0 && c == 0 ) return jump_back( 1 );

      _s1 = mul64( pow64( LC_MULT_INV, e, c ), _s1 - LC_CONST ) + LC_CONST - mul64( LC_CONST, gs64( LC_MULT_INV, e, c ) );

      Bitmatrix<uint64_t> A, B;

      if ( e ) {
         B = MATRIX_INV;
         for ( uint64_t i = 0; i < e; i++ ) B *= B;
      }
      A = MATRIX_INV;
      A = A^c;
      if ( e ) A *= B;
      _s2 = A * _s2;

      uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT1_INV, e, c, MWC_MOD1 ), a, MWC_MOD1 );
      _s4 = ( uint32_t )( a >> 32u );
      _s3 = ( uint32_t )( a );

      a = _s5 + ( (uint64_t)_s6 << 32u );
      a = mul_mod64( pow_mod64( MWC_MULT2_INV, e, c, MWC_MOD2 ), a, MWC_MOD2 );
      _s6 = ( uint32_t )( a >> 32u );
      _s5 = ( uint32_t )( a );
   }

   virtual void jump_cycle( void ) { // jump ahead a full cycle of jlkiss64

      std::bitset<252> p( std::string( "10100110100101011010110101010010100111001011100000" ) +
                          std::string( "01110101001010000011000110011000011010111111010110" ) +
                          std::string( "00110100011111100010110001010100110100000100001101" ) +
                          std::string( "11101011111111111111111111111111111111000000000000" ) +
                          std::string( "00000000000000000000000000000000000000000000000000" ) +
                          std::string( "00" ) );
      for ( size_t i = 0; i < p.size(); ++i ) if ( p.test(i) ) jump_ahead( i, 0 );
   }

   uint32_t rng32( void ) { // returns the next random number (as a 32-bit unsigned int)

      return uint32_t( rng64() );
   }

   uint64_t rng64( void ) { // returns the next random number (as a 64-bit unsigned integer)

      _s1 = LC_MULT * _s1 + LC_CONST;

      _s2 ^= ( _s2 << 21 ), _s2 ^= ( _s2 >> 17 ), _s2 ^= ( _s2 << 30 );

      uint64_t a = MWC_MULT1 * _s3 + _s4;
      _s4 = ( a >> 32u );    // upper 32 bits of a
      _s3 = uint32_t( a );   // lower 32 bits of a

      a = MWC_MULT2 * _s5 + _s6;
      _s6 = ( a >> 32u );    // upper 32 bits of a
      _s5 = uint32_t( a );   // lower 32 bits of a

      // use _s3 for lower 32 bits and _s5 << 32 for upper 32 bits of 64-bit word
      return _s1 + _s2 + _s3 + ( ( uint64_t )_s5 << 32 );
   }

   double rng32_01( void ) { // returns a random number in the half-open interval [0,1)

      return double( rng32() ) * TWO32_INV;
   }

   long double rng64_01( void ) { // returns a random number in the half-open interval [0,1)

      return ( long double )( rng64() ) * TWO64_INV;
   }

 private:

   uint64_t _s1, _s2;             // two 64-bit
   uint32_t _s3, _s4, _s5, _s6;   // important that these be 32-bit and not 64-bit

}; // end jlkiss64 class
} // end namespace JLKISS64

#endif // JLKISS64_H
```

Listing D-11. Generator.h

```cpp
// Generator.h: template class file for random number generators
// R. Saucier, July 2016

#ifndef GENERATOR_H
#define GENERATOR_H

#include "Bitmatrix.h"
#include "mod_math.h"
#include <vector>
#include <bitset>
#include <iostream>

template <class T>   // for 32-bit and 64-bit generators
class Generator {

public:

   virtual ~Generator() {};// std::cout << "deleting Generator" << std::endl; }
   virtual void setState( std::vector<T> seed ) = 0;
   virtual void getState( std::vector<T>& seed ) = 0;
   virtual void jump_ahead( uintmax_t ) = 0;
   virtual void jump_ahead( uintmax_t,  uintmax_t ) = 0;
   virtual void jump_back( uintmax_t ) = 0;
   virtual void jump_back( uintmax_t,  uintmax_t ) = 0;
   virtual void jump_cycle( void ) = 0;

   virtual uint32_t    rng32( void ) = 0;      // returns 32-bit integer
   virtual uint64_t    rng64( void ) = 0;      // returns 64-bit integer
   virtual double      rng32_01( void ) = 0;   // returns double in [0,1)
   virtual long double rng64_01( void ) = 0;   // returns long double in [0,1)

   inline double u32( double a = 0., double b = 1. ) { return a + ( b - a ) * this->rng32_01(); }
   inline double u64( double a = 0., double b = 1. ) { return a + ( b - a ) * this->rng64_01(); }
};

// 32-bit generators
#include "kiss.h"
#include "jkiss.h"
#include "lfsr88.h"
#include "lfsr113.h"

// 64-bit generators
#include "jlkiss.h"
#include "jlkiss64.h"
#include "lfsr258.h"

#endif
```

Listing D-12. Random.h

```cpp
// Random.h: Definition and Implementation of Random Number Distribution Class
// This rewrite of the following reference decouples the distributions from the generators
// Ref: Richard Saucier, "Computer Generation of Statistical Distributions," ARL-TR-2168,
//      US Army Research Laboratory, Aberdeen Proving Ground, MD, 21005-5068, March 2000.
// R. Saucier, December 2016

#ifndef RANDOM_H
#define RANDOM_H

#include "Generator.h"
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <functional>
#include <cassert>
#include <cmath>
#include <climits>
#include <cfloat>     // for FLT_MIN and FLT_MAX
#include <unistd.h>   // for getpid
#include <map>

namespace rnd {   // rnd namespace

// for convenience, define some data structures for bivariate distributions

typedef std::pair<double,double> cartesianCoord;   // x     = first, y     = second
typedef std::pair<double,double> sphericalCoord;   // theta = first, phi   = second
typedef std::pair<double,double> polarCoord;       // r     = first, theta = second

struct point2d {   // cartesian coordinates in 2-D for use in stochasticInterpolation

   double x, y;
   point2d& operator+=( const point2d& p ) {
      x += p.x;
      y += p.y;
      return *this;
   }
   point2d& operator-=( const point2d& p ) {
      x -= p.x;
      y -= p.y;
      return *this;
   }
   point2d& operator*=( double scalar ) {
      x *= scalar;
      y *= scalar;
      return *this;
   }
   point2d& operator/=( double scalar ) {
      x /= scalar;
      y /= scalar;
      return *this;
   }
```

```
};

// comparison functor for determining the neighborhood of a data point
struct dSquared : public std::binary_function< point2d, point2d, bool > {

   bool operator()( point2d p, point2d q ) { return p.x * p.x + p.y * p.y < q.x * q.x + q.y * q.y; }
};

template <class Typename>   // for 32-bit and 64-bit generators
class Random {

public:

   Random( Generator<Typename> *gen ) { _gen = gen; }
  ~Random( void ) {}  // default destructor

// Continuous Distributions

   double arcsine( double xMin = 0., double xMax = 1. ) { // Arc Sine

      double q = sin( M_PI_2 * _u() );
      return xMin + ( xMax - xMin ) * q * q;
   }

   double beta( double v, double w,                  // Beta
                double xMin = 0., double xMax = 1. ) { // (v > 0. and w > 0.)

      if ( v < w ) return xMax - ( xMax - xMin ) * beta( w, v );
      double y1 = gamma( 0., 1., v );
      double y2 = gamma( 0., 1., w );
      return xMin + ( xMax - xMin ) * y1 / ( y1 + y2 );
   }

   double cauchy( double a = 0., double b = 1. ) { // Cauchy (or Lorentz)

      // a is the location parameter and b is the scale parameter
      // b is the half width at half maximum (HWHM) and variance doesn’t exist

      assert( b > 0. );

      return a + b * tan( M_PI * uniform( -0.5, 0.5 ) );
   }

   double chiSquare( int df ) { // Chi-Square

      assert( df >= 1 );

      return gamma( 0., 2., 0.5 * double( df ) );
   }

   double cosine( double xMin = 0., double xMax = 1. ) { // Cosine

      assert( xMin < xMax );

      double a = 0.5 * ( xMin + xMax );    // location parameter
      double b = ( xMax - xMin ) / M_PI;   // scale parameter

      return a + b * asin( uniform( -1., 1. ) );
   }

   double doubleLog( double xMin = -1., double xMax = 1. ) { // Double Log

      assert( xMin < xMax );

      double a = 0.5 * ( xMin + xMax );    // location parameter
      double b = 0.5 * ( xMax - xMin );    // scale parameter

      if ( bernoulli( 0.5 ) ) return a + b * _u() * _u();
      else                    return a - b * _u() * _u();
   }

   double erlang( double b, int c ) { // Erlang (b > 0. and c >= 1)

      assert( b > 0. && c >= 1 );

      double prod = 1.;
      for ( int i = 0; i < c; i++ ) prod *= _u();

      return -b * log( prod );
   }

   double exponential( double a = 0., double c = 1. ) { // Exponential
                                                    // location a, shape c
      assert( c > 0.0 );

      return a - c * log( _u() );
   }

   double extremeValue( double a = 0., double c = 1. ) { // Extreme Value
                                                     // location a, shape c
      assert( c > 0. );

      return a + c * log( -log( _u() ) );
   }

   double fRatio( int v, int w ) { // F Ratio (v and w >= 1)

      assert( v >= 1 && w >= 1 );

      return ( chiSquare( v ) / v ) / ( chiSquare( w ) / w );
   }

   double gamma( double a, double b, double c ) { // Gamma
                                                 // location a, scale b, shape c
      assert( b > 0. && c > 0. );

      if ( c < 1. ) {
         const double C = 1. + c / M_E;
         while ( true ) {
            double p = C * _u();
            if ( p > 1. ) {
               double y = -log( ( C - p ) / c );
               if ( _u() <= pow( y, c - 1. ) ) return a + b * y;
            }
            else {
               double y = pow( p, 1. / c );
               if ( _u() <= exp( -y ) ) return a + b * y;
            }
         }
      }
      else if ( c == 1.0 ) return exponential( a, b );
      else {
         const double A = 1. / sqrt( 2. * c - 1. );
         const double B = c - log( 4. );
         const double Q = c + 1. / A;
         const double T = 4.5;
         const double D = 1. + log( T );
         while ( true ) {
            double p1 = _u();
            double p2 = _u();
            double v = A * log( p1 / ( 1. - p1 ) );
            double y = c * exp( v );
            double z = p1 * p1 * p2;
            double w = B + Q * v - y;
            if ( w + D - T * z >= 0. || w >= log( z ) ) return a + b * y;
         }
      }
   }

   double laplace( double a = 0., double b = 1. ) { // Laplace
                                                // (or double exponential)
      assert( b > 0. );

      // composition method

      if ( bernoulli( 0.5 ) ) return a + b * log( _u() );
      else                    return a - b * log( _u() );
   }

   double logarithmic( double xMin = 0., double xMax = 1. ) { // Logarithmic

      assert( xMin < xMax );

      double a = xMin;          // location parameter
      double b = xMax - xMin;   // scale parameter

      // use convolution formula for product of two IID uniform variates

      return a + b * _u() * _u();
   }

   double logistic( double a = 0., double c = 1. ) { // Logistic

      assert( c > 0. );

      return a - c * log( 1. / _u() - 1. );
   }

   double lognormal( double a, double mu, double sigma ) { // Lognormal

      return a + exp( normal( mu, sigma ) );
   }

   double normal( double mu = 0., double sigma = 1. ) { // Normal

      assert( sigma > 0. );

      static bool f = true;
      static double p2, q;
      double p1, p;

      if ( f ) {
         do {
            p1 = uniform( -1., 1. );
            p2 = uniform( -1., 1. );
            p = p1 * p1 + p2 * p2;
         } while ( p >= 1. );
         f = false;
         q = sqrt( -2. * log( p ) / p );
         return mu + sigma * p1 * q;
      }
      f = true;
      return mu + sigma * p2 * q;
   }

   double parabolic( double xMin = 0., double xMax = 1. ) { // Parabolic

      assert( xMin < xMax );

      double a    = 0.5 * ( xMin + xMax );        // location parameter
      double yMax = _parabola( a, xMin, xMax );   // maximum function range

      return userSpecified( _parabola, xMin, xMax, 0., yMax );
   }

   double pareto( double c ) { // Pareto
                               // shape c
      assert( c > 0. );

      return pow( _u(), -1. / c );
   }

   double pearson5( double b, double c ) { // Pearson Type 5
                                           // scale b, shape c
      assert( b > 0. && c > 0. );

      return 1. / gamma( 0., 1. / b, c );
   }

   double pearson6( double b, double v, double w ) { // Pearson Type 6
                                                     // scale b, shape v & w
      assert( b > 0. && v > 0. && w > 0. );

      return gamma( 0., b, v ) / gamma( 0., b, w );
   }

   double power( double c ) { // Power
                              // shape c
      assert( c > 0. );

      return pow( _u(), 1. / c );
   }

   double rayleigh( double a, double b ) { // Rayleigh
                                           // location a, scale b
      assert( b > 0. );

      return a + b * sqrt( -log( _u() ) );
   }

   double studentT( int df ) { // Student's T
                               // degres of freedom df
      assert( df >= 1 );

      return normal() / sqrt( chiSquare( df ) / df );
   }

   double triangular( double xMin = 0.,     // Triangular
                      double xMax = 1.,     // with default interval [0,1)
                      double c    = 0.5 ) { // and default mode 0.5

      assert( xMin < xMax && xMin <= c && c <= xMax );

      double p = _u(), q = 1. - p;

      if ( p <= ( c - xMin ) / ( xMax - xMin ) )
         return xMin + sqrt( ( xMax - xMin ) * ( c - xMin ) * p );
      else
         return xMax - sqrt( ( xMax - xMin ) * ( xMax - c ) * q );
   }

   double uniform( double xMin = 0., double xMax = 1. ) { // Uniform
                                                          // on [xMin,xMax)
      assert( xMin < xMax );

      return xMin + ( xMax - xMin ) * _u();
   }

   double userSpecified(                // User-Specified Distribution
        double( *usf )(                 // pointer to user-specified function
             double,                    // x
             double,                    // xMin
             double ),                  // xMax
        double xMin, double xMax,       // function domain
        double yMin, double yMax ) {    // function range

      assert( xMin < xMax && yMin < yMax );

      double x, y, areaMax = ( xMax - xMin ) * ( yMax - yMin );

      // acceptance-rejection method

      do {
         x = uniform( 0.0, areaMax ) / ( yMax - yMin ) + xMin;
         y = uniform( yMin, yMax );

      } while ( y > usf( x, xMin, xMax ) );

      return x;
   }

   double weibull( double a, double b, double c ) { // Weibull
                                                    // location a, scale b,
      assert( b > 0. && c > 0. );                   // shape c

      return a + b * pow( -log( _u() ), 1. / c );
   }

// Discrete Distributions

   bool bernoulli( double p = 0.5 ) { // Bernoulli Trial

      assert( 0. <= p && p <= 1. );

      return _u() < p;
   }

   int binomial( int n, double p ) { // Binomial

      assert( n >= 1 && 0. <= p && p <= 1. );

      int sum = 0;
      for ( int i = 0; i < n; i++ ) sum += bernoulli( p );
      return sum;
   }

   int geometric( double p ) { // Geometric

      assert( 0. < p && p < 1. );

      return int( log( _u() ) / log( 1. - p ) );
   }

   int hypergeometric( int n, int N, int K ) {          // Hypergeometric
                                                        // trials n, size N,
      assert( 0 <= n && n <= N && N >= 1 && K >= 0 );   // successes K

      int count = 0;
      for ( int i = 0; i < n; i++, N-- ) {

         double p = double( K ) / double( N );
         if ( bernoulli( p ) ) { count++; K--; }
      }
      return count;
   }

   void multinomial( int    n,            // Multinomial
                     double p[],          // trials n, probability vector p,
                     int    count[],      // success vector count,
                     int    m ) {         // number of disjoint events m

      assert( m >= 2 );   // at least 2 events
      double sum = 0.;
      for ( int bin = 0; bin < m; bin++ ) sum += p[ bin ];    // probabilities
      assert( sum == 1. );                                    // must sum to 1

      for ( int bin = 0; bin < m; bin++ ) count[ bin ] = 0;   // initialize

      // generate n uniform variates in the interval [0,1) and bin the results

      for ( int i = 0; i < n; i++ ) {

         double lower = 0., upper = 0., u = _u();

         for ( int bin = 0; bin < m; bin++ ) {

         // locate subinterval, which is of length p[ bin ],
         // that contains the variate and increment the corresponding counter

            lower = upper;
            upper += p[ bin ];
            if ( lower <= u && u < upper ) { count[ bin ]++; break; }
         }
      }
   }

   int negativeBinomial( int s, double p ) { // Negative Binomial
                                             // successes s, probability p
      assert( s >= 1 && 0. < p && p < 1. );

      int sum = 0;
      for ( int i = 0; i < s; i++ ) sum += geometric( p );
      return sum;
   }

   int pascal( int s, double p ) { // Pascal
                                   // successes s, probability p
      return negativeBinomial( s, p ) + s;
   }

   int poisson( double mu ) { // Poisson

      assert ( mu > 0. );

      double a = exp( -mu );
      double b = 1.;

      int i;
      for ( i = 0; b >= a; i++ ) b *= _u();
      return i - 1;
   }

   int uniformDiscrete( int i, int j ) { // Uniform Discrete
                                         // inclusive i to j
      assert( i < j );

      return i + int( ( j - i + 1 ) * _u() );
   }

// Empirical and Data-Driven Distributions

   double empirical( void ) { // Empirical Continuous

      static std::vector< double > x, cdf;
      static int              n;
      static bool             init = false;

      if ( !init ) {
         std::ifstream in( "empiricalDistribution" );
         if ( !in ) {
            std::cerr << "Cannot open \"empiricalDistribution\" input file" << std::endl;
            exit( 1 );
         }
         double value, prob;
         while ( in >> value >> prob ) {   // read in empirical distribution
            x.push_back( value );
            cdf.push_back( prob );
         }
         n = x.size();
         init = true;

         // check that this is indeed a cumulative distribution

         assert( 0. == cdf[ 0 ] && cdf[ n - 1 ] == 1. );
         for ( int i = 1; i < n; i++ ) assert( cdf[ i - 1 ] < cdf[ i ] );
      }

      double p = _u();
      for ( int i = 0; i < n - 1; i++ )
         if ( cdf[ i ] <= p && p < cdf[ i + 1 ] )
            return x[ i ] + ( x[ i + 1 ] - x[ i ] ) *
                            ( p - cdf[ i ] ) / ( cdf[ i + 1 ] - cdf[ i ] );
      return x[ n - 1 ];
   }

   int empiricalDiscrete( void ) { // Empirical Discrete

      static std::vector< int >    k;
      static std::vector< double > f[ 2 ];   // f[ 0 ] is pdf and f[ 1 ] is cdf
      static double           max;
      static int              n;
      static bool             init = false;

      if ( !init ) {
         std::ifstream in ( "empiricalDiscrete" );
         if ( !in ) {
            std::cerr << "Cannot open \"empiricalDiscrete\" input file" << std::endl;
            exit( 1 );
         }
         int value;
         double freq;
         while ( in >> value >> freq ) {   // read in empirical data
            k.push_back( value );
            f[ 0 ].push_back( freq );
         }
         n = k.size();
         init = true;

         // form the cumulative distribution

         f[ 1 ].push_back( f[ 0 ][ 0 ] );
         for ( int i = 1; i < n; i++ )
            f[ 1 ].push_back( f[ 1 ][ i - 1 ] + f[ 0 ][ i ] );

         // check that the integer points are in ascending order

         for ( int i = 1; i < n; i++ ) assert( k[ i - 1 ] < k[ i ] );

         max = f[ 1 ][ n - 1 ];
      }

      // select a uniform variate between 0 and the max value of the cdf

      double p = uniform( 0., max );

      // locate and return the corresponding index

      for ( int i = 0; i < n; i++ ) if ( p <= f[ 1 ][ i ] ) return k[ i ];
      return k[ n - 1 ];
   }

   double sample( bool replace = true ) { // Sample w or w/o replacement from a
                                          // distribution of 1-D data in a file
      static std::vector< double > v;     // vector for sampling with replacement
      static bool init = false;           // flag that file has been read in
      static int n;                       // number of data elements in the file
      static int index = 0;               // subscript in the sequential order

      if ( !init ) {
         std::ifstream in( "sampleData" );
         if ( !in ) {
            std::cerr << "Cannot open \"sampleData\" file" << std::endl;
            exit( 1 );
         }
         double d;
         while ( in >> d ) v.push_back( d );
         in.close();
         n = v.size();
         init = true;
         if ( replace == false ) {   // sample without replacement

            // shuffle contents of v once and for all
            // Ref: Knuth, D. E., The Art of Computer Programming, Vol. 2:
            //      Seminumerical Algorithms. London: Addison-Wesley, 1969.

            for ( int i = n - 1; i > 0; i-- ) {
               int j = int( ( i + 1 ) * _u() );
               std::swap( v[ i ], v[ j ] );
            }
         }
      }

      // return a random sample

      if ( replace )                                // sample w/ replacement
         return v[ uniformDiscrete( 0, n - 1 ) ];
      else {                                        // sample w/o replacement
         assert( index < n );                       // retrieve elements
         return v[ index++ ];                       // in sequential order
      }
   }

   void sample( double x[], int ndim ) { // Sample from a given distribution
                                         // of multi-dimensional data
      static const int N_DIM = 6;
      assert( ndim <= N_DIM );

      static std::vector< double > v[ N_DIM ];
      static bool init = false;
      static int n;

      if ( !init )  {
         std::ifstream in( "sampleData" );
         if ( !in ) {
            std::cerr << "Cannot open \"sampleData\" file" << std::endl;
            exit( 1 );
         }
         double d;
         while ( !in.eof() ) {
            for ( int i = 0; i < ndim; i++ ) {
               in >> d;
               v[ i ].push_back( d );
            }
         }
         in.close();
         n = v[ 0 ].size();
         init = true;
      }
      int index = uniformDiscrete( 0, n - 1 );
      for ( int i = 0; i < ndim; i++ ) x[ i ] = v[ i ][ index ];
   }

   point2d stochasticInterpolation( void ) { // Stochastic Interpolation

   // Refs: Taylor, M. S. and J. R. Thompson, Computational Statistics & Data Analysis, Vol. 4, pp. 93-101, 1986;
   //       Thompson, J. R., Empirical Model Building, pp. 108-114, Wiley, 1989;
   //       Bodt, B. A. and M. S. Taylor, A Data Based Random Number Generator for A Multivariate Distribution -
   //       A User’s Manual, ARBRL-TR-02439, BRL, APG, MD, Nov. 1982.

      static std::vector<point2d> data;
      static point2d              min, max;
      static int                  m;
      static double               lower, upper;
      static bool                 init = false;

      if ( !init ) {
         std::ifstream in( "stochasticData" );
         if ( !in ) {
            std::cerr << "Cannot open \"stochasticData\" input file" << std::endl;
            exit( 1 );
         }

         // read in the data and set min and max values

         min.x = min.y = FLT_MAX;
         max.x = max.y = FLT_MIN;
         point2d p;
         while ( in >> p.x >> p.y ) {

            min.x = ( p.x < min.x ? p.x : min.x );
            min.y = ( p.y < min.y ? p.y : min.y );
            max.x = ( p.x > max.x ? p.x : max.x );
            max.y = ( p.y > max.y ? p.y : max.y );

            data.push_back( p );
         }
         in.close();
         init = true;

         // scale the data so that each dimension will have equal weight

         for ( unsigned int i = 0; i < data.size(); i++ ) {

            data[ i ].x = ( data[ i ].x - min.x ) / ( max.x - min.x );
            data[ i ].y = ( data[ i ].y - min.y ) / ( max.y - min.y );
         }

         // set m, the number of points in a neighborhood of a given point

         m = data.size() / 20;       // 5% of all the data points
         if ( m < 5  ) m = 5;        // but no less than 5
         if ( m > 20 ) m = 20;       // and no more than 20

         lower = ( 1. - sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
         upper = ( 1. + sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
      }

      // uniform random selection of a data point (with replacement)

      point2d origin = data[ uniformDiscrete( 0, data.size() - 1 ) ];

      // make this point the origin of the coordinate system

      for ( unsigned int n = 0; n < data.size(); n++ ) data[ n ] -= origin;

      // sort the data with respect to its distance (squared) from this origin

      std::sort( data.begin(), data.end(), dSquared() );

      // find the mean value of the data in the neighborhood about this point

      point2d mean;
      mean.x = mean.y = 0.;
      for ( int n = 0; n < m; n++ ) mean += data[ n ];
      mean /= double( m );

      // select a random linear combination of the points in this neighborhood

      point2d p;
      p.x = p.y = 0.;
      for ( int n = 0; n < m; n++ ) {

         double rn;
         if ( m == 1 ) rn = 1.;
         else          rn = uniform( lower, upper );
         p.x += rn * ( data[ n ].x - mean.x );
         p.y += rn * ( data[ n ].y - mean.y );
      }

      // restore the data to its original form

      for ( unsigned int n = 0; n < data.size(); n++ ) data[ n ] += origin;

      // use mean and original point to translate the randomly-chosen point

      p += mean;
      p += origin;

      // scale randomly-chosen point to the dimensions of the original data

      p.x = p.x * ( max.x - min.x ) + min.x;
      p.y = p.y * ( max.y - min.y ) + min.y;

      return p;
   }

// Multivariate Distributions

   cartesianCoord bivariateNormal( double muX    = 0.,
                                   double sigmaX = 1.,
                                   double muY    = 0.,
                                   double sigmaY = 1. ) { // Bivariate Gaussian

      assert( sigmaX > 0. && sigmaY > 0. );

      return make_pair( normal( muX, sigmaX ), normal( muY, sigmaY ) );
   }

   cartesianCoord bivariateUniform( double xMin = -1.,
                                    double xMax =  1.,
                                    double yMin = -1.,
                                    double yMax =  1. ) { // Bivariate Uniform

      assert( xMin < xMax && yMin < yMax );

      double x0 = 0.5 * ( xMin + xMax );
      double y0 = 0.5 * ( yMin + yMax );
      double a  = 0.5 * ( xMax - xMin );
      double b  = 0.5 * ( yMax - yMin );
      double x, y;

      do {
         x = uniform( -1., 1. );
         y = uniform( -1., 1. );

      } while( x * x + y * y > 1. );

      return make_pair( x0 + a * x, y0 + b * y );
   }

   polarCoord circularUniform( double rMin  = 0.,
                               double rMax  = 1.,
                               double thMin = 0.,
                               double thMax = 2. * M_PI ) { // Circular Uniform

      assert( 0 <= rMin && rMin <= rMax && thMin <= thMax );

      double r = sqrt( uniform( rMin * rMin, rMax * rMax ) );
      double th = uniform( thMin, thMax );

      return make_pair( r, th );
   }

   cartesianCoord corrNormal( double r,
                              double muX    = 0.,
                              double sigmaX = 1.,
                              double muY    = 0.,
                              double sigmaY = 1. ) { // Correlated Normal

      assert( -1. <= r && r <= 1. &&          // bounds on correlation coeff
              sigmaX > 0. && sigmaY > 0. );   // positive std dev

      double x = normal();
      double y = normal();

      y = r * x + sqrt( 1. - r * r ) * y;     // correlate the variables

      // translate and scale
      return make_pair( muX + sigmaX * x, muY + sigmaY * y );
   }

   cartesianCoord corrUniform( double r,
                               double xMin = 0.,
                               double xMax = 1.,
                               double yMin = 0.,
                               double yMax = 1. ) { // Correlated Uniform

      assert( -1. <= r && r <= 1. &&          // bounds on correlation coeff
              xMin < xMax && yMin < yMax );   // bounds on domain

      double x0 = 0.5 * ( xMin + xMax );
      double y0 = 0.5 * ( yMin + yMax );
      double a  = 0.5 * ( xMax - xMin );
      double b  = 0.5 * ( yMax - yMin );
      double x, y;

      do {
         x = uniform( -1., 1. );
         y = uniform( -1., 1. );

      } while ( x * x + y * y > 1. );

      y = r * x + sqrt( 1. - r * r ) * y;   // correlate the variables

      // translate and scale
      return make_pair( x0 + a * x, y0 + b * y );
   }

private:
   // function returns an associative array where each element of the vector is the key and the rank is the value
   inline std::map<double,int> _rank( std::vector<double> v ) {   // NB: pass a copy of the vector, not a reference

      std::sort( v.begin(), v.end() );   // sort the vector in ascending order
      std::map<double,int> r;
      for ( unsigned int i = 0; i < v.size(); i++ ) r[v[i]] = i;   // map the element to its rank
      return r;
   }

public:

   // correlate two distributions without changing the marginal distributions
   // (Thanks to Dr. Joseph Collins for describing this technique.  For an understanding of the theory,
   // and more general cases of any number of distributions, see "Inducing Dependence in Multivariate Random Samples,"
        // unpublished paper, August 25, 2011, and references cited therein.)
   void corrDist( std::vector<double>& dist1, std::vector<double>& dist2, double rankCorr ) {   // the two distributions are reordered to induce the correlation

      std::vector<double> t1( dist1 ), t2( dist2 );   // copy the two distributions
      std::sort( t1.begin(), t1.end() );   // sort the copies in ascending order
      std::sort( t2.begin(), t2.end() );

      double x, y, c = 2. * sin( rankCorr * M_PI / 6. ), s = sqrt( 1. - c * c );
      const int N = dist1.size();
      std::vector<double> u(N), v(N);
      for ( int i = 0; i < N; i++ ) {   // generate two correlated vectors with the given rank correlation

         x = normal();
         y = normal();
         y = c * x + s * y;   // perform a rotation to induce the correlation
         u[i] = x ;
         v[i] = y;
      }
      std::map<double,int> rank_u = _rank( u );   // generate maps from the values to the corresponding ranks
      std::map<double,int> rank_v = _rank( v );

      for ( int i = 0; i < N; i++ ) {   // apply these maps as index functions to the sorted distributions

         dist1[i] = t1[ rank_u[ u[i] ] ];
         dist2[i] = t2[ rank_v[ v[i] ] ];
      }
   }

   sphericalCoord spherical( double thMin = 0.,
                             double thMax = M_PI,
                             double phMin = 0.,
                             double phMax = 2. * M_PI ) { // Uniform Spherical

      assert( 0. <= thMin && thMin < thMax && thMax <= M_PI &&
              0. <= phMin && phMin < phMax && phMax <= 2. * M_PI );

      // return polar angle and azimuth
      return make_pair( acos( uniform( cos( thMax ), cos( thMin ) ) ), uniform( phMin, phMax ) );
   }

   void sphericalND( double x[], int n ) { // Uniform over the surface of an n-dimensional unit sphere

      // generate a point inside the unit n-sphere by normal polar method

      double r2 = 0.;
      for ( int i = 0; i < n; i++ ) {
         x[ i ] = normal();
         r2 += x[ i ] * x[ i ];
      }

      // project the point onto the surface of the unit n-sphere by scaling

      const double A = 1. / sqrt( r2 );
      for ( int i = 0; i < n; i++ ) x[ i ] *= A;
   }

// Number Theoretic Distributions

   double avoidance( void ) { // Maximal Avoidance (1-D), overloaded for convenience

      double x[ 1 ];
      avoidance( x, 1 );
      return x[ 0 ];
   }

   void avoidance( double x[], unsigned int ndim ) { // Maximal Avoidance (N-D)

      static const unsigned int MAXBIT = 30;
      static const unsigned int MAXDIM = 6;

      assert( ndim <= MAXDIM );

      static unsigned long ix[ MAXDIM + 1 ] = { 0 };
      static unsigned long *u[ MAXBIT + 1 ];
      static unsigned long mdeg[ MAXDIM + 1 ] = { // degree of
         0,                                        // primitive polynomial
         1, 2, 3, 3, 4, 4
      };
      static unsigned long p[ MAXDIM + 1 ] = {     // decimal encoded
         0,                                       // interior bits
         0, 1, 1, 2, 1, 4
      };
      static unsigned long v[ MAXDIM * MAXBIT + 1 ] = {
          0,
          1,  1, 1,  1,  1,  1,
          3,  1, 3,  3,  1,  1,
          5,  7, 7,  3,  3,  5,
         15, 11, 5, 15, 13,  9
      };

      static double fac;
      static int in = -1;
      unsigned int j, k;
      unsigned long i, m, pp;

      if ( in == -1 ) {
         in = 0;
         fac = 1. / ( 1L << MAXBIT );
         for ( j = 1, k = 0; j <= MAXBIT; j++, k += MAXDIM ) u[ j ] = &v[ k ];
         for ( k = 1; k <= MAXDIM; k++ ) {
            for ( j = 1; j <= mdeg[ k ]; j++ ) u[ j ][ k ] <<= ( MAXBIT - j );
            for ( j = mdeg[ k ] + 1; j <= MAXBIT; j++ ) {
               pp = p[ k ];
               i = u[ j - mdeg[ k ] ][ k ];
               i ^= ( i >> mdeg[ k ] );
               for ( int n = mdeg[ k ] - 1; n >= 1; n-- ) {
                  if ( pp & 1 ) i ^= u[ j - n ][ k ];
                  pp >>= 1;
               }
               u[ j ][ k ] = i;
            }
         }
      }
      m = in++;
      for ( j = 0; j < MAXBIT; j++, m >>= 1 ) if ( !( m & 1 ) ) break;
      if ( j >= MAXBIT ) exit( 1 );
      m = j * MAXDIM;
      for ( k = 0; k < ndim; k++ ) {
         ix[ k + 1 ] ^= v[ m + k + 1 ];
         x[ k ] = ix[ k + 1 ] * fac;
      }
   }

private:

   Generator<Typename> *_gen;
   static const unsigned int N_BITS = CHAR_BIT * sizeof( Typename );   // number of bits (32 or 64)

   long double _u( void ) {

      if ( N_BITS == 32 ) return _gen->rng32_01();
      else                return _gen->rng64_01();
   }

   static double _parabola( double x, double xMin, double xMax ) { // parabola

      if ( x < xMin || x > xMax ) return 0.0;

      double a    = 0.5 * ( xMin + xMax );   // location parameter
      double b    = 0.5 * ( xMax - xMin );   // scale parameter
      double yMax = 0.75 / b;

      return yMax * ( 1. - ( x - a ) * ( x - a ) / ( b * b ) );
   }
}; // Random class
} // rnd namespace

#endif // RANDOM_H
```