

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

Richard Saucier

February 2017

Contents

List of Figures	iii
List of Tables	v
List of Listings	vi
1 Summary	1
2 Introduction	2
3 Methods for Generating Random Number Distributions	2
3.1 Inverse Transformation	2
3.2 Composition	4
3.3 Convolution	5
3.4 Acceptance–Rejection	6
3.5 Sampling and Data–Driven Techniques	7
3.6 Techniques Based on Number Theory	8
3.7 Monte Carlo Simulation	8
3.8 Correlated Bivariate Distributions	9
3.9 Truncated Distributions	10
4 Parameter Estimation	11
4.1 Linear Regression (Least–Squares Estimate)	11
4.2 Maximum Likelihood Estimation	11
5 Probability Distribution Functions	12
5.1 Continuous Distributions	12
5.1.1 Arcsine	14
5.1.2 Beta	15
5.1.3 Cauchy (Lorentz)	16
5.1.4 Chi-Square	17
5.1.5 Cosine	18
5.1.6 Double Log	19
5.1.7 Erlang	20
5.1.8 Exponential	21
5.1.9 Extreme Value	22
5.1.10 F Ratio	23
5.1.11 Gamma	24
5.1.12 Laplace (Double Exponential)	26
5.1.13 Logarithmic	27
5.1.14 Logistic	28
5.1.15 Lognormal	29
5.1.16 Normal (Gaussian)	30
5.1.17 Parabolic	31
5.1.18 Pareto	32
5.1.19 Pearson’s Type 5 (Inverted Gamma)	33
5.1.20 Pearson’s Type 6	34
5.1.21 Power	35
5.1.22 Raab-Green	36
5.1.23 Rayleigh	37
5.1.24 Student’s t	38
5.1.25 Triangular	39
5.1.26 Uniform	40

5.1.27	User-Specified	41
5.1.28	Weibull	42
5.2	Discrete Distributions	43
5.2.1	Bernoulli	44
5.2.2	Binomial	45
5.2.3	Geometric	46
5.2.4	Hypergeometric	47
5.2.5	Multinomial	48
5.2.6	Negative Binomial	49
5.2.7	Pascal	50
5.2.8	Poisson	51
5.2.9	Uniform Discrete	52
5.3	Empirical and Data-Driven Distributions	53
5.3.1	Empirical	54
5.3.2	Empirical Discrete	55
5.3.3	Sampling with and without Replacement	56
5.3.4	Stochastic Interpolation	57
5.4	Bivariate Distributions	59
5.4.1	Bivariate Normal	60
5.4.2	Bivariate Uniform	61
5.4.3	Circular Uniform	62
5.4.4	Correlated Normal	63
5.4.5	Correlated Uniform	64
5.4.6	Spherical Uniform	65
5.4.7	Spherical Uniform in N-Dimensions	66
5.5	Distributions Generated From Number Theory	67
5.5.1	Tausworthe Random Bit Generator	67
5.5.2	Maximal Avoidance (Quasi-Random)	69
6	Discussion and Examples	70
6.1	Making Sense of the Discrete Distributions	70
6.2	Adding New Distributions	71
7	Comparison of the Generators	72
	References	73
	Appendices	74
A	Linear Congruential Generator	75
B	Linear Feedback Shift Generator	80
C	Multiply with Carry Generator	90
D	Code Listings	93

List of Figures

1	Inverse transform method	3
2	Probability density generated from uniform areal density	7
3	Histogram of a randomly oriented cube via Monte Carlo simulation	9
4	Coordinate rotation to induce correlations	9
5	Plot of arcsine PDF	14
6	Plot of arcsine CDF	14
7	Plot of beta PDF	15
8	Plot of beta CDF	15
9	Plot of Cauchy PDF	16
10	Plot of Cauchy CDF	16
11	Plot of chi-square PDF	17
12	Plot of chi-square CDF	17
13	Plot of cosine PDF	18
14	Plot of cosine CDF	18
15	Plot of double log PDF	19
16	Plot of double log CDF	19
17	Plot of Erlang PDF	20
18	Plot of Erlang CDF	20
19	Plot of Exponential PDF	21
20	Plot of exponential CDF	21
21	Plot of extreme value PDF	22
22	Plots of extreme value CDF	22
23	Plot of F Ratio DF	23
24	Plot of F Ratio CDF	23
25	Plot of gamma PDF	25
26	Plot of gamma CDF	25
27	Plot of Laplace PDF	26
28	Plot of Laplace CDF	26
29	Plot of logarithmic PDF	27
30	Plot of logarithmic CDF	27
31	Plot of logistic PDF	28
32	Plot of logistic CDF	28
33	Plot of lognormal PDF	29
34	Plot of lognormal CDF	29
35	Plot of normal PDF	30
36	Plot of normal CDF	30
37	Plot of parabolic PDF	31
38	Plot of parabolic CDF	31
39	Plot of Pareto PDF	32
40	Plot of Pareto CDF	32
41	Plot of Pearson's type 5 PDF	33
42	Plot of Pearson's type 5 CDF	33
43	Plot of Pearson's type 6 PDF	34
44	Plot of Pearson's type 6 CDF	34
45	Plot of Power PDF	35
46	Plot of Power CDF	35
47	Plot of Raab-Green PDF	36
48	Plot of Raab-Green CDF	36
49	Plot of Rayleigh PDF	37
50	Plot of Rayleigh CDF	37
51	Plot of Student's t PDF	38
52	Plot of Student's t CDF	38

53	Plot of triangular PDF	39
54	Plot of triangular CDF	39
55	Plot of uniform PDF	40
56	Plot of uniform CDF	40
57	Plot of user-specified PDF	41
58	Plot of user-specified CDF	41
59	Plot of Weibull PDF	42
60	Plot of Weibull CDF	42
61	Histogram of binomial PDF	45
62	Histogram of binomial CDF	45
63	Histogram of geometric PDF	46
64	Histogram of Geometric CDF	46
65	Histogram of hypergeometric PDF	47
66	Histogram of hypergeometric CDF	47
67	Histogram of negative binomial PDF	49
68	Histogram of negative binomial CDF	49
69	Histogram of Pascal PDF	50
70	Histogram of Pascal CDF	50
71	Histogram of Poisson PDF	51
72	Histogram of Poisson CDF	51
73	Histogram of uniform discrete PDF	52
74	Histogram of uniform discrete CDF	52
75	bivariateNormal(0, 1, 0, 1)	60
76	bivariateNormal(0, 1, -1, 0.5)	60
77	bivariateUniform(0, 1, 0, 1)	61
78	bivariateUniform(0, 1, -1, 0.5)	61
79	circularUniform()	62
80	circularUniform(0.25, 0.75, 0, 270 * M_PI / 180)	62
81	corrNormal(0.5, 0, 1, 0, 1)	63
82	corrNormal(-0.75, 0, 1, 0, 0.5)	63
83	corrUniform(0.5, 0, 1, 0, 1)	64
84	corrUniform(-0.75, 0, 1, 0, 0.5)	64
85	Uniform spherical distribution via calls to spherical()	65
86	100 maximal avoidance data points	69
87	500 maximal avoidance data points	69
88	1000 maximal avoidance data points	69
89	1000 uniformly distributed data points	69

List of Tables

1	Properties for Selecting the Appropriate Continuous Distribution	13
2	Parameters and Description for Selecting the Appropriate Discrete Distribution	43
3	Parameters and Description for Selecting the Appropriate Empirical Distribution	53
4	Description and Output for Selecting the Appropriate Bivariate Distribution	59
5	Performance of RNGs (in Millions/s)	72
6	Results from TestU01 battery of tests	72
7	Cycle Length and Jump Time	72
8	Constants for “Jump Back” formula	79
9	Constants for Linear Congruential Generators	79
10	<code>mod_math</code> Reference Guide	93

List of Listings

D-1	mod_math.h	93
D-2	mod_math.cpp	99
D-3	Bitmatrix.h	101
D-4	lfsr88.h	102
D-5	lfsr113.h	104
D-6	kiss.h	106
D-7	jkiss.h	109
D-8	lfsr258.h	112
D-9	jlkiss.h	116
D-10	jlkiss64.h	118
D-11	Generator.h	122
D-12	Random.h	122

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:

```
1 #include "Random.h"
2 #include <iostream>
3 #include <cstdlib>
4
5 int main( void ) {
6
7     Generator<uint32_t> *rng = new JKISS::jkiss; // create a new generator using jkiss as the engine
8     dist::Distribution<uint32_t> rnd( rng );      // create a distribution object and initialize it to use this generator
9     for ( int i = 0; i < 1000; i++ )
10         std::cout << rnd.normal() << std::endl; // output a normally-distributed pseudorandom number
11
12     return 0;
13 }
```

No libraries are required and there is nothing to build; one merely needs to include the header file 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 interval $[0, 1)$. 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 $U \sim U(0, 1)$.
2. Return $X = F^{-1}(U)$.

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

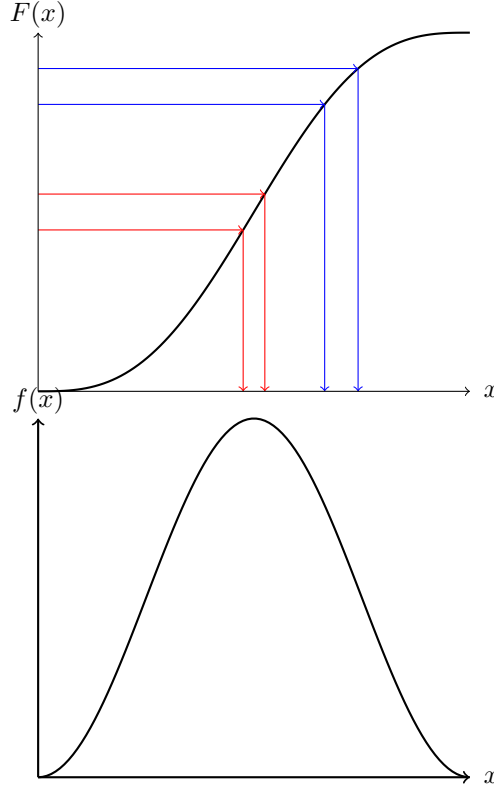


Figure 1. Inverse transform method

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

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

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

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

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

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

where

$$\sum_{i=1}^n p_i = 1 \quad (4)$$

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 $U \sim U(0, 1)$.
3. Return $X = F_i^{-1}(U)$.

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

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

This can also be written as

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

where

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

Now each of these has an indefinite integral, namely

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

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

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

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

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

* 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

$$F_{X+Y}(u) = \iint_{x+y \leq u} f(x, y) \, dx \, dy = \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{u-x} f(x, y) \, dy \right) \, dx. \quad (11)$$

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

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

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

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

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

$$f_{X+Y}(u) = \int_0^1 f(u-x) \, dx = \int_{u-1}^u f(t) \, dt = \begin{cases} u & u \leq 1 \\ 2-u & 1 < u \leq 2 \end{cases} \quad (14)$$

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

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

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

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

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 $i = 1, 2, \dots, n$.
2. Set $X = X_1 + X_2 + \dots + X_n$.

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

$$X = -b \ln X_1 - \dots - b \ln X_c = -b \ln(X_1 \cdots X_c). \quad (17)$$

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

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

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

$$F_{XY}(u) = \iint_{xy \leq u} f(x, y) \, dx \, dy = \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{u/x} f(x, y) \, dy \right) dx. \quad (19)$$

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

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

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

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

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

$$f_{XY}(u) = \int_u^1 \frac{1}{x} \, dx = -\ln u. \quad (22)$$

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:

$$F_{Y/X}(u) = \iint_{y/x \leq u} f(x, y) \, dx \, dy = \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{ux} f(x, y) \, dy \right) dx. \quad (23)$$

Differentiating this to get the density,

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

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

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

and we find that

$$f_{Y/X}(u) = \frac{1}{\pi} \int_0^{\infty} e^{-(1+u^2)x^2/2} x \, dx = \frac{1}{\pi(1+u^2)}. \quad (26)$$

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.

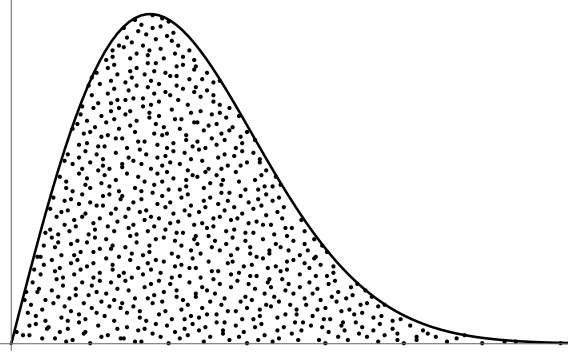


Figure 2. Probability density generated from uniform areal density

A simple way to do this is as follows.

1. Select $X \sim U(x_{\min}, x_{\max})$.
2. Independently select $Y \sim U(0, y_{\max})$.
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

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

Then the more efficient algorithm is as follows:

1. Select $A \sim U(0, A_{\max})$.
2. Compute $X = \hat{F}^{-1}(A)$.
3. Independently select $Y \sim U(0, \hat{f}(X))$.
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 $A \sim U(0, 2.97188)$.
2. Compute $X = A/2.97188$.
3. Independently select $Y \sim U(0, 2.97188)$.
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

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

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

$$\gamma(\theta, \phi) = \sin \theta \cos \phi + \sin \theta \sin \phi + \cos \theta. \quad (29)$$

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 $f(\gamma)$.

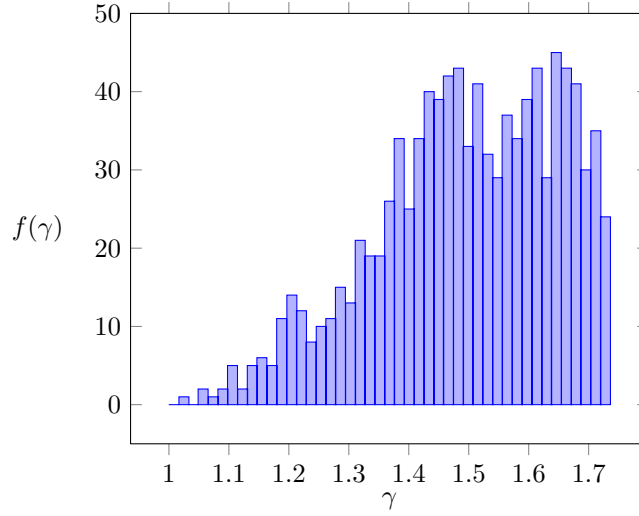


Figure 3. Histogram of a randomly oriented cube via Monte Carlo simulation

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.

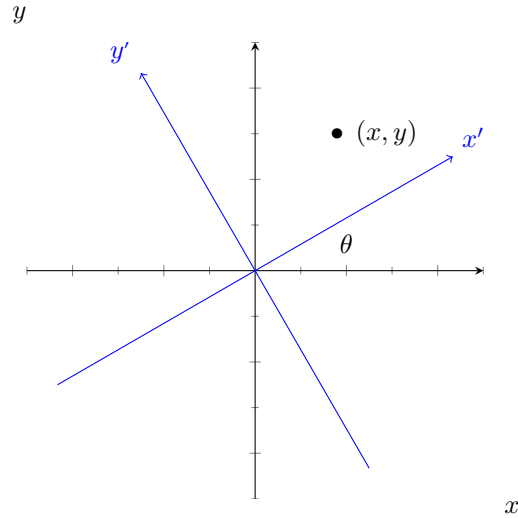


Figure 4. Coordinate rotation to induce correlations

The transformation between the two coordinate systems is given by

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

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

$$x' = \rho x + \sqrt{1 - \rho^2} y \quad (31)$$

induces the desired correlation. To check this,

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

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

Here are some special cases:

$$\left\{ \begin{array}{lll} \theta = 0 & \rho = 1 & x' = x \\ \theta = \pi/2 & \rho = 0 & x' \text{ is independent of } x \\ \theta = \pi & \rho = -1 & x' = -x \end{array} \right. \quad (33)$$

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:

$$\tilde{f}(x) \equiv \begin{cases} \frac{f(x)}{F(b) - F(a)} & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}, \quad (34)$$

which has corresponding truncated distribution

$$\tilde{F}(x) \equiv \begin{cases} 0 & x < a \\ \frac{F(x) - F(a)}{F(b) - F(a)} & a \leq x \leq b \\ 1 & x > b \end{cases}. \quad (35)$$

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

1. Generate $U \sim \text{U}(0, 1)$.
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 $a \leq X \leq b$, 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 $a = 0$:

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

We first sort the data x_i in ascending order:

$$x_1 \leq x_2 \leq x_3 \leq \cdots \leq x_N. \quad (37)$$

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

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

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:

$$L(\beta) = f_\beta(X_1)f_\beta(X_2) \cdots f_\beta(X_N). \quad (39)$$

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

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

The log-likelihood function is

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

* 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,

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

This is a maximum when both

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

and we find

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

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	$f(x) = \begin{cases} \frac{1}{\pi\sqrt{x(1-x)}} & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} 0 & x < 0 \\ \frac{2}{\pi} \sin^{-1}(\sqrt{x}) & 0 \leq x \leq 1 \\ 1 & x > 1 \end{cases}$
Input	x_{\min} , minimum value of random variable; x_{\max} , maximum value of random variable
Output	$x \in [x_{\min}, x_{\max}]$
Mode	x_{\min} and x_{\max}
Median	$(x_{\min} + x_{\max})/2$
Mean	$(x_{\min} + x_{\max})/2$
Variance	$(x_{\max} - x_{\min})^2/8$
Regression Equation	$\sin^2(F_i\pi/2) = x_i/(x_{\max} - x_{\min}) - x_{\min}/(x_{\max} - x_{\min})$, where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \dots, N$.
Algorithm	1. Generate $U \sim U(0, 1)$. 2. Return $X = x_{\min} + (x_{\max} - x_{\min}) \sin^2(U\pi/2)$.

Source Code

```

1 double arcsine( double xMin, double xMax ) {
2
3     assert( xMin < xMax );
4
5     double q = sin( M_PI_2 * uniform( 0, 1 ) );
6     return xMin + ( xMax - xMin ) * q * q;
7 }

```

Notes

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

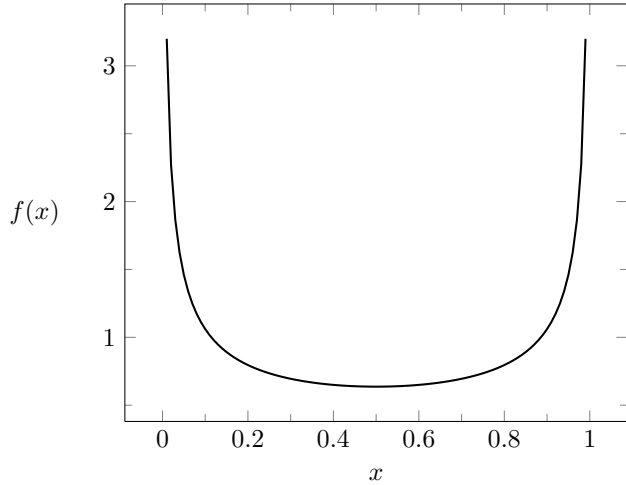


Figure 5. Plot of arcsine PDF

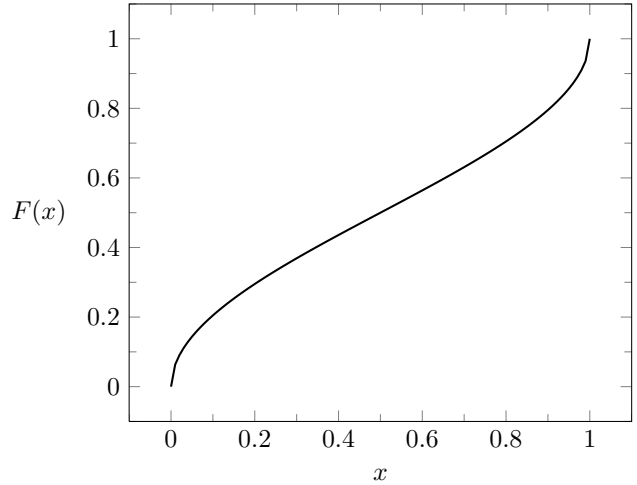


Figure 6. Plot of arcsine CDF

5.1.2 Beta

Density Function	$f(x) = \begin{cases} \frac{x^{v-1}(1-x)^{w-1}}{B(v,w)} & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$ <p>where $B(v, w)$ is the <i>beta function</i>, defined by $B(v, w) \equiv \int_0^1 t^{v-1}(1-t)^{w-1} dt$</p>
Distribution Function	$F(x) = \begin{cases} B_x(v, w)/B(v, w) & 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$ <p>where the <i>incomplete beta function</i> is defined by $B_x(v, w) \equiv \int_0^x t^{v-1}(1-t)^{w-1} dt$</p>
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	<ol style="list-style-type: none"> 1. Generate two IID gamma variates, $Y_1 \sim \text{gamma}(1, v)$ and $Y_2 \sim \text{gamma}(1, w)$. 2. Return $X = \begin{cases} x_{\min} + (x_{\max} - x_{\min})Y_1/(Y_1 + Y_2) & \text{if } v \geq w \\ x_{\min} - (x_{\max} - x_{\min})Y_2/(Y_1 + Y_2) & \text{if } v < w. \end{cases}$
Source Code	<pre> 1 double Random::beta(double v, double w, double xMin, double xMax) { 2 3 if (v < w) return xMax - (xMax - xMin) * beta(w, v); 4 double y1 = gamma(0., 1., v); 5 double y2 = gamma(0., 1., w); 6 return xMin + (xMax - xMin) * y1 / (y1 + y2); 7 } </pre>
Notes	<ol style="list-style-type: none"> 1. $X \sim B(v, w)$ if and only if $1 - X \sim B(w, v)$. 2. When $v = w = 1/2$, this reduces to the arcsine distribution. 3. When $v = w = 1$, this reduces to the uniform distribution.

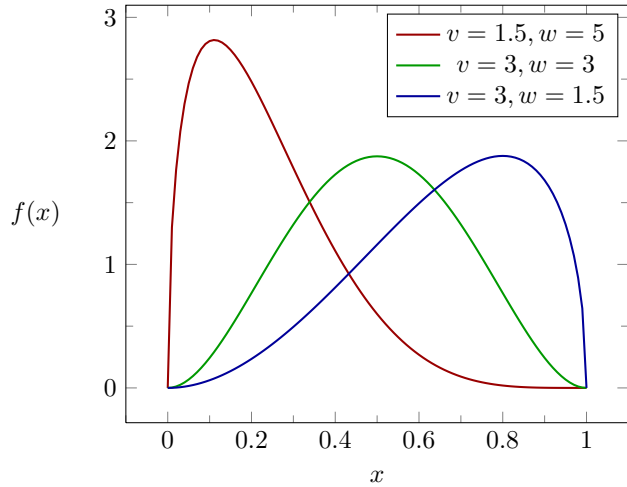


Figure 7. Plot of beta PDF

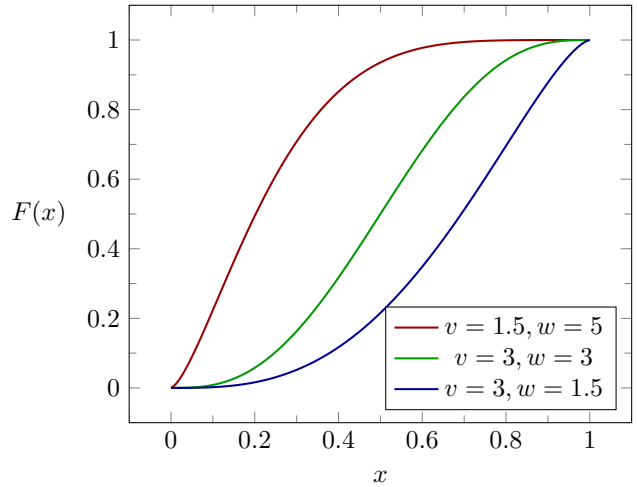


Figure 8. Plot of beta CDF

5.1.3 Cauchy (Lorentz)

Density Function	$f(x) = \frac{1}{\pi b} \left[1 + \left(\frac{x-a}{b} \right)^2 \right]^{-1} \quad -\infty < x < \infty$
Distribution Function	$F(x) = \frac{1}{2} + \frac{1}{\pi} \tan^{-1} \left(\frac{x-a}{b} \right) \quad -\infty < x < \infty$
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 $U \sim U(-1/2, 1/2)$. 2. Return $X = a + b \tan(\pi U)$.

Source Code

```

1 double cauchy( double a, double b ) {
2
3     assert( b > 0 );
4     return a + b * tan( M_PI * uniform( -0.5, 0.5 ) );
5 }

```

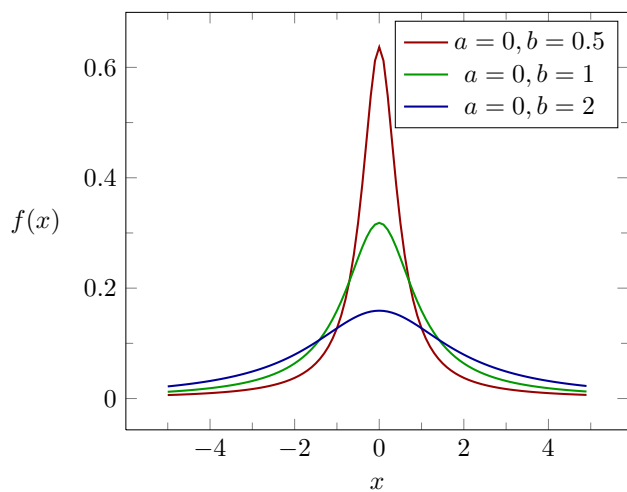


Figure 9. Plot of Cauchy PDF

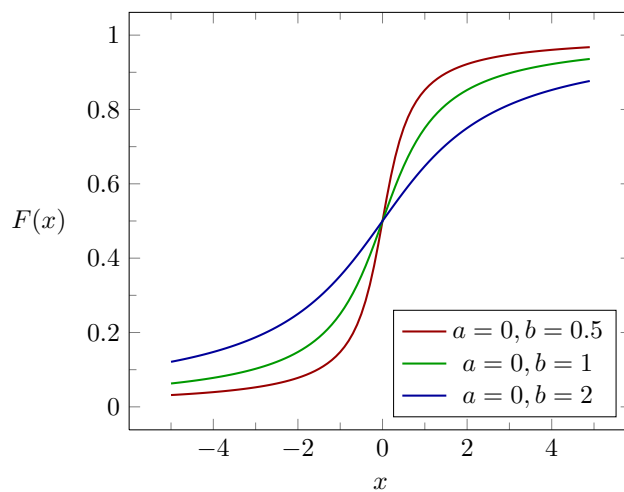


Figure 10. Plot of Cauchy CDF

5.1.4 Chi-Square

Density Function

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

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

Distribution Function

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

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

```
1 double Random::chiSquare( int df ) {
2
3     assert( df >= 1 );
4     return gamma( 0, 2, 0.5 * double( df ) );
5 }
```

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

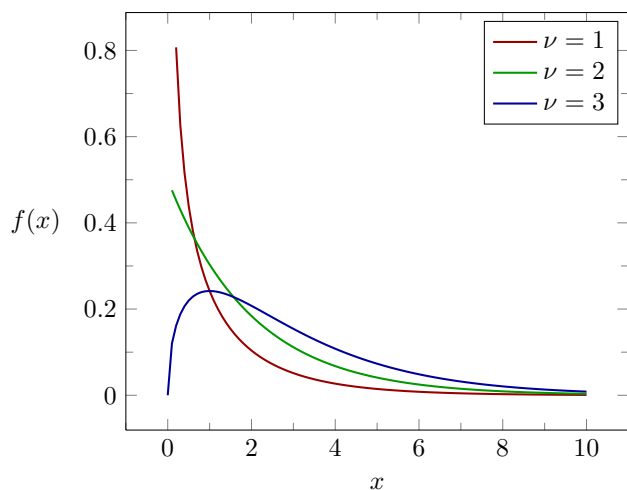


Figure 11. Plot of chi-square PDF

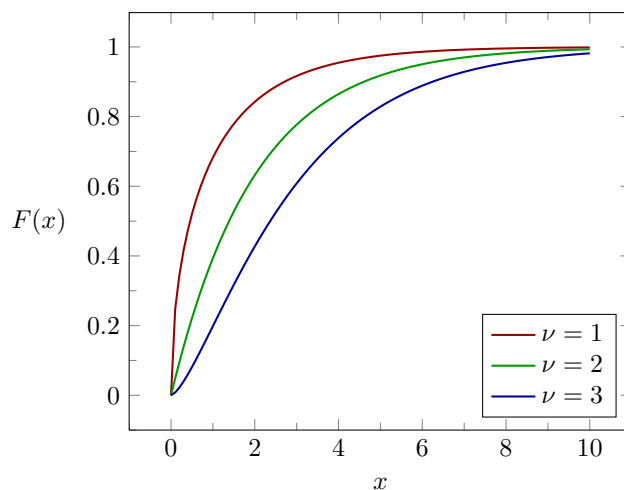


Figure 12. Plot of chi-square CDF

5.1.5 Cosine

Density Function	$f(x) = \begin{cases} \frac{1}{2b} \cos\left(\frac{x-a}{b}\right) & x_{\min} \leq x \leq x_{\max} \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} 0 & x < x_{\min} \\ \frac{1}{2} \left[1 + \sin\left(\frac{x-a}{b}\right) \right] & x_{\min} \leq x \leq x_{\max} \\ 1 & x > x_{\max} \end{cases}$
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 $i = 1, 2, \dots, N$.
Algorithm	1. Generate $U \sim U(-1, 1)$. 2. Return $X = a + b \sin^{-1} U$.

Source Code

```

1 double Random::cosine( double xMin, double xMax ) {
2
3     assert( xMin < xMax );
4     double a = 0.5 * ( xMin + xMax ); // location parameter
5     double b = ( xMax - xMin ) / M_PI; // scale parameter
6     return a + b * asin( uniform( -1, 1 ) );
7 }

```

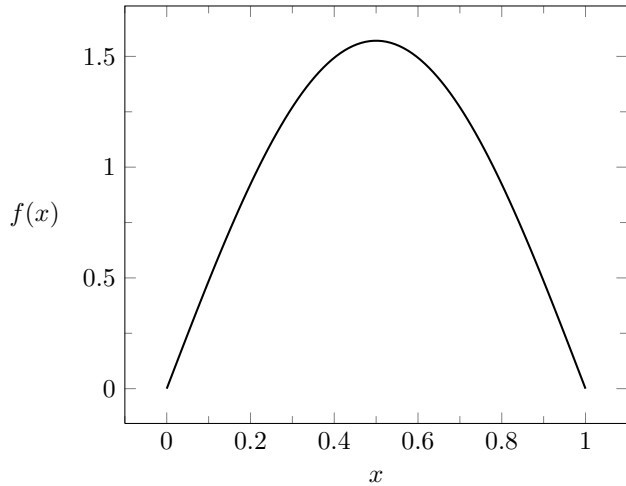


Figure 13. Plot of cosine PDF

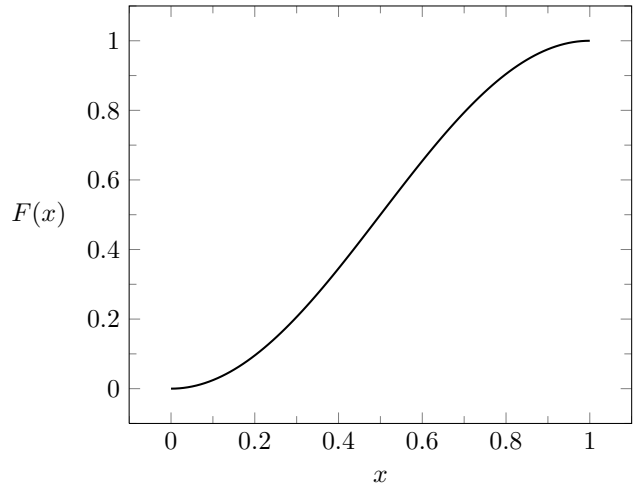


Figure 14. Plot of cosine CDF

5.1.6 Double Log

Density Function	$f(x) = \begin{cases} -\frac{1}{2b} \ln \left(\frac{ x-a }{b} \right) & x_{\min} \leq x \leq x_{\max} \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} \frac{1}{2} - \left(\frac{ x-a }{2b} \right) \left[1 - \ln \left(\frac{ x-a }{b} \right) \right] & x_{\min} \leq x \leq a \\ \frac{1}{2} + \left(\frac{ x-a }{2b} \right) \left[1 - \ln \left(\frac{ x-a }{b} \right) \right] & a \leq x \leq x_{\max} \end{cases}$
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 $\lim_{x \rightarrow a} f(x) = \infty$.)
Median	a
Mean	a
Variance	$(x_{\max} - x_{\min})^2/36$
Algorithm	Based on composition and convolution for the product of two uniform densities: 1. Generate two IID uniform variates, $U_i \sim U(0, 1), i = 1, 2$. 2. Generate a Bernoulli variate, $U \sim \text{Bernoulli}(0.5)$. 3. If $U = 1$, return $X = a + bU_1U_2$; else if $U = 0$, return $X = a - bU_1U_2$.

Source Code

```

1 double Random::doubleLog( double xMin, double xMax ) {
2
3     assert( xMin < xMax );
4     double a = 0.5 * ( xMin + xMax ); // location parameter
5     double b = 0.5 * ( xMax - xMin ); // scale parameter
6
7     if ( bernoulli( 0.5 ) ) return a + b * uniform() * uniform();
8     else return a - b * uniform() * uniform();
9 }

```

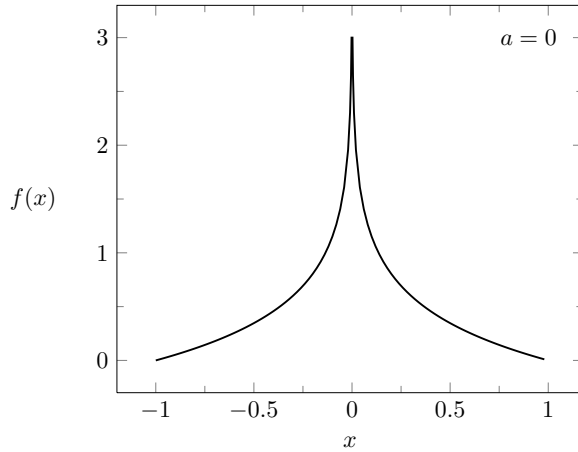


Figure 15. Plot of double log PDF

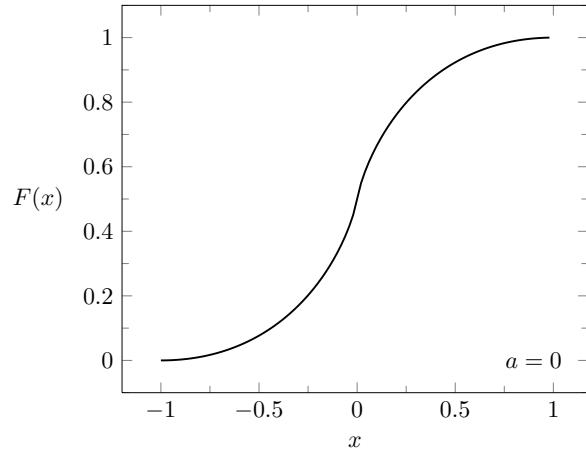


Figure 16. Plot of double log CDF

5.1.7 Erlang

Density Function	$f(x) = \begin{cases} \frac{(x/b)^{c-1} e^{-x/b}}{b(c-1)!} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} 1 - e^{-x/b} \sum_{i=0}^{c-1} \frac{(x/b)^i}{i!} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$
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	<p>This algorithm is based on the convolution formula.</p> <ol style="list-style-type: none"> 1. Generate c IID uniform variates, $U \sim U(0, 1), i = 1, \dots, c$. 2. Return $X = -b \sum_{i=1}^c \ln U_i = -b \ln \prod_{i=1}^c U_i$.

Source Code

```

1 double Random::erlang( double b, int c ) {
2
3     assert( b > 0. && c >= 1 );
4
5     double prod = 1;
6     for ( int i = 0; i < c; i++ ) prod *= uniform( 0, 1 );
7     return -b * log( prod );
8 }

```

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 $c = 1$).

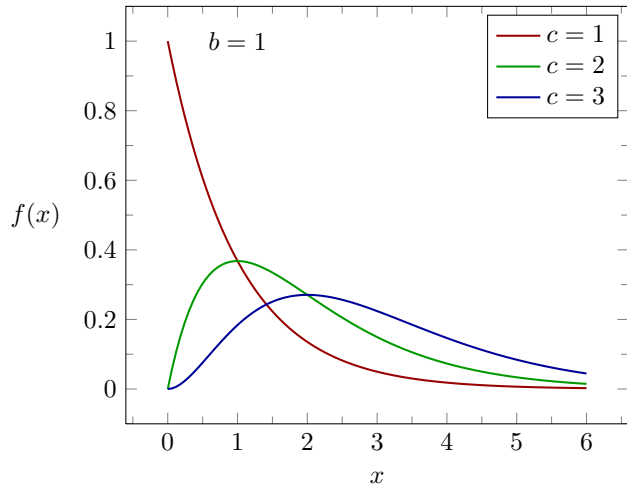


Figure 17. Plot of Erlang PDF

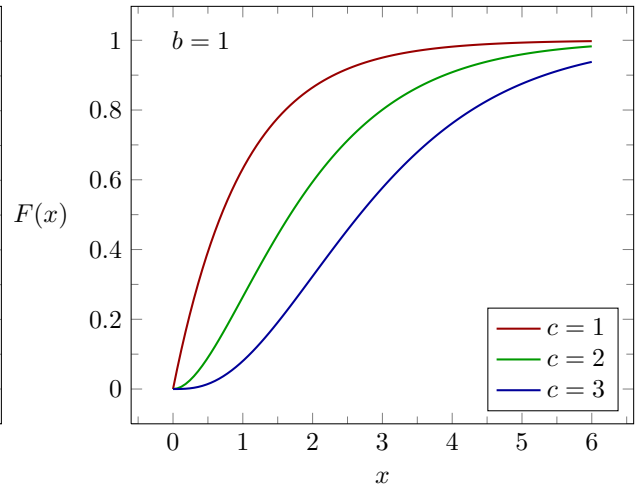


Figure 18. Plot of Erlang CDF

5.1.8 Exponential

Density Function	$f(x) = \begin{cases} \frac{1}{b} e^{-(x-a)/b} & x \geq a \\ 0 & \text{otherwise} \end{cases}$
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 $i = 1, 2, \dots, N$.
Maximum Likelihood	$b = \bar{X}$, the mean value of the random variates
Algorithm	1. Generate $U \sim U(0, 1)$. 2. Return $X = a - b \ln U$.

Source Code

```

1 double exponential( double a, double b ) {
2
3     assert( b > 0 );
4     return a - b * log( uniform( 0, 1 ) );
5 }

```

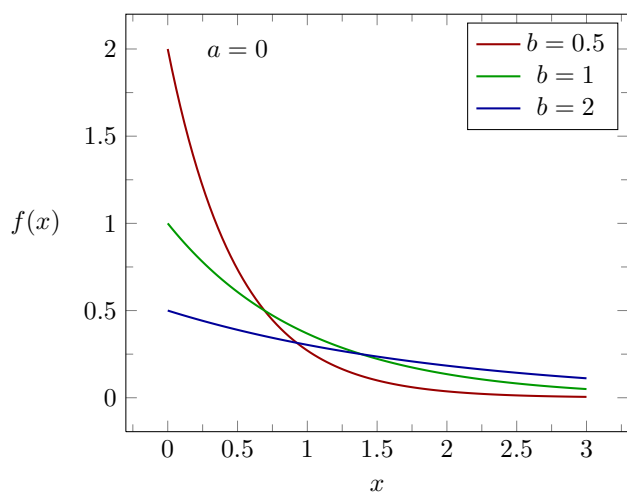


Figure 19. Plot of Exponential PDF

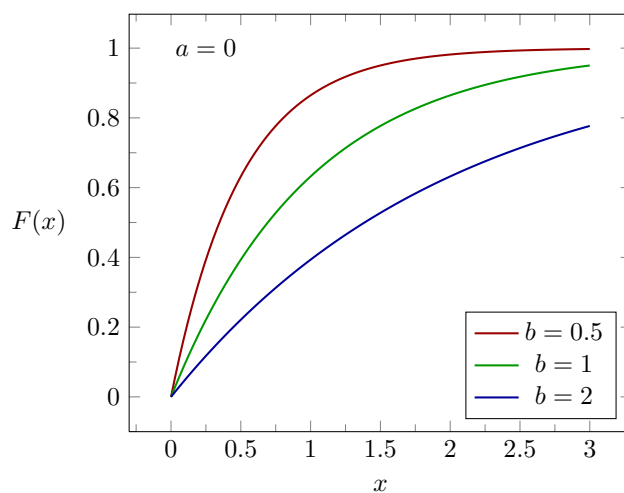


Figure 20. Plot of exponential CDF

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 $i = 1, 2, \dots, N$.
Algorithm	1. Generate $U \sim U(0, 1)$. 2. Return $X = a + b \ln(-\ln U)$.

Source Code

```

1 double extremeValue( double a, double b ) {
2
3     assert( b > 0 );
4
5     return a + b * log( -log( uniform( 0, 1 ) ) );
6 }

```

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., $X \rightarrow -(X - a)$.

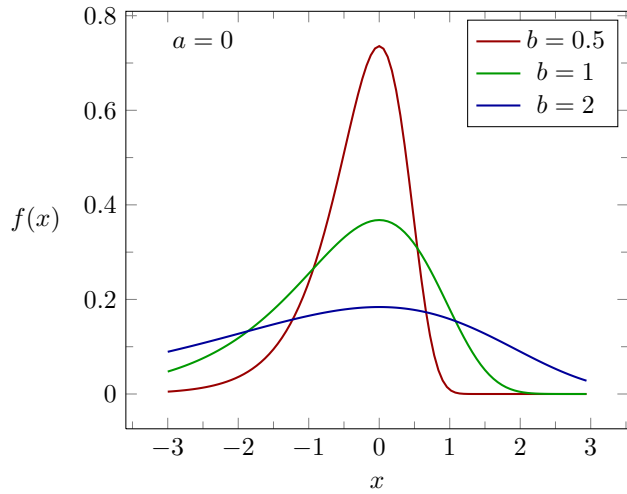


Figure 21. Plot of extreme value PDF

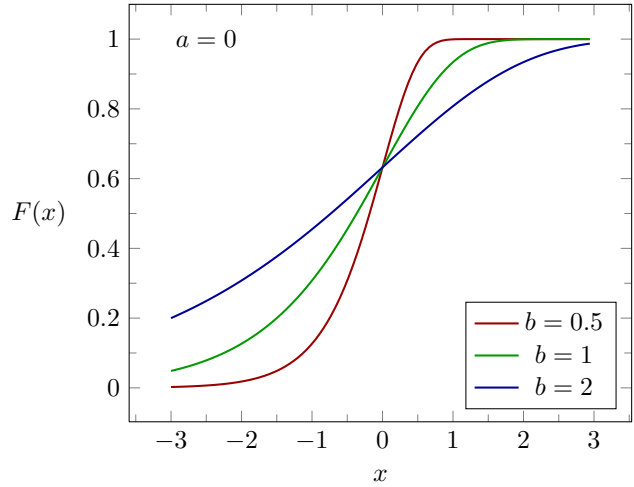


Figure 22. Plots of extreme value CDF

5.1.10 F Ratio

Density Function	$f(x) = \begin{cases} \frac{\Gamma[(v+w)/2]}{\Gamma(v/2)\Gamma(w/2)} \frac{(v/w)^{v/2} x^{(v-2)/2}}{(1+xv/w)^{(v+w)/2}} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$
	where $\Gamma(z)$ is the <i>gamma function</i> , 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	$\frac{w(v-2)}{v(w+2)}$ for $v > 2$
Mean	$\frac{w}{w-2}$ for $w > 2$
Variance	$\frac{2w^2(v+w-2)}{v(w-2)^2(w-4)}$ for $w > 4$
Algorithm	<ol style="list-style-type: none"> 1. Generate $V \sim \chi^2(v)$ and $W \sim \chi^2(w)$. 2. Return $X = \frac{V/v}{W/w}$.
Source Code	

```

1 double fRatio( int v, int w ) {
2
3     assert( v >= 1 && w >= 1 );
4     return ( chiSquare( v ) / v ) / ( chiSquare( w ) / w );
5 }
6

```

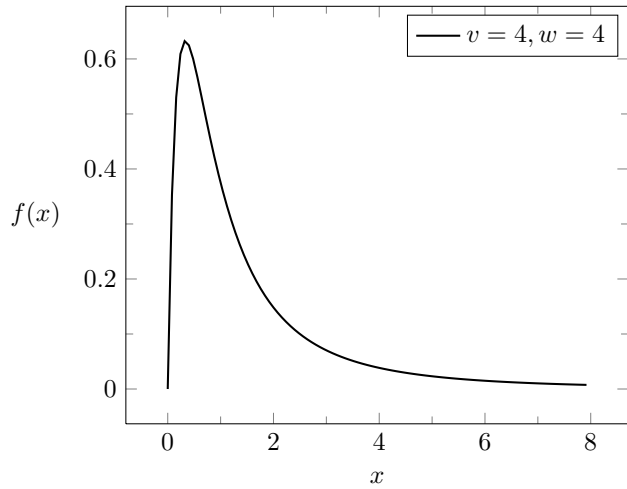


Figure 23. Plot of F Ratio DF

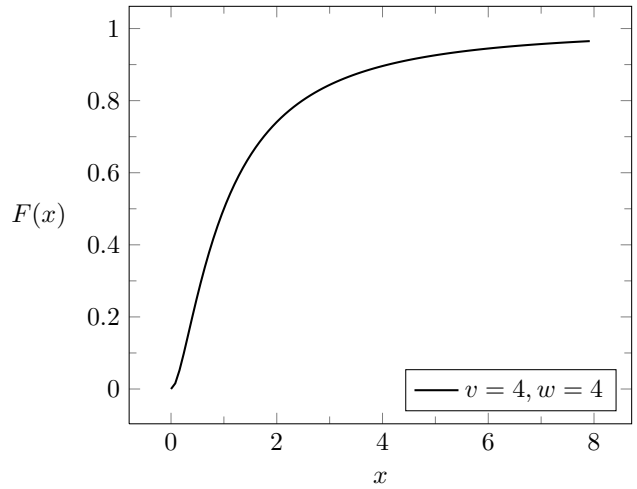


Figure 24. Plot of F Ratio CDF

5.1.11 Gamma

Density Function	$f(x) = \begin{cases} \frac{1}{\Gamma(c)} b^{-c} (x-a)^{c-1} e^{-(x-a)/b} & x > a \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	<p>where $\Gamma(z)$ is the <i>gamma function</i>, defined by $\Gamma(z) \equiv \int_0^\infty t^{z-1} e^{-t} dt$</p> <p>No closed form, in general. However, if c is a positive integer, then</p> $F(x) = \begin{cases} 1 - e^{-(x-a)/b} \sum_{k=0}^{c-1} \frac{1}{k!} \left(\frac{x-a}{b} \right)^k & x > a \\ 0 & \text{otherwise} \end{cases}$
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	$b^2 c$
Algorithm	<p>There are three algorithms (Law and Kelton, 1991), depending upon the value of the shape parameter c:</p> <p>Case 1: $c < 1$ Let $\beta = 1 + c/e$. 1. Generate $U_1 \sim U(0, 1)$ 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 $U_2 \sim U(0, 1)$. If $U_2 \leq e^{-Y}$, return $X = Y$; otherwise, go back to step 1. 3. Set $Y = -\ln[(\beta - P)/c]$ and generate $U_2 \sim U(0, 1)$. If $U_2 \leq Y^{c-1}$, return $X = Y$; otherwise, go back to step 1.</p> <p>Case 2: $c = 1$ Return $X \sim \text{exponential}(a, b)$.</p> <p>Case 3: $c > 1$ 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, $U_1 \sim U(0, 1)$ and $U_2 \sim U(0, 1)$. 2. Set $V = \alpha \ln[U_1/(1 - U_1)]$, $Y = ce^V$, $Z = U_1^2 U_2$, 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.</p>

Source Code

```

1 double gamma( double a, double b, double c ) {
2
3     assert( b > 0. && c > 0. );
4
5     static const double A = 1. / sqrt( 2. * c - 1. );
6     static const double B = c - log( 4. );
7     static const double Q = c + 1. / A;
8     static const double T = 4.5;
9     static const double D = 1. + log( T );
10    static const double C = 1. + c / M.E;
11
12    if ( c < 1. ) {
13        while ( true ) {
14            double p = C * _u();
15            if ( p > 1. ) {
16                double y = -log( ( C - p ) / c );
17                if ( _u() <= pow( y, c - 1. ) ) return a + b * y;
18            }
19            else {
20                double y = pow( p, 1. / c );
21                if ( _u() <= exp( -y ) ) return a + b * y;
22            }
23        }
24    }
25    else if ( c == 1.0 ) return exponential( a, b );
26    else {
27        while ( true ) {
28            double p1 = _u();
29            double p2 = _u();
30            double v = A * log( p1 / ( 1. - p1 ) );
31            double y = c * exp( v );
32            double z = p1 * p1 * p2;
33            double w = B + Q * v - y;
34            if ( w + D - T * z >= 0. || w >= log( z ) ) return a + b * y;
35        }
36    }
37 }

```

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.

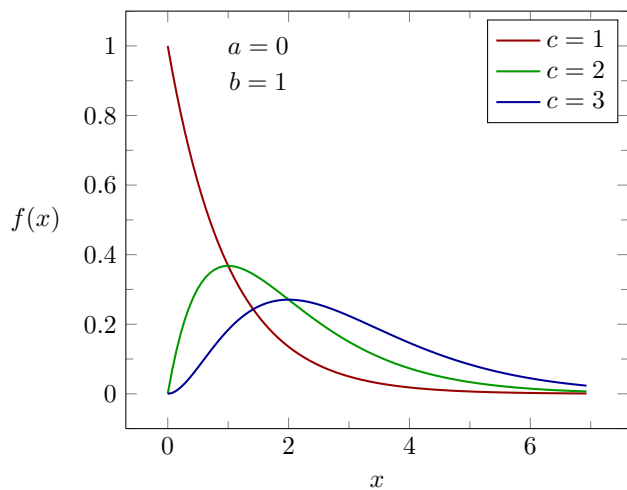


Figure 25. Plot of gamma PDF

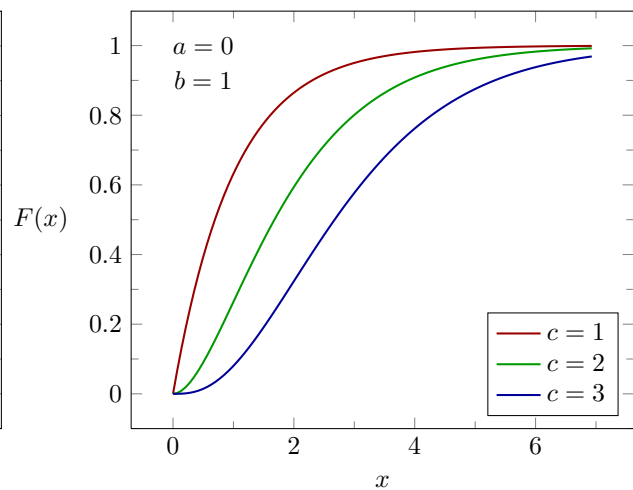


Figure 26. Plot of gamma CDF

5.1.12 Laplace (Double Exponential)

Density Function	$f(x) = \frac{1}{2b} \exp\left(-\frac{ x-a }{b}\right) \quad -\infty < x < \infty$
Distribution Function	$F(x) = \begin{cases} \frac{1}{2}e^{(x-a)/b} & x \leq a \\ 1 - \frac{1}{2}e^{-(x-a)/b} & x \geq a \end{cases}$
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	$\begin{cases} \ln(2F_i) = x_i/b - a/b & 0 \leq F_i \leq 1/2 \\ -\ln[2(1-F_i)] = x_i/b - a/b & 1/2 \leq F_i \leq 1 \end{cases}$ <p>where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \dots, N$.</p>
Algorithm	<ol style="list-style-type: none"> 1. Generate two IID random variates, $U_1 \sim U(0, 1)$ and $U_2 \sim U(0, 1)$. 2. Return $X = \begin{cases} a + b \ln U_2 & \text{if } U_1 \geq 1/2 \\ a - b \ln U_2 & \text{if } U_1 < 1/2 \end{cases}$

Source Code

```

1 double laplace( double a, double b ) {
2
3     assert( b > 0 );
4
5     // composition method
6     if ( bernoulli( 0.5 ) ) return a + b * log( uniform( 0, 1 ) );
7     else return a - b * log( uniform( 0, 1 ) );
8 }

```

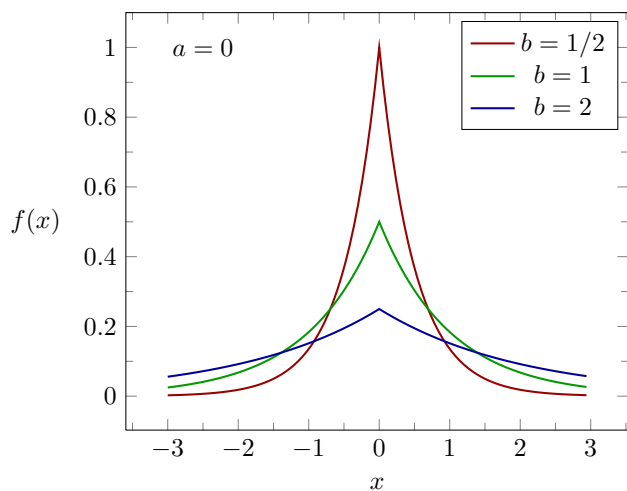


Figure 27. Plot of Laplace PDF

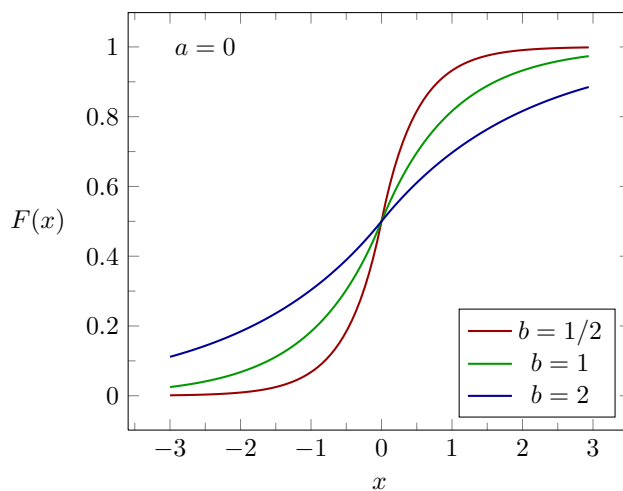


Figure 28. Plot of Laplace CDF

5.1.13 Logarithmic

Density Function	$f(x) = \begin{cases} -\frac{1}{b} \ln\left(\frac{x-a}{b}\right) & x_{\min} \leq x \leq x_{\max} \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} 0 & x < x_{\min} \\ \left(\frac{x-a}{b}\right) \left[1 - \ln\left(\frac{x-a}{b}\right)\right] & x_{\min} \leq x \leq x_{\max} \\ 1 & x > x_{\max} \end{cases}$
Input	x_{\min} , minimum value of random variable; x_{\max} , maximum value of random variable; location parameter $a = x_{\min}$; 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	$\frac{7}{144}(x_{\max} - x_{\min})^2$
Algorithm	Based on the convolution formula for the product of two uniform densities. 1. Generate two IID uniform variates, $U_1 \sim U(0, 1)$ and $U_2 \sim U(0, 1)$. 2. Return $X = a + bU_1U_2$.

Source Code

```

1 double logarithmic( double xMin, double xMax ) {
2
3     assert( xMin < xMax );
4
5     double a = xMin;           // location parameter
6     double b = xMax - xMin;    // scale parameter
7     return a + b * uniform( 0, 1 ) * uniform( 0, 1 );
8 }

```

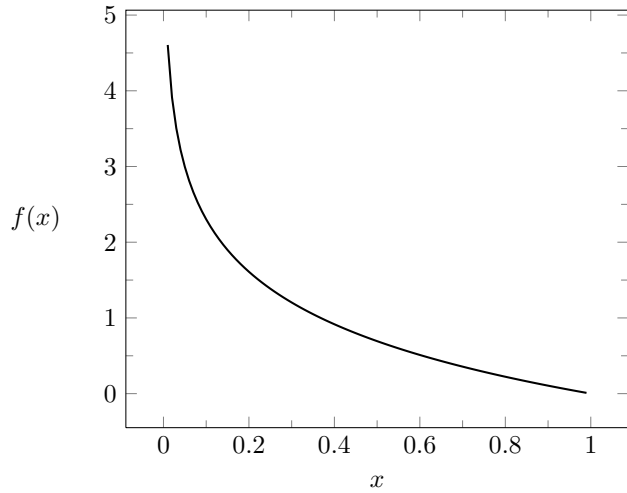


Figure 29. Plot of logarithmic PDF

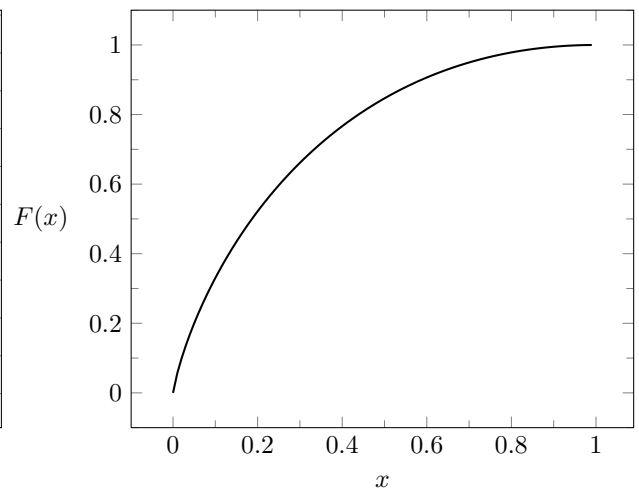


Figure 30. Plot of logarithmic CDF

5.1.14 Logistic

Density Function	$f(x) = \frac{1}{b} \frac{e^{(x-a)/b}}{[1 + e^{(x-a)/b}]^2} \quad -\infty < x < \infty$
Distribution Function	$F(x) = \frac{1}{1 + e^{-(x-a)/b}} \quad -\infty < x < \infty$
Input	Location parameter a , any real number; scale parameter $b > 0$
Output	$x \in (-\infty, \infty)$
Mode	a
Median	a
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 $i = 1, 2, \dots, N$.
Algorithm	1. Generate $U \sim U(0, 1)$. 2. Return $X = a - b \ln(U^{-1} - 1)$.

Source Code

```

1 double logistic( double a, double b ) {
2
3     assert( b > 0 );
4
5     return a - b * log( 1 / uniform( 0, 1 ) - 1 );
6 }

```

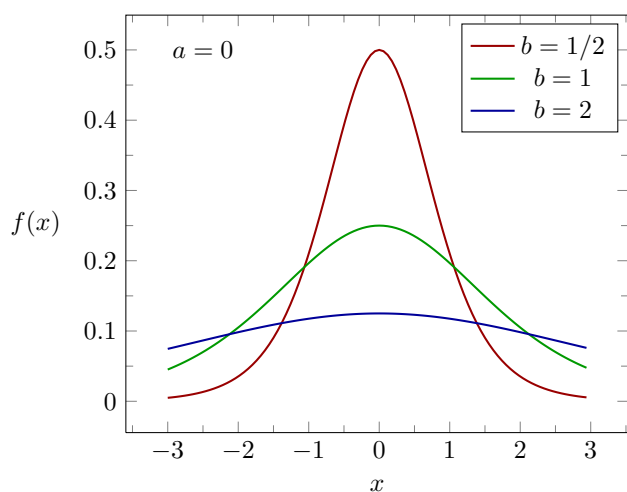


Figure 31. Plot of logistic PDF

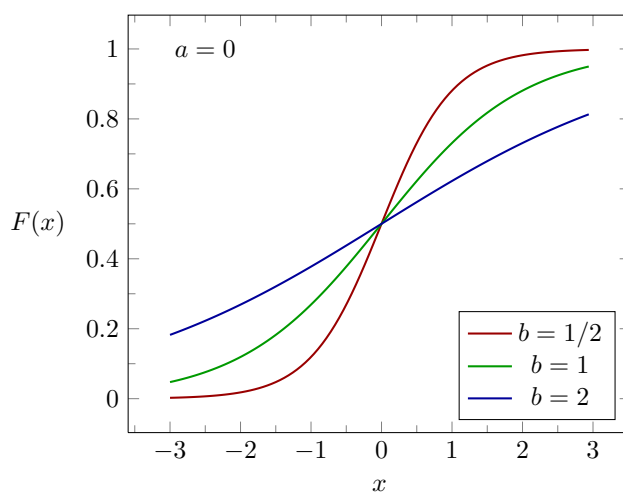


Figure 32. Plot of logistic CDF

5.1.15 Lognormal

Density Function	$f(x) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma(x-a)} \exp\left[-\frac{[\ln(x-a)-\mu]^2}{2\sigma^2}\right] & x > a \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} \frac{1}{2} \left\{ 1 + \operatorname{erf}\left[\frac{\ln(x-a)-\mu}{\sqrt{2}\sigma}\right] \right\} & x > a \\ 0 & \text{otherwise} \end{cases}$
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	$\mu = \frac{1}{N} \sum_{i=1}^N \ln x_i$ and $\sigma^2 = \frac{1}{N} \sum_{i=1}^N (\ln x_i - \mu)^2$
Algorithm	1. Generate $V \sim N(\mu, \sigma^2)$. 2. Return $X = a + e^V$.
Source Code	<pre> 1 double lognormal(double a, double mu, double sigma) { 2 3 return a + exp(normal(mu, sigma)); 4 } </pre>
Note	$X \sim \text{LN}(\mu, \sigma)$ if and only if $\ln X \sim N(\mu, \sigma^2)$, where N is the normal distribution.

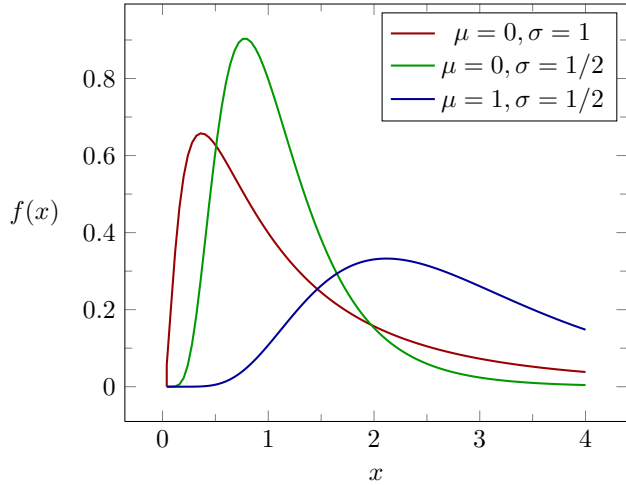


Figure 33. Plot of lognormal PDF

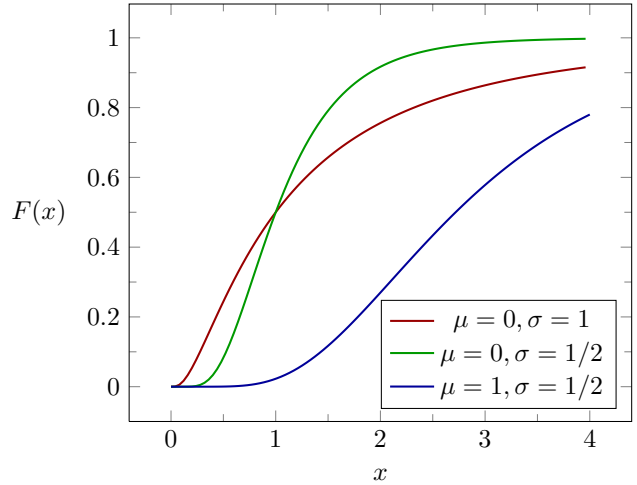


Figure 34. Plot of lognormal CDF

5.1.16 Normal (Gaussian)

Density Function	$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \quad -\infty < x < \infty$
Distribution Function	$F(x) = \frac{1}{2} \left[1 + \operatorname{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right) \right] \quad -\infty < x < \infty$
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 $i = 1, 2, \dots, N$.
Maximum Likelihood	$\mu = \frac{1}{N} \sum_{i=1}^N x_i$ and $\sigma^2 = \frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2$
Algorithm	<ol style="list-style-type: none"> 1. Independently generate $U_1 \sim U(0, 1)$ and $U_2 \sim U(0, 1)$. 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

```

1 double normal( double mu, double sigma ) {
2     assert( sigma > 0 );
3     static bool f = true;
4     static double p2, q;
5     double p1, p;
6     if ( f ) {
7         do { p1 = uniform( -1, 1 ); p2 = uniform( -1, 1 ); p = p1 * p1 + p2 * p2; } while ( p >= 1 );
8         f = false;
9         q = sqrt( -2 * log( p ) / p );
10        return mu + sigma * p1 * q;
11    }
12    f = true;
13    return mu + sigma * p2 * q;
14 }

```

Note If $X \sim N(\mu, \sigma)$, then $\exp(X) \sim \Lambda(\mu, \sigma^2)$, the [lognormal](#) distribution.

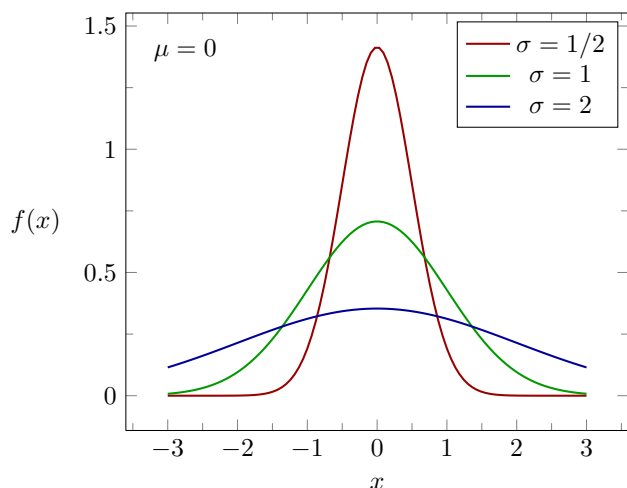


Figure 35. Plot of normal PDF

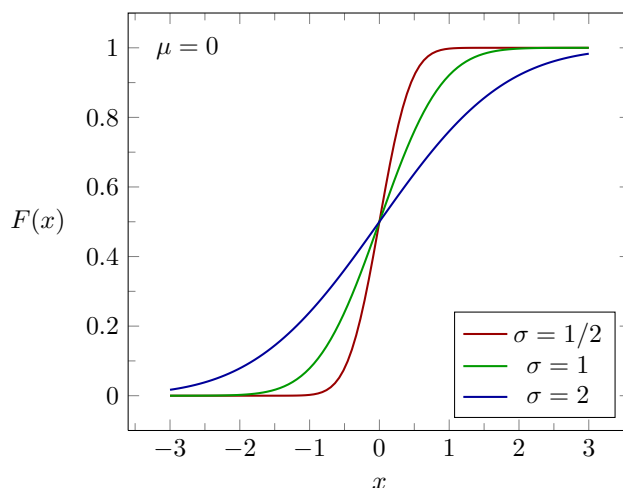


Figure 36. Plot of normal CDF

5.1.17 Parabolic

Density Function	$f(x) = \frac{3}{4b} \left[1 - \left(\frac{x-a}{b} \right)^2 \right] \quad x_{\min} \leq x \leq x_{\max}$
Distribution Function	$F(x) = \frac{(a+2b-x)(x-a+b)^2}{4b^3} \quad x_{\min} \leq x \leq x_{\max}$
Input	x_{\min} , minimum value of random variable; x_{\max} , maximum value of random variable;
Output	location parameter $a = (x_{\min} + x_{\max})/2$; scale parameter $b = (x_{\max} - x_{\min})/2$;
Mode	$x \in [x_{\min}, x_{\max})$
Median	$(x_{\min} + x_{\max})/2$
Mean	$(x_{\min} + x_{\max})/2$
Variance	$(x_{\min} - x_{\max})^2/20$
Algorithm	Uses the <i>acceptance-rejection method</i> on the above density function, $f(x)$.
Source Code	

```

1 static double parabola( double x, double xMin, double xMax ) // parabolic density function
2
3 if ( x < xMin || x > xMax ) return 0;
4
5 double a = 0.5 * ( xMin + xMax ); // location parameter
6 double b = 0.5 * ( xMax - xMin ); // scale parameter
7 double yMax = 0.75 / b;
8
9 return yMax * ( 1. - ( x - a ) * ( x - a ) / ( b * b ) );
10 }
11
12 double parabolic( double xMin, double xMax ) { // Parabolic distribution
13
14     assert( xMin < xMax );
15
16     double a = 0.5 * ( xMin + xMax ); // location parameter
17     double yMax = parabola( a, xMin, xMax ); // maximum function range
18
19     return userSpecified( parabola, xMin, xMax, 0, yMax );
20 }

```

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

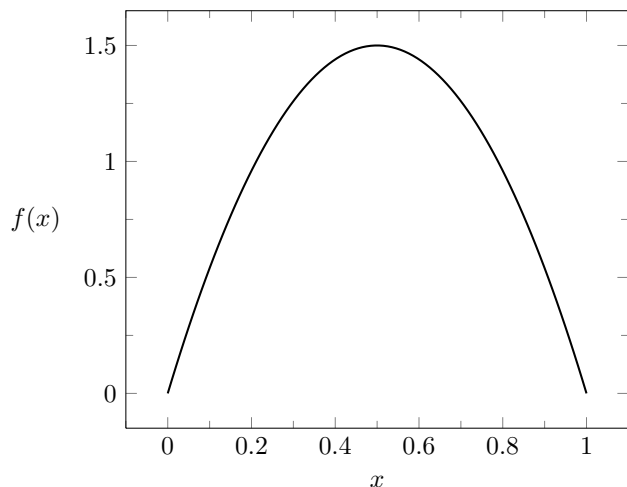


Figure 37. Plot of parabolic PDF

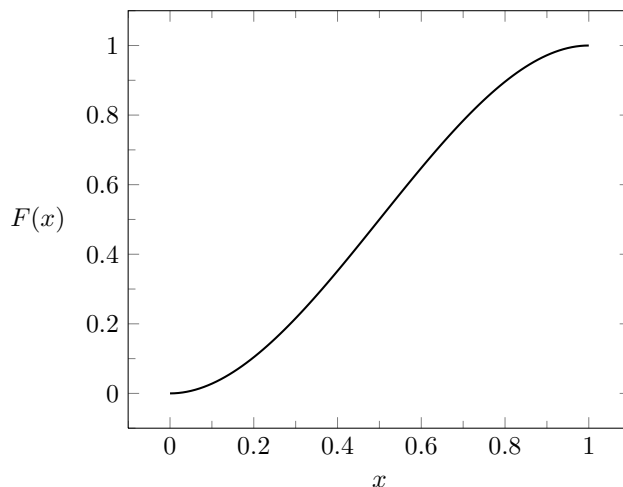


Figure 38. Plot of parabolic CDF

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 $i = 1, 2, \dots, N$.

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

Source Code

```
1 double pareto( double c ) {
2
3     assert( c > 0 );
4
5     return pow( uniform( 0, 1 ), -1 / c );
6 }
```

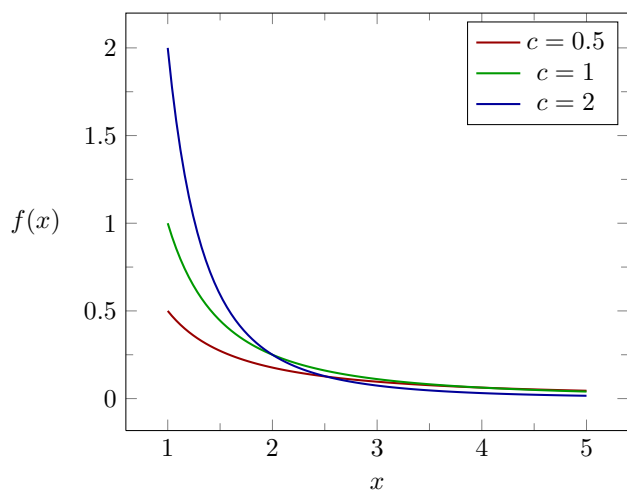


Figure 39. Plot of Pareto PDF

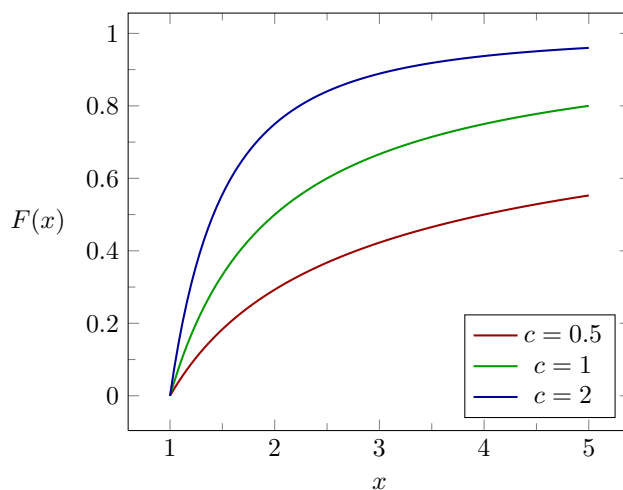


Figure 40. Plot of Pareto CDF

5.1.19 Pearson's Type 5 (Inverted Gamma)

Density Function

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

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

Distribution Function

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

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

Input Scale parameter, $b > 0$; shape 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

```
1 double pearson5( double c, double b ) {
2
3     assert( c > 0 && b > 0 );
4
5     return 1 / gamma( 0, c, 1 / b );
6 }
```

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*.

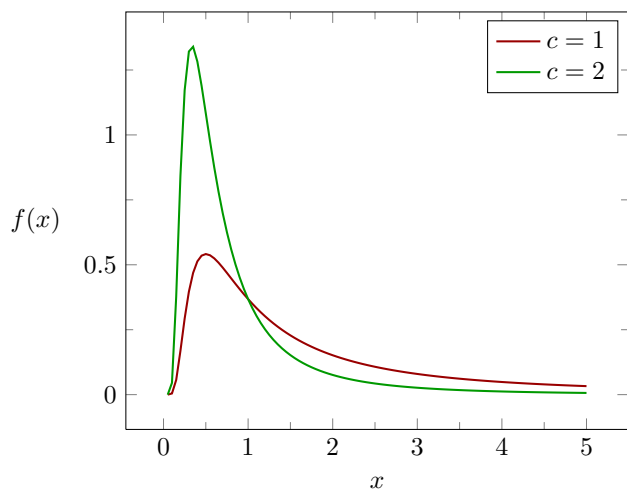


Figure 41. Plot of Pearson's type 5 PDF

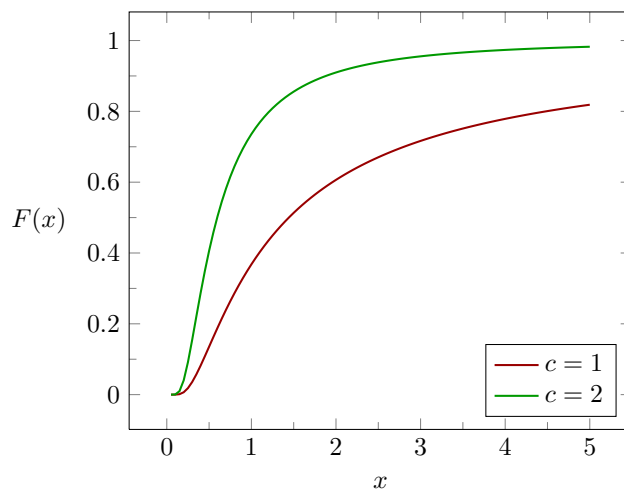


Figure 42. Plot of Pearson's type 5 CDF

5.1.20 Pearson's Type 6

Density Function	$f(x) = \begin{cases} \frac{(x/b)^{v-1}}{bB(v, w)[1 + (x/b)]^{v+w}} & x > 0 \\ 0 & \text{otherwise} \end{cases}$ <p>where $B(v, w)$ is the <i>Beta function</i>, defined by $B(v, w) \equiv \int_0^1 t^{v-1}(1-t)^{w-1}dt$</p>
Distribution Function	$F(x) = \begin{cases} F_B\left(\frac{x}{x+b}\right) & x > 0 \\ 0 & \text{otherwise} \end{cases}$ <p>where $F_B(x)$ is the distribution function of a $B(v, w)$ random variable.</p>
Input	Shape parameters $v > 0$ and $w > 0$ and scale parameter $b > 0$
Output	$x \in [0, \infty)$
Mode	$\begin{cases} \frac{b(v-1)}{(w+1)} & \text{if } v \geq 1 \\ 0 & \text{otherwise} \end{cases}$
Mean	$\frac{bv}{w-1}$ for $w > 1$
Variance	$\frac{b^2v(v+w-1)}{(w-1)^2(w-2)}$ for $w > 2$
Algorithm	<ol style="list-style-type: none"> 1. Generate $Y \sim \text{gamma}(0, v, b)$ and $Z \sim \text{gamma}(0, w, b)$. 2. Return $X = Y/Z$.
Source Code	<pre> 1 double pearson6(double a1, double a2, double b) { 2 3 assert(v > 0 && w > 0 && b > 0); 4 5 return gamma(0, v, b) / gamma(0, w, b); 6 } </pre>
Notes	$X \sim \text{PearsonType6}(v, w, 1)$ if and only if $X/(1+X) \sim \text{beta}(v, w)$.

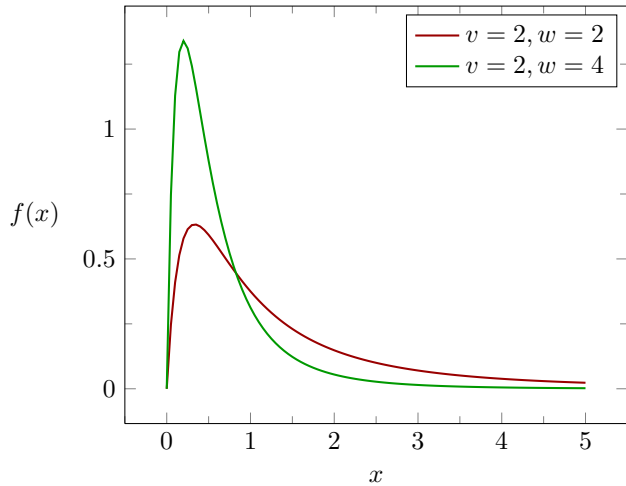


Figure 43. Plot of Pearson's type 6 PDF

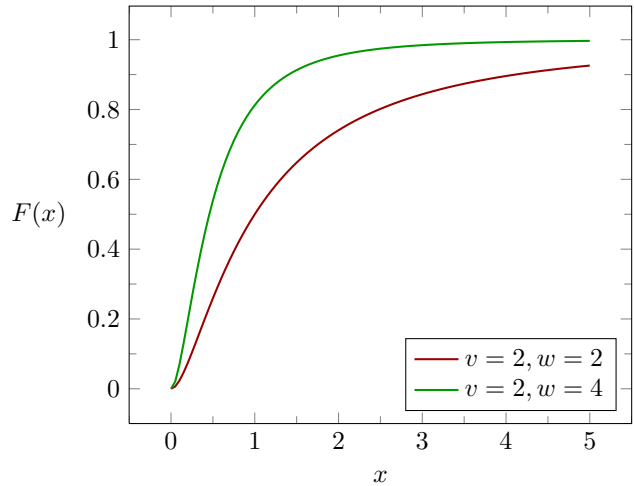


Figure 44. Plot of Pearson's type 6 CDF

5.1.21 Power

Density Function	$f(x) = cx^{c-1} \quad 0 \leq x \leq 1$
Distribution Function	$F(x) = x^c \quad 0 \leq x \leq 1$
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	$\frac{c}{c+1}$
Variance	$\frac{c}{(c+1)^2(c+2)}$
Regression Equation	$\ln F_i = c \ln x_i$, where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \dots, N$.
Algorithm	<ol style="list-style-type: none"> 1. Generate $U \sim U(0, 1)$. 2. Return $X = U^{1/c}$.

Source Code

```

1 double power( double c ) {
2
3     assert( c > 0 );
4
5     return pow( uniform( 0, 1 ), 1 / c );
6 }

```

Notes

This reduces to the [uniform](#) distribution when $c = 1$.

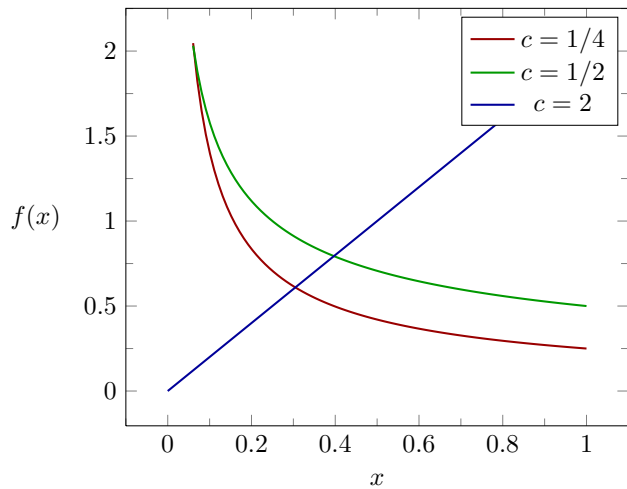


Figure 45. Plot of Power PDF

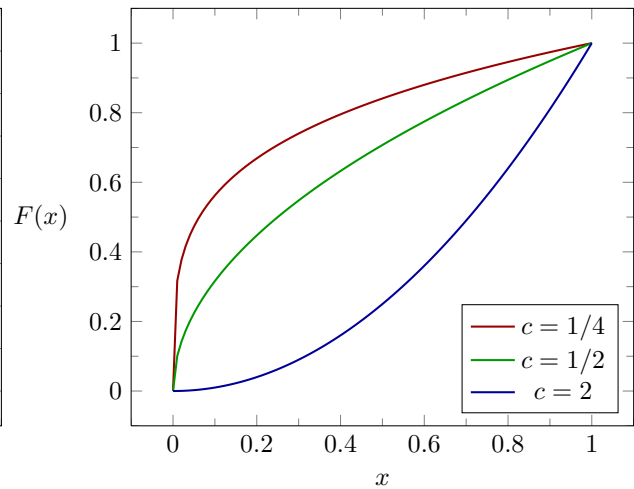


Figure 46. Plot of Power CDF

5.1.22 Raab-Green

Density Function

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

Distribution Function

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

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 $b = (x_{\max} - x_{\min})/2\pi$.
 $x \in [-x_{\min}, x_{\max})$

Output

a

Mode

a

Median

a

Mean

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

Variance

Algorithm

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

Source Code

```

1 double raab_green( void ) {
2
3     const double x = uniform( -M_PI, M_PI );
4     const double y = uniform( 0, 2 );
5     double w = 0, v = 1;
6     int n = 0;
7
8     while ( true ) {
9
10        n++;
11        v *= x * x / ( ( 2 * n ) * ( 2 * n - 1 ) );
12        w += v;
13        if ( y >= w ) return x;
14
15        n++;
16        v *= x * x / ( ( 2 * n ) * ( 2 * n - 1 ) );
17        w -= v;
18        if ( y <= w ) return M_PI * sgn( x ) - x;
19    }
20    double raab_green( double xMin, double xMax ) {
21
22        assert( xMin < xMax );
23
24        double a = ( xMin + xMax ) / 2.;
25        double b = ( xMax - xMin ) / ( 2. * M_PI ); // location parameter
26        return a + b * raab_green(); // scale parameter
27    }

```

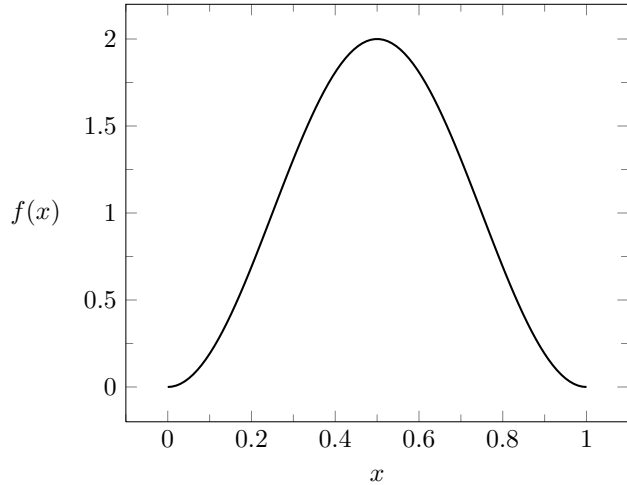


Figure 47. Plot of Raab-Green PDF

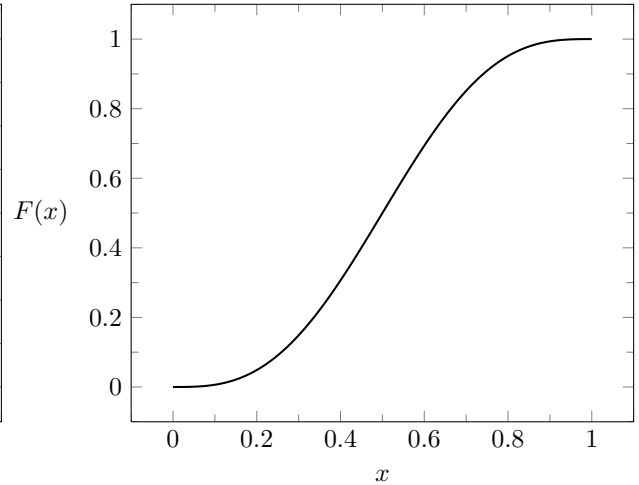


Figure 48. Plot of Raab-Green CDF

5.1.23 Rayleigh

Density Function	$f(x) = \begin{cases} \frac{2}{x-a} \left(\frac{x-a}{b} \right)^2 \exp \left[- \left(\frac{x-a}{b} \right)^2 \right] & x > a \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} 1 - \exp \left[- \left(\frac{x-a}{b} \right)^2 \right] & x > a \\ 0 & \text{otherwise} \end{cases}$
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 $U \sim U(0, 1)$. 2. Return $X = a + b\sqrt{-\ln U}$.
Source Code	<pre> 1 double rayleigh(double a, double b) { 2 3 assert(b > 0); 4 5 return a + b * sqrt(-log(uniform(0, 1))); 6 } </pre>

Notes Rayleigh is a special case of the [Weibull](#) when the shape parameter $c = 2$.

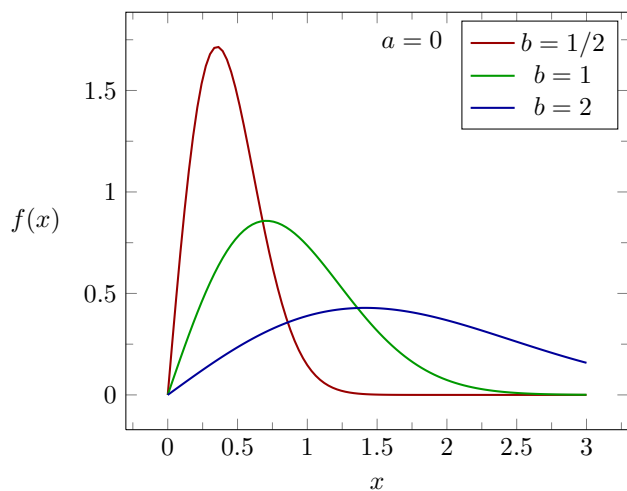


Figure 49. Plot of Rayleigh PDF

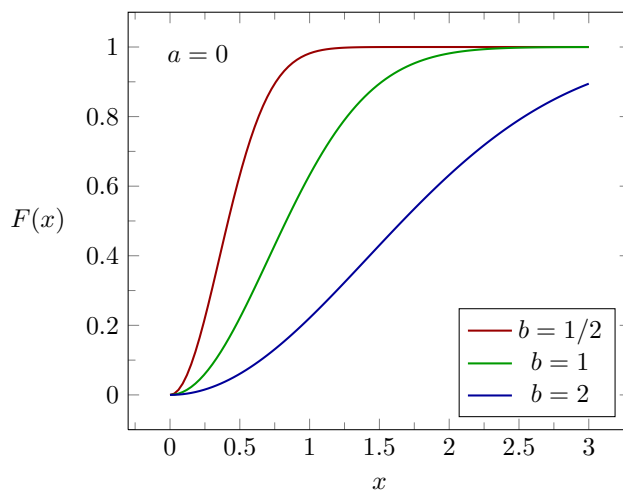


Figure 50. Plot of Rayleigh CDF

5.1.24 Student's t

Density Function	$f(x) = \frac{\Gamma[(\nu+1)/2]}{\sqrt{\pi\nu}\Gamma(\nu/2)} \left(1 + \frac{x^2}{\nu}\right)^{-(\nu+1)/2} \quad -\infty < x < \infty$ <p>where $\Gamma(z)$ is the <i>gamma function</i>, defined by $\Gamma(z) \equiv \int_0^\infty t^{z-1} e^{-t} dt$</p>
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	<ol style="list-style-type: none"> 1. Generate $Y \sim N(0, 1)$ and $Z \sim \chi^2(\nu)$. 2. Return $X = Y/\sqrt{Z/\nu}$.

Source Code

```

1 double studentT( int df ) {
2
3     assert( df >= 1 );
4
5     return normal( 0, 1 ) / sqrt( chiSquare( df ) / df );
6 }

```

Notes

For $\nu \geq 30$, this distribution can be approximated with the unit [normal](#) distribution.

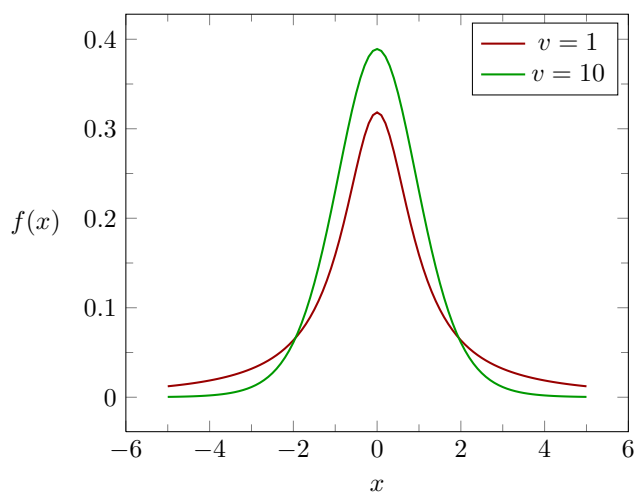


Figure 51. Plot of Student's t PDF

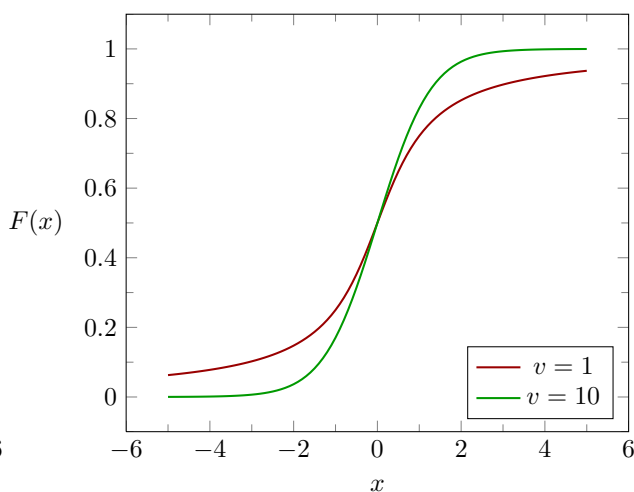


Figure 52. Plot of Student's t CDF

5.1.25 Triangular

Density Function	$f(x) = \begin{cases} \frac{2(x - x_{\min})}{(x_{\max} - x_{\min})(c - x_{\min})} & x_{\min} \leq x \leq c \\ \frac{2(x_{\max} - x)}{(x_{\max} - x_{\min})(x_{\max} - c)} & c \leq x \leq x_{\max} \end{cases}$
Distribution Function	$F(x) = \begin{cases} \frac{(x - x_{\min})^2}{(x_{\max} - x_{\min})(c - x_{\min})} & x_{\min} \leq x \leq c \\ 1 - \frac{(x_{\max} - x)^2}{(x_{\max} - x_{\min})(x_{\max} - c)} & c \leq x \leq x_{\max} \end{cases}$
Input	x_{\min} , minimum value of random variable; x_{\max} , maximum value of random variable; c , location of mode
Output	$x \in [x_{\min}, x_{\max})$
Mode	c
Median	$\begin{cases} x_{\min} + \sqrt{(x_{\max} - x_{\min})(c - x_{\min})/2} & \text{if } c \geq (x_{\min} + x_{\max})/2 \\ x_{\max} - \sqrt{(x_{\max} - x_{\min})(x_{\max} - c)/2} & \text{if } c \leq (x_{\min} + x_{\max})/2 \end{cases}$
Mean	$(x_{\min} + x_{\max} + c)/3$
Variance	$[3(x_{\max} - x_{\min})^2 + (x_{\min} + x_{\max} - 2c)^2]/72$
Algorithm	1. Generate $U \sim U(0, 1)$. 2. Return $X = \begin{cases} x_{\min} + \sqrt{(x_{\max} - x_{\min})(c - x_{\min})U} & U \leq (c - x_{\min})/(x_{\max} - x_{\min}) \\ x_{\max} - \sqrt{(x_{\max} - x_{\min})(x_{\max} - c)(1 - U)} & U > (c - x_{\min})/(x_{\max} - x_{\min}) \end{cases}$

Source Code

```

1 double triangular( double xMin, double xMax, double c ) {
2
3     assert( xMin < xMax && xMin <= c && c <= xMax );
4
5     double p = uniform( 0, 1 ), q = 1 - p;
6     if ( p <= ( c - xMin ) / ( xMax - xMin ) )
7         return xMin + sqrt( ( xMax - xMin ) * ( c - xMin ) * p );
8     else
9         return xMax - sqrt( ( xMax - xMin ) * ( xMax - c ) * q );
10 }

```

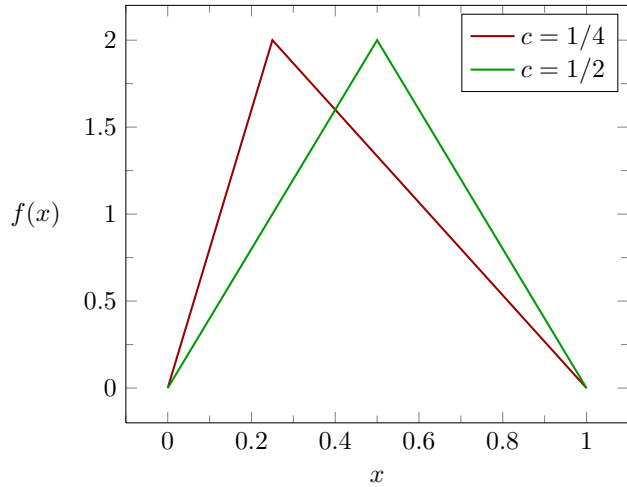


Figure 53. Plot of triangular PDF

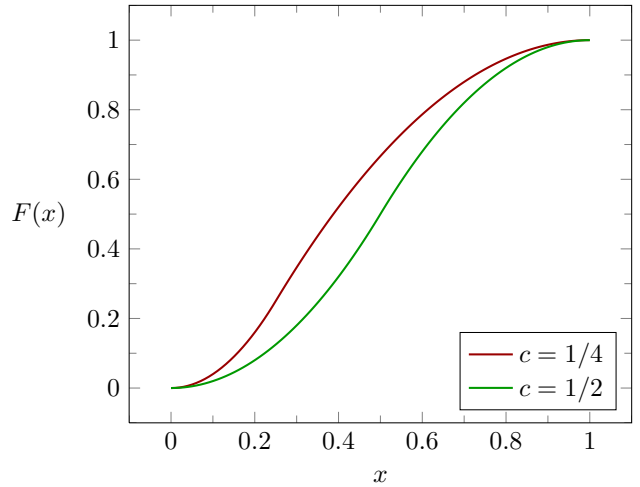


Figure 54. Plot of triangular CDF

5.1.26 Uniform

Density Function	$f(x) = \begin{cases} \frac{1}{x_{\max} - x_{\min}} & x_{\min} \leq x \leq x_{\max} \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} 0 & x < x_{\min} \\ \frac{x - x_{\min}}{x_{\max} - x_{\min}} & x_{\min} \leq x \leq x_{\max} \\ 1 & x > x_{\max} \end{cases}$
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	$(x_{\max} - x_{\min})^2/12$
Algorithm	<ol style="list-style-type: none"> 1. Generate $U \sim U(0, 1)$. 2. Return $X = x_{\min} + (x_{\max} - x_{\min})U$.
Source Code	<pre> 1 double uniform(double xMin, double xMax) { 2 3 assert(xMin < xMax); 4 5 return xMin + (xMax - xMin) * _u01(); 6 } </pre>
Notes	<ol style="list-style-type: none"> 1. The source code for <code>_u01()</code> 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 <code>beta</code> distribution when $v = w = 1$.

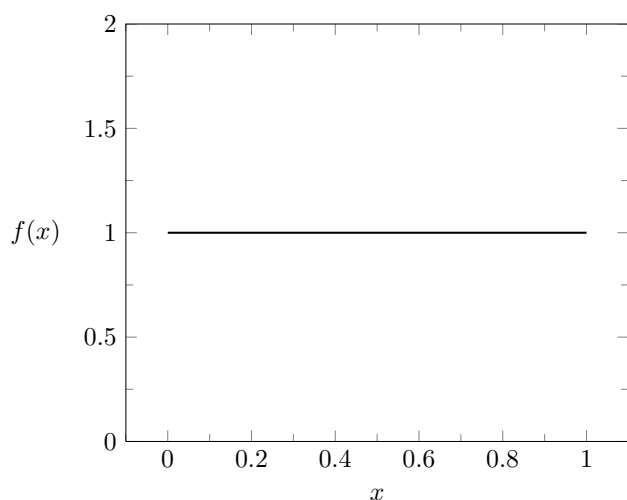


Figure 55. Plot of uniform PDF

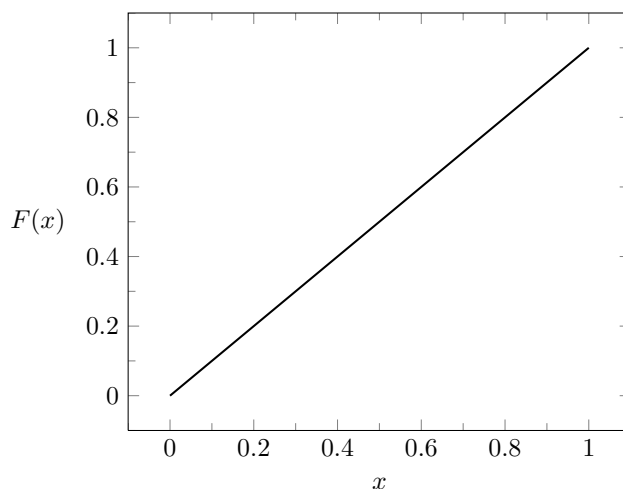


Figure 56. Plot of uniform CDF

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	<ol style="list-style-type: none"> 1. Generate $A \sim U(0, A_{\max})$ and $Y \sim U(y_{\min}, y_{\max})$, 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

```

1 double
2 Random::userSpecified( double( *usf )( double, // function
3 double, // xMin
4 double ), // xMax
5 double xMin, double xMax, // domain
6 double yMin, double yMax ) { // range
7
8     assert( xMin < xMax && yMin < yMax );
9     double x, y, areaMax = ( xMax - xMin ) * ( yMax - yMin );
10
11     // acceptance-rejection method
12     do {
13         x = uniform( 0.0, areaMax ) / ( yMax - yMin ) + xMin;
14         y = uniform( yMin, yMax );
15     } while ( y > usf( x, xMin, xMax ) );
16     return x;
17 }
18

```

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.

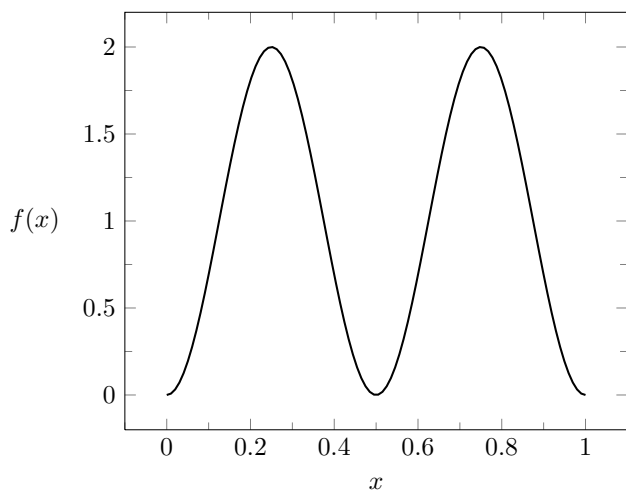


Figure 57. Plot of user-specified PDF

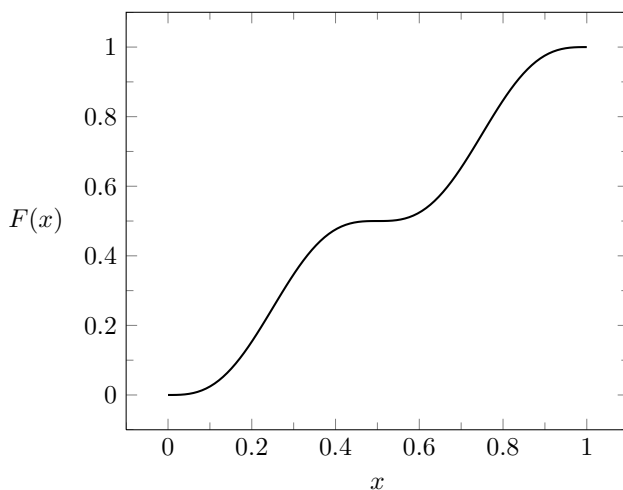


Figure 58. Plot of user-specified CDF

5.1.28 Weibull

Density Function	$f(x) = \begin{cases} \frac{c}{x-a} \left(\frac{x-a}{b} \right)^c \exp \left[- \left(\frac{x-a}{b} \right)^c \right] & x > a \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(x) = \begin{cases} 1 - \exp \left[- \left(\frac{x-a}{b} \right)^c \right] & x > a \\ 0 & \text{otherwise} \end{cases}$
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 $i = 1, 2, \dots, N$.
Algorithm	1. Generate $U \sim U(0, 1)$. 2. Return $X = a + b(-\ln U)^{1/c}$.
Source Code	<pre> 1 double weibull(double a, double b, double c) { 2 3 assert(b > 0 && c > 0); 4 5 return a + b * pow(-log(uniform(0, 1)), 1 / c); 6 } </pre>
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.

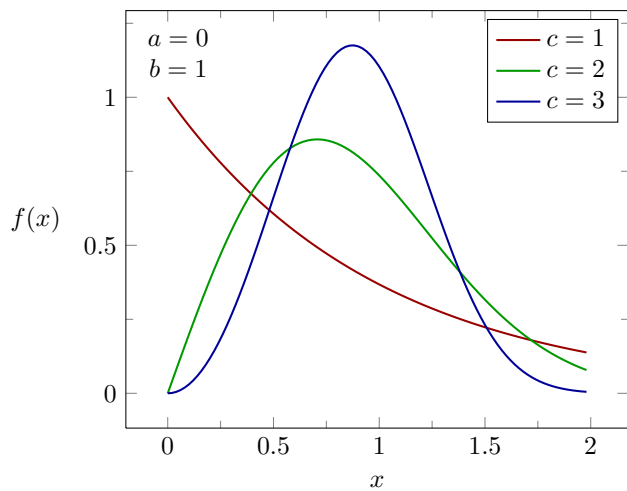


Figure 59. Plot of Weibull PDF

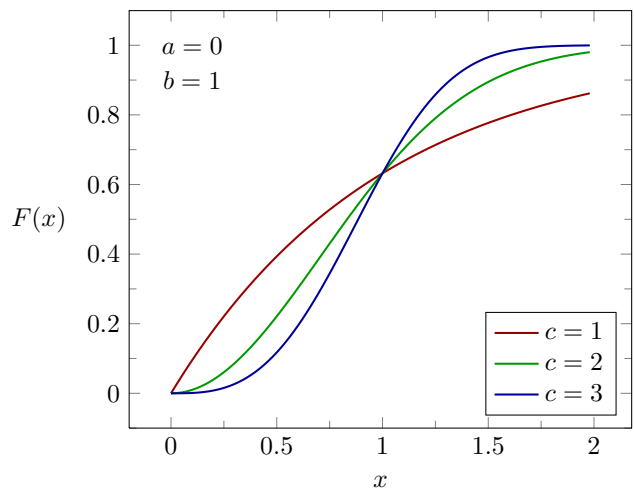


Figure 60. Plot of Weibull CDF

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, \dots, 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 $F(k) = \begin{cases} 1 - p & \text{if } 0 \leq k < 1 \\ 1 & \text{if } k \geq 1 \end{cases}$

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

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 $U \sim U(0, 1)$.
2. Return $X = \begin{cases} 1 & \text{if } U < p \\ 0 & \text{if } U \geq p \end{cases}$

Source Code

```
1 bool bernoulli( double p ) {  
2  
3     assert( 0 <= p && p <= 1 );  
4  
5     return uniform( 0, 1 ) < p;  
6 }
```

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	$f(k) = \begin{cases} \binom{n}{k} p^k (1-p)^{n-k} & k \in \{0, 1, \dots, n\} \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(k) = \begin{cases} \sum_{i=0}^k \binom{n}{i} p^i (1-p)^{n-i} & \text{if } 0 \leq k \leq n \\ 1 & \text{if } k > n \end{cases}$
	where the <i>binomial coefficient</i> $\binom{n}{k} \equiv \frac{n!}{k!(n-k)!}$.
Input	Probability of event, p , where $0 \leq p \leq 1$ and number of trials, $n \geq 1$
Output	The number of successes $k \in \{0, 1, \dots, n\}$
Mode	The integer k that satisfies $p(n+1) - 1 \leq k \leq p(n+1)$
Mean	np
Variance	$np(1-p)$
Maximum Likelihood	$p = \bar{X}/n$, where \bar{X} is the mean value of the random variates.
Algorithm	<ol style="list-style-type: none"> 1. Generate n IID Bernoulli trials, $X_i \sim \text{bernoulli}(p)$, where $i = 1, \dots, n$. 2. Return $X = X_1 + \dots + X_n$.

Source Code

```

1 int binomial( int n, double p ) {
2
3     assert( 0 <= p && p <= 1 && n >= 1 );
4
5     int sum = 0;
6     for ( int i = 0; i < n; i++ ) sum += bernoulli( p );
7     return sum;
8 }

```

Notes

1. The binomial reduces to the [bernoulli](#) when $n = 1$.
2. [Poisson](#)(np) approximates $\text{binomial}(n, p)$ 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 $0.1 \leq p \leq 0.9$ —and for all values of p when $np > 25$.

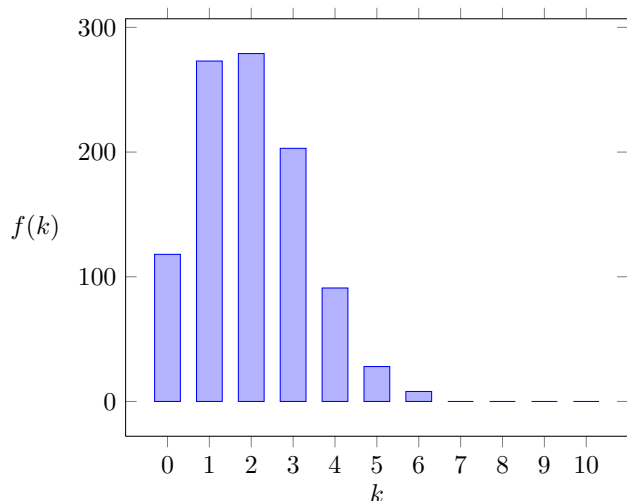


Figure 61. Histogram of binomial PDF

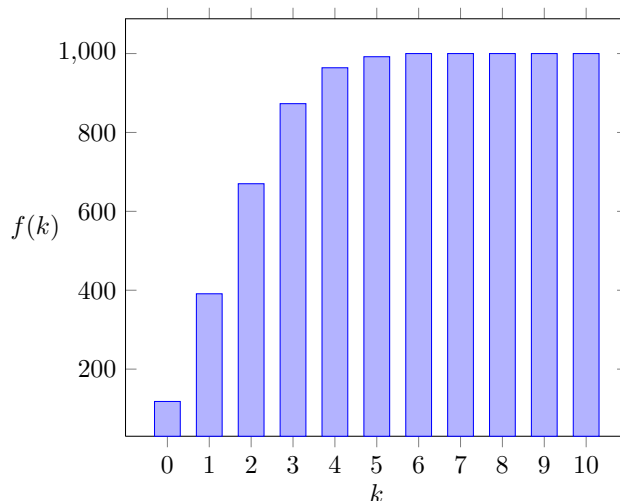


Figure 62. Histogram of binomial CDF

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	$f(k) = \begin{cases} p(1-p)^k & k \in \{0, 1, \dots\} \\ 0 & \text{otherwise} \end{cases}$
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 $k \in \{0, 1, \dots\}$
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 $U \sim U(0, 1)$. 2. Return $X = \text{int}(\ln U / (\ln(1-p)))$.

Source Code

```
1 int geometric( double p ) {
2     assert( 0 < p && p < 1 );
3     return int( log( uniform( 0, 1 ) ) / log( 1 - p ) );
4 }
```

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, \dots is a sequence of independent [Bernoulli](#)(p) random variates and $X = \min\{i \mid X_i = 1\} - 1$, then $X \sim \text{geometric}(p)$.

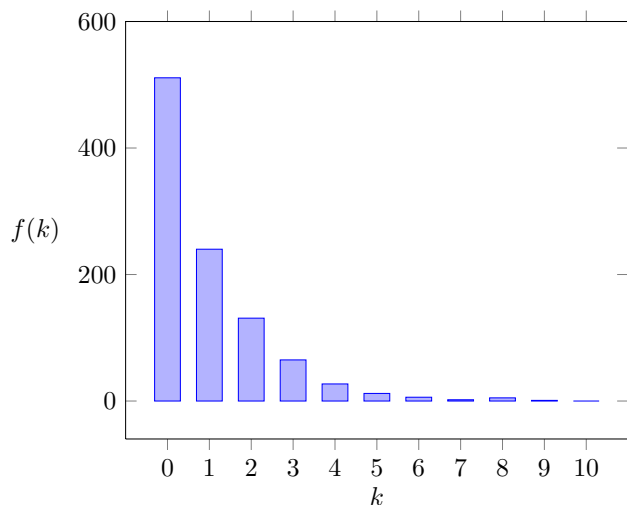


Figure 63. Histogram of geometric PDF

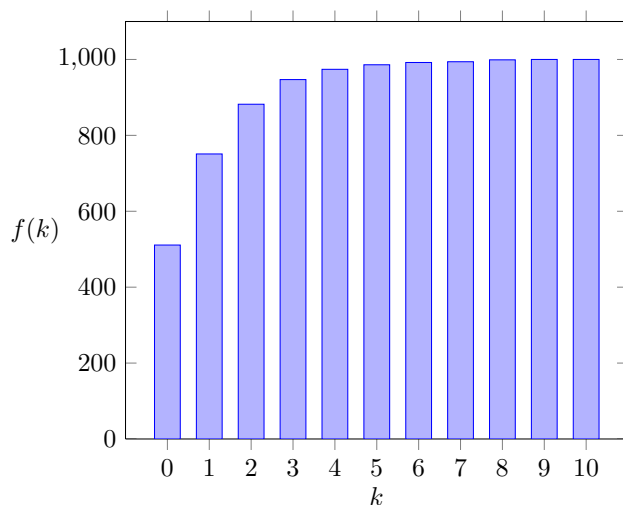


Figure 64. Histogram of Geometric CDF

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

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

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

Mean np , where $p = K/N$

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

Algorithm The distribution is generated through simulation of bernoulli trials.

Source Code

```
1 int hypergeometric( int nTrials, int nPopulation, int nSuccess ) {
2
3     assert( 0 <= nTrials && nTrials <= nPopulation );
4     assert( nPopulation >= 1 && nSuccess >= 0 );
5
6     int count = 0;
7     for ( int i = 0; i < nTrials; i++, nPopulation-- ) {
8
9         double p = double( nSuccess ) / double( nPopulation );
10        if ( bernoulli( p ) ) { count++; nSuccess--; }
11    }
12    return count;
13 }
```

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

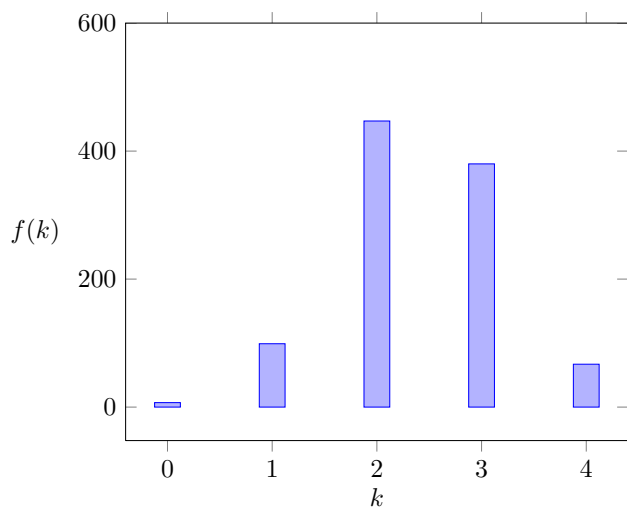


Figure 65. Histogram of hypergeometric PDF

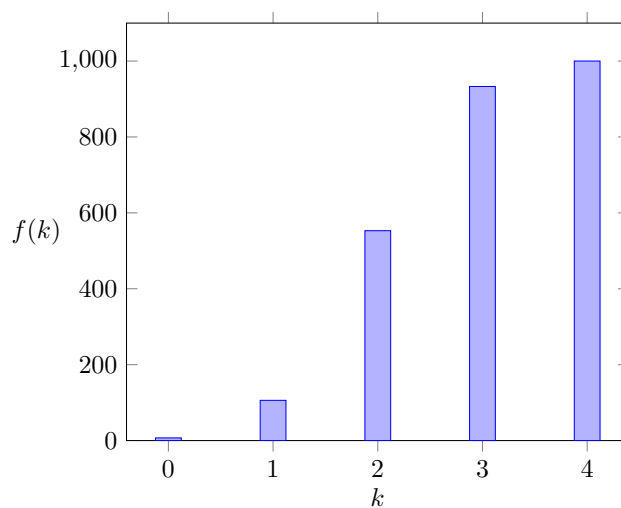


Figure 66. Histogram of hypergeometric CDF

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 $i \in \{1, 2, \dots, m\}$, and where $p_1 + p_2 + \dots + p_m = 1$. 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 $k_1 + \dots + k_m = n$ trials.

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

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

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

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

Source Code

```

1 void multinomial( int    n,           // Multinomial
2                 double p[],         // trials n, probability vector p,
3                 int    count[],      // success vector count,
4                 int    m ) {         // number of disjoint events m
5
6     assert( m >= 2 ); // at least 2 events
7     double sum = 0.;
8     for ( int bin = 0; bin < m; bin++ ) sum += p[ bin ]; // probabilities
9     assert( sum == 1 ); // must sum to 1
10
11     for ( int bin = 0; bin < m; bin++ ) count[ bin ] = 0; // initialize
12
13     // generate n uniform variates in the interval [0,1) and bin the results
14
15     for ( int i = 0; i < n; i++ ) {
16
17         double lower = 0, upper = 0, u = _u01();
18
19         for ( int bin = 0; bin < m; bin++ ) {
20
21             // locate subinterval, which is of length p[ bin ],
22             // that contains the variate and increment the corresponding counter
23
24             lower = upper;
25             upper += p[ bin ];
26             if ( lower <= u && u < upper ) { count[ bin ]++; break; }
27         }
28     }
29 }

```

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 s th success in a sequence of independent Bernoulli trials, where the probability of success in a single trial is p .

Density Function	$f(k) = \begin{cases} \binom{s+k-1}{k} p^s (1-p)^k & k \in \{0, 1, \dots\} \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(k) = \begin{cases} \sum_{i=0}^k \binom{s+i-1}{i} p^k (1-p)^i & \text{if } k \in \{0, 1, \dots\} \\ 0 & \text{otherwise} \end{cases}$
Input	Probability of event, p , where $0 \leq p \leq 1$ and number of successes, $s \geq 1$
Output	The number of failures $k \in \{0, 1, \dots\}$
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 $\leq y$
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 $X = X_1 + \dots + X_s$.

Source Code

```

1 int negativeBinomial( int s, double p ) {
2     assert( s >= 1 );
3     int sum = 0;
4     for ( int i = 0; i < s; i++ ) sum += geometric( p );
5     return sum;
6 }

```

Notes

1. If X_1, \dots, X_s are [geometric](#)(p) variates, then the sum is [negativeBinomial](#)(s, p).
2. The [negativeBinomial](#)(1, p) reduces to [geometric](#)(p).

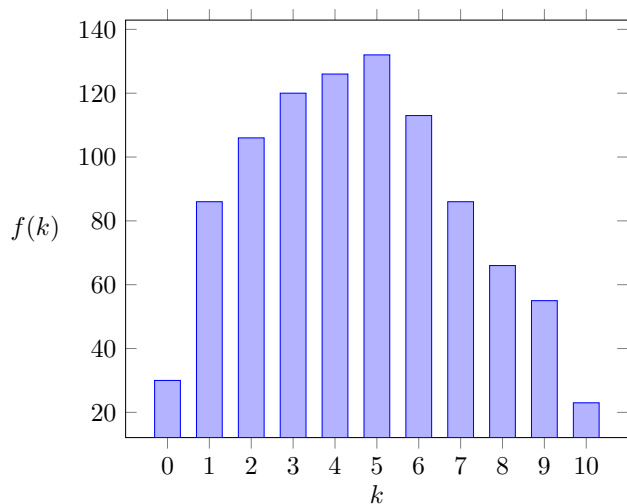


Figure 67. Histogram of negative binomial PDF

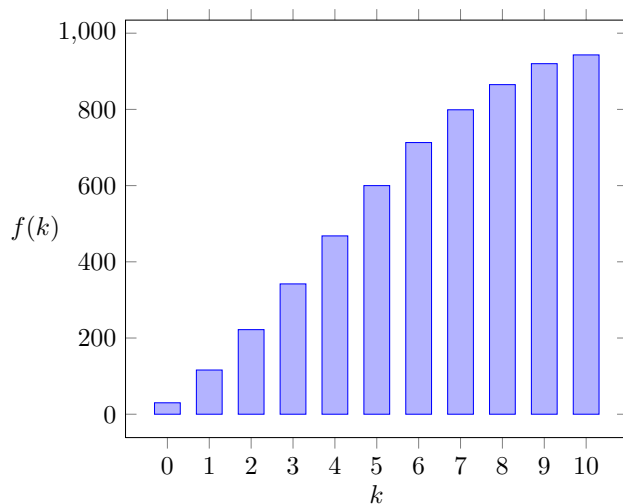


Figure 68. Histogram of negative binomial CDF

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

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

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

Input Probability of event, p , where $0 \leq p \leq 1$ and number of successes, $s \geq 1$

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

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

Mean s/p

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

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

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

Source Code

```
1 int pascal( int s, double p ) {
2   return negativeBinomial( s, p ) + s;
3 }
```

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. $\text{Pascal}(s, p) = \text{negativeBinomial}(s, p) + s$ and $\text{Pascal}(p, 1) = \text{geometric}(p) + 1$.

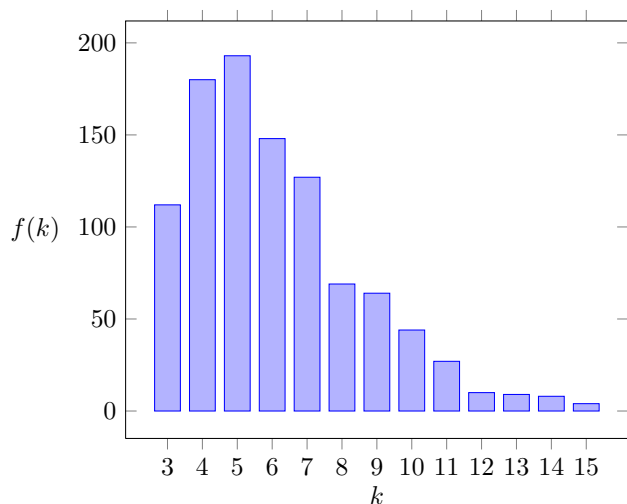


Figure 69. Histogram of Pascal PDF

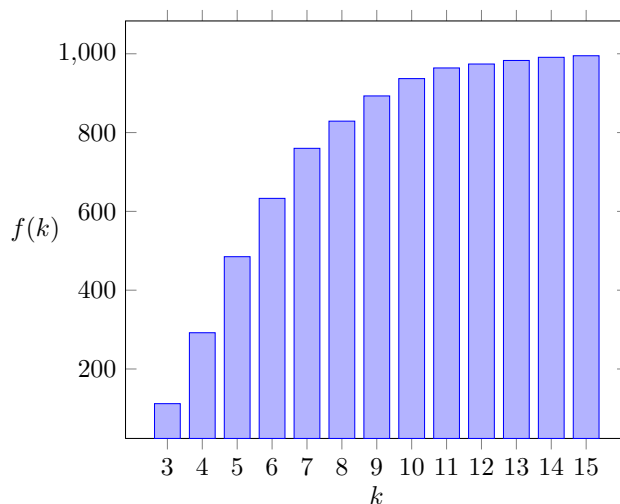


Figure 70. Histogram of Pascal CDF

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

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

Input Rate of occurrence, $\mu > 0$

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

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 $U_{i+1} \sim U(0, 1)$ 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

```
1 int poisson( double mu ) {
2     assert( mu > 0 );
3     double b = 1;
4     int i;
5     for ( i = 0; b >= exp( -mu ); i++ ) b *= uniform( 0, 1 );
6     return i - 1;
7 }
```

Notes

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

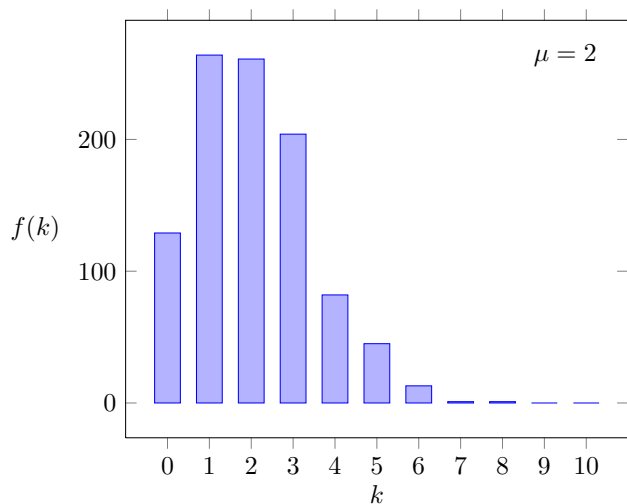


Figure 71. Histogram of Poisson PDF

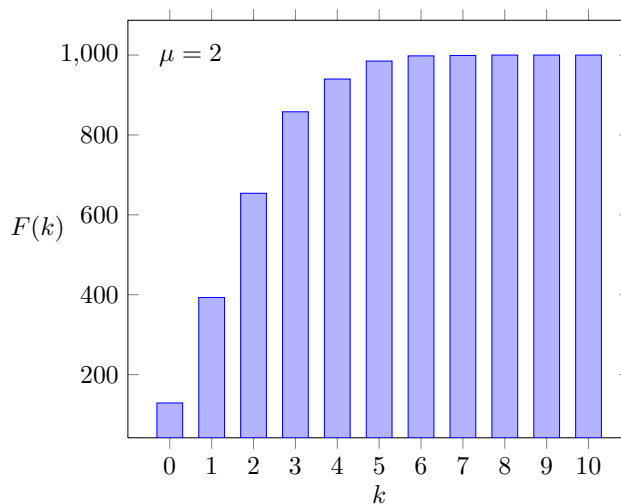


Figure 72. Histogram of Poisson CDF

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

Distribution Function $F(k) = \begin{cases} \frac{k - i_{\min} + 1}{i_{\max} - i_{\min} + 1} & i_{\min} \leq k \leq i_{\max} \\ 1 & k \geq i_{\max} \end{cases}$

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 $U \sim U(0, 1)$.
2. Return $X = i_{\min} + \text{int}([i_{\max} - i_{\min} + 1]U)$.

Source Code

```
1 int uniformDiscrete( int i, int j ) {
2
3     assert( i < j );
4     return i + int( ( j - i + 1 ) * uniform( 0, 1 ) );
5 }
```

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.

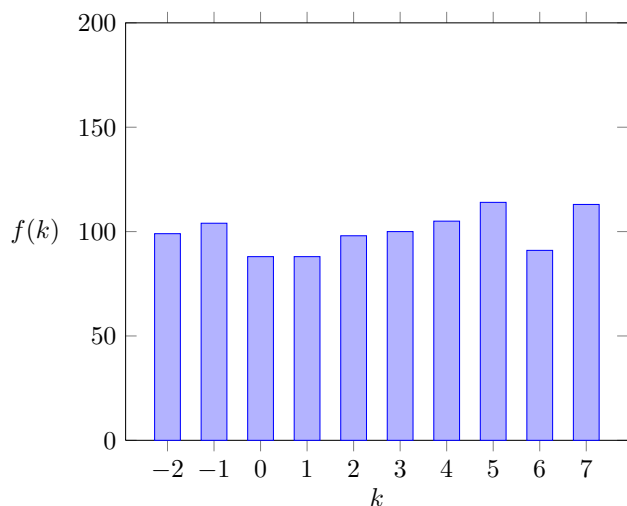


Figure 73. Histogram of uniform discrete PDF

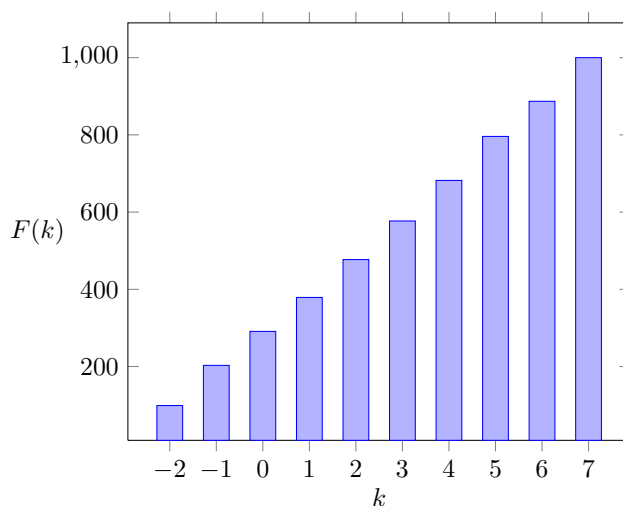


Figure 74. Histogram of uniform discrete CDF

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:

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

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

Output
Algorithm

This algorithm works by the inverse transform method.

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

Source Code

```

1 double Random::empirical( void ) {
2
3     static vector< double > x, cdf;
4     static int n;
5     static bool init = false;
6
7     if ( !init ) {
8         ifstream in( "empiricalDistribution" );
9         if ( !in ) {
10             cerr << "Cannot open \"empiricalDistribution\" file" << endl;
11             exit( 1 );
12         }
13         double value, prob;
14         while ( in >> value >> prob ) { // read in empirical data
15             x.push_back( value );
16             cdf.push_back( prob );
17         }
18         n = x.size();
19         init = true;
20
21         // check that this is indeed a cumulative distribution
22
23         for ( int i = 1; i < n; i++ )
24             assert( cdf[ i - 1 ] < cdf[ i ] );
25         assert( cdf[ n - 1 ] == 1.0 );
26     }
27
28     double p = uniform( 0., 1. );
29     for ( int i = 0; i < n - 1; i++ )
30         if ( cdf[ i ] <= p && p < cdf[ i + 1 ] )
31             return x[ i ] + ( x[ i + 1 ] - x[ i ] ) * ( p - cdf[ i ] ) /
32                 ( cdf[ i + 1 ] - cdf[ i ] );
33     return x[ n - 1 ];
34 }

```

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 $U \sim U(0, 1)$.

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

3. Return $X = k_j$.

Source Code

```

1  int empiricalInt( void ) {
2
3      static vector< int > k;
4      static vector< double > f[ 2 ]; // pdf is f[ 0 ] and cdf is f[ 1 ]
5      static double max;
6      static int n;
7      static bool init = false;
8
9      if ( !init ) {
10         ifstream in ( "empiricalDiscrete" );
11         if ( !in ) {
12             cerr << "Cannot open \"empiricalDiscrete\" file" << endl;
13             exit( 1 );
14         }
15         int value;
16         double freq;
17         while ( in >> value >> freq ) { // read in empirical data
18             k.push_back( value );
19             f[ 0 ].push_back( freq );
20         }
21         n = k.size();
22         init = true;
23
24         // form the cumulative distribution
25
26         f[ 1 ].push_back( f[ 0 ][ 0 ] );
27         for ( int i = 1; i < n; i++ )
28             f[ 1 ].push_back( f[ 1 ][ i - 1 ] + f[ 0 ][ i ] );
29
30         // check that the integer points are in ascending order and that
31         // the cumulative distribution has a maximum in the interval (0,1]
32
33         for ( int i = 1; i < n; i++ ) assert( k[ i - 1 ] < k[ i ] );
34         assert( 0. < f[ 1 ][ n - 1 ] && f[ 1 ][ n - 1 ] <= 1. );
35
36         max = f[ 1 ][ n - 1 ];
37     }
38
39     // select a uniform random number between 0 and the maximum value
40     // of the cumulative distribution
41
42     double p = uniform( 0., max );
43
44     // locate and return the corresponding index
45
46     for ( int i = 0; i < n; i++ ) if ( p <= f[ 1 ][ i ] ) return k[ i ];
47     return k[ n - 1 ];
48 }

```

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 $0 \leq k \leq n$. Sampling *with replacement* effectively makes each sample independent and the probability is given by the formula

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

(See the binomial distribution for comparison.) Let the data be represented by $\{x_1, x_2, \dots, x_N\}$. 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 $\{x_1, x_2, \dots, x_N\}$. (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.

```

1 double sample( bool replace = true ) { // Sample w or w/o replacement from a
2                                     // distribution of 1-D data in a file
3     static vector< double > v;      // vector for sampling with replacement
4     static bool init = false;      // flag that file has been read in
5     static int n;                  // number of data elements in the file
6     static int index = 0;          // subscript in the sequential order
7
8     if ( !init ) {
9         ifstream in( "sampleData" );
10        if ( !in ) {
11            cerr << "Cannot open \"sampleData\" file" << endl;
12            exit( 1 );
13        }
14        double d;
15        while ( in >> d ) v.push_back( d );
16        in.close();
17        n = v.size();
18        init = true;
19        if ( replace == false ) { // sample without replacement
20
21            // shuffle contents of v once and for all
22            // Ref: Knuth, D. E., The Art of Computer Programming, Vol. 2: Seminumerical Algorithms. London: Addison-Wesley, 1969.
23            for ( int i = n - 1; i > 0; i-- ) {
24                int j = int( ( i + 1 ) * _u() );
25                swap( v[ i ], v[ j ] );
26            }
27        }
28    }
29
30    // return a random sample
31    if ( replace ) // sample w/ replacement
32        return v[ uniformDiscrete( 0, n - 1 ) ];
33    else { // sample w/o replacement
34        assert( index < n ); // retrieve elements
35        return v[ index++ ]; // in sequential order
36    }
37 }

```

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 $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{m-1}$ and compute the sample mean:

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

3. Generate m IID uniform variates

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

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.

```

1 struct dSquared : public binary_function< point, point, bool > {
2     bool operator()( point p, point q ) {
3         return p.x * p.x + p.y * p.y < q.x * q.x + q.y * q.y;
4     }
5 };
6
7 point stochasticInterpolation( void ) {
8
9     // Refs: Taylor, M. S. and J. R. Thompson, Computational Statistics & Data
10    // Analysis, Vol. 4, pp. 93-101, 1986; Thompson, J. R., Empirical Model
11    // Building, pp. 108-114, Wiley, 1989; Bodt, B. A. and M. S. Taylor,
12    // A Data Based Random Number Generator for A Multivariate Distribution -
13    // A User's Manual, ARBRL-TR-02439, BRL, APG, MD, Nov. 1982.
14
15    static vector< point > data;
16    static point      min, max;
17    static int        m;
18    static double      lower, upper;
19    static bool        init = false;
20
21    if ( !init ) {
22        ifstream in( "stochasticData" );
23        if ( !in ) {
24            cerr << "Cannot open \"stochasticData\" input file" << endl;
25            exit( 1 );
26        }
27
28        // read in the data and set min and max values
29
30        min.x = min.y = FLT_MAX;
31        max.x = max.y = FLT_MIN;
32        point p;
33        while ( in >> p.x >> p.y ) {
34
35            min.x = ( p.x < min.x ? p.x : min.x );
36            min.y = ( p.y < min.y ? p.y : min.y );
37            max.x = ( p.x > max.x ? p.x : max.x );
38            max.y = ( p.y > max.y ? p.y : max.y );
39
40            data.push_back( p );
41        }
42        in.close();
43        init = true;
44
45        // scale the data so that each dimension will have equal weight
46
47        for ( int i = 0; i < data.size(); i++ ) {
48            data[ i ].x = ( data[ i ].x - min.x ) / ( max.x - min.x );

```



```

50     data[ i ].y = ( data[ i ].y - min.y ) / ( max.y - min.y );
51 }
52
53 // set m, the number of points in a neighborhood of a given point
54
55 m = data.size() / 20; // 5% of all the data points
56 if ( m < 5 ) m = 5; // but no less than 5
57 if ( m > 20 ) m = 20; // and no more than 20
58
59 lower = ( 1. - sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
60 upper = ( 1. + sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
61 }
62
63 // uniform random selection of a data point (with replacement)
64 point origin = data[ uniformInt( 0, data.size() - 1 ) ];
65
66 // make this point the origin of the coordinate system
67 for ( int n = 0; n < data.size(); n++ ) data[ n ] -= origin;
68
69 // sort the data with respect to its distance (squared) from this origin
70 sort( data.begin(), data.end(), dSquared() );
71
72 // find the mean value of the data in the neighborhood about this point
73
74 point mean;
75 mean.x = mean.y = 0.;
76 for ( int n = 0; n < m; n++ ) mean += data[ n ];
77 mean /= double( m );
78
79 // select a random linear combination of the points in this neighborhood
80
81 point p;
82 p.x = p.y = 0.;
83 for ( int n = 0; n < m; n++ ) {
84     double rn;
85     if ( m == 1 ) rn = 1.;
86     else rn = uniform( lower, upper );
87
88     p.x += rn * ( data[ n ].x - mean.x );
89     p.y += rn * ( data[ n ].y - mean.y );
90 }
91
92 // restore the data to its original form
93 for ( int n = 0; n < data.size(); n++ ) data[ n ] += origin;
94
95 // use the mean and the original point to translate the randomly-chosen point
96
97 p += mean;
98 p += origin;
99
100 // scale the randomly-chosen point to the dimensions of the original data
101
102 p.x = p.x * ( max.x - min.x ) + min.x;
103 p.y = p.y * ( max.y - min.y ) + min.y;
104
105 return p;
106 }

```

- 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	<code>cartesianCoord</code>
Bivariate Uniform	uniform distribution in two dimensions	<code>cartesianCoord</code>
Correlated Normal	normal distribution in two dimensions with correlation	<code>cartesianCoord</code>
Correlated Uniform	uniform distribution in two dimensions with correlation	<code>cartesianCoord</code>
Circular Uniform	uniform distribution over the unit circle	<code>polarCoord</code>
Spherical Uniform	uniform distribution over the surface of the unit sphere	<code>sphericalCoord</code>
SphericalND	uniform distribution over the surface of the N -D unit sphere	<code>sphericalCoord</code>

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

Source Code

```

1 std::pair<double,double> bivariateNormal( double muX, double sigmaX,
2                                           double muY, double sigmaY ) {
3
4     assert( sigmaX > 0 && sigmaY > 0 );
5
6     return std::make_pair( normal( muX, sigmaX ), normal( muY, sigmaY ) );
7 }

```

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.

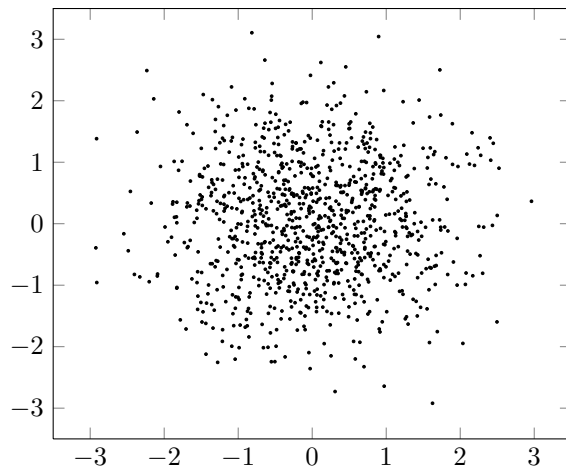


Figure 75. `bivariateNormal(0, 1, 0, 1)`

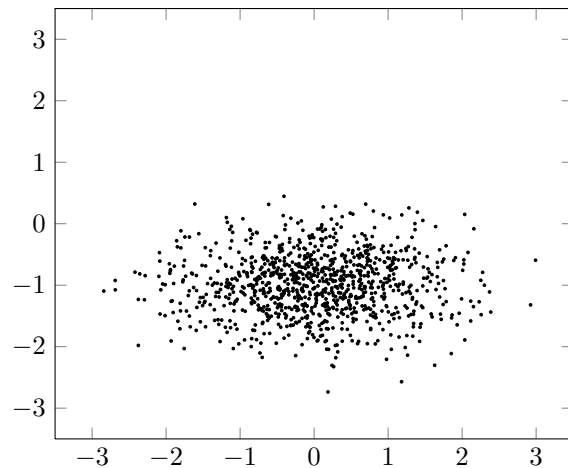


Figure 76. `bivariateNormal(0, 1, -1, 0.5)`

5.4.2 Bivariate Uniform

Density Function	$f(x, y) = \begin{cases} \frac{1}{\pi ab} & 0 \leq \frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} \leq 1 \\ 0 & \text{otherwise} \end{cases}$
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	<ol style="list-style-type: none"> 1. Independently generate $X \sim U(-1, 1)$ and $Y \sim U(-1, 1)$. 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

```

1  std::pair<double,double> bivariateUniform( double xMin, double xMax,
2                                           double yMin, double yMax ) {
3
4      assert( xMin < xMax && yMin < yMax );
5      double x0 = 0.5 * ( xMin + xMax );
6      double y0 = 0.5 * ( yMin + yMax );
7      double a = 0.5 * ( xMax - xMin );
8      double b = 0.5 * ( yMax - yMin );
9      double x, y;
10
11     do {
12         x = uniform( -1, 1 );
13         y = uniform( -1, 1 );
14     } while ( x * x + y * y > 1 );
15
16     return std::make_pair( x0 + a * x, y0 + b * y );
17 }

```

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.

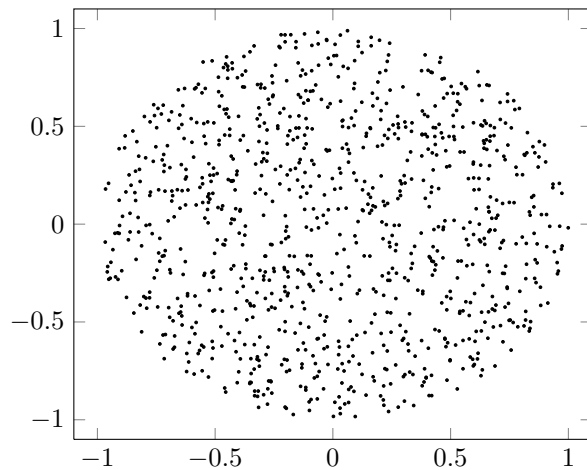


Figure 77. `bivariateUniform(0, 1, 0, 1)`

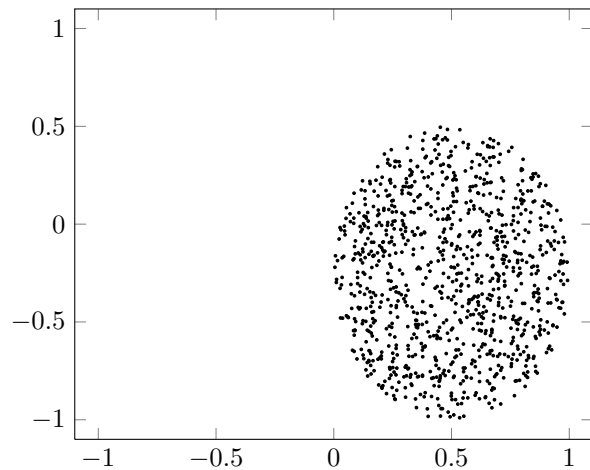


Figure 78. `bivariateUniform(0, 1, -1, 0.5)`

5.4.3 Circular Uniform

Density Function	$f(r, \theta) = \begin{cases} \frac{2}{(r_{\max}^2 - r_{\min}^2)(\theta_{\max} - \theta_{\min})} & 0 \leq r_{\min} \leq r_{\max} \text{ and } \theta_{\min} \leq \theta_{\max} \\ 0 & \text{otherwise} \end{cases}$
Distribution Function	$F(r, \theta) = \begin{cases} \frac{(r^2 - r_{\min}^2)(\theta - \theta_{\min})}{(r_{\max}^2 - r_{\min}^2)(\theta_{\max} - \theta_{\min})} & r_{\min} \leq r \leq r_{\max} \text{ and } \theta_{\min} \leq \theta \leq \theta_{\max} \\ 0 & \text{otherwise} \end{cases}$

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

Source Code

```

1  std::pair<double, double> circularUniform( double rMin, double rMax,
2                                           double thMin, double thMax ) {
3
4     assert( 0 <= rMin && rMin <= rMax && thMin <= thMax );
5
6     double r = sqrt( uniform( rMin * rMin, rMax * rMax ) );
7     double th = uniform( thMin, thMax );
8
9     return make_pair( r, th );
10 }
```

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](#).

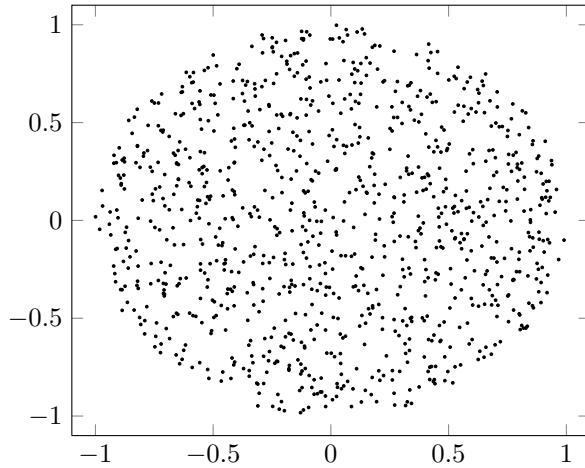


Figure 79. `circularUniform()`

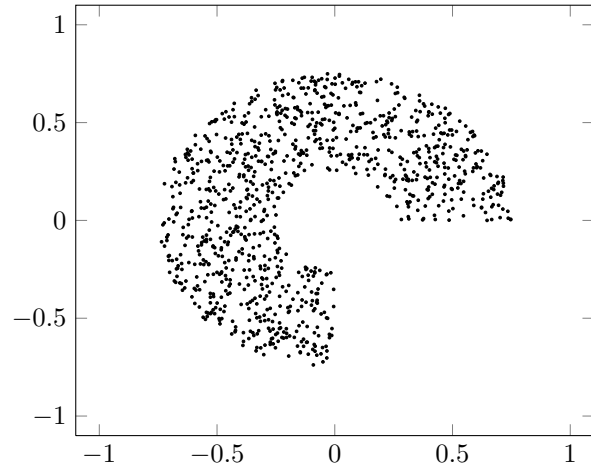


Figure 80. `circularUniform(0.25, 0.75, 0, 270 * M_PI / 180)`

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, $-1 \leq \rho \leq 1$.

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

Source Code

```
1  std::pair<double,double> corrNormal( double r, double muX, double sigmaX,
2                                     double muY, double sigmaY ) {
3
4      assert( -1 <= r && r <= 1 );
5      assert( sigmaX > 0 && sigmaY > 0 );
6
7      double x = normal(0,1);
8      double y = normal(0,1);
9
10     y = r * x + sqrt( 1 - r * r ) * y;
11
12     return std::make_pair( muX + sigmaX * x, muY + sigmaY * y );
13 }
```

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.

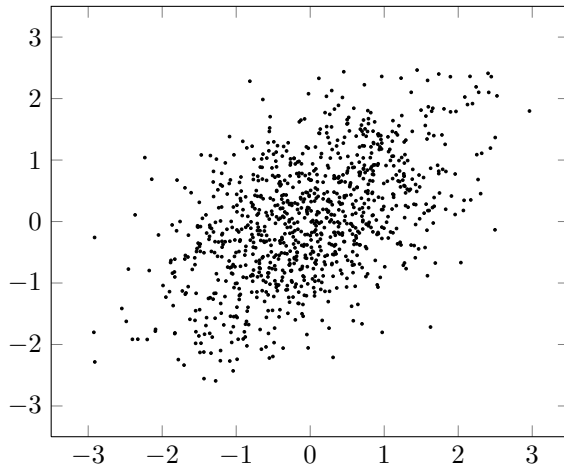


Figure 81. `corrNormal(0.5, 0, 1, 0, 1)`

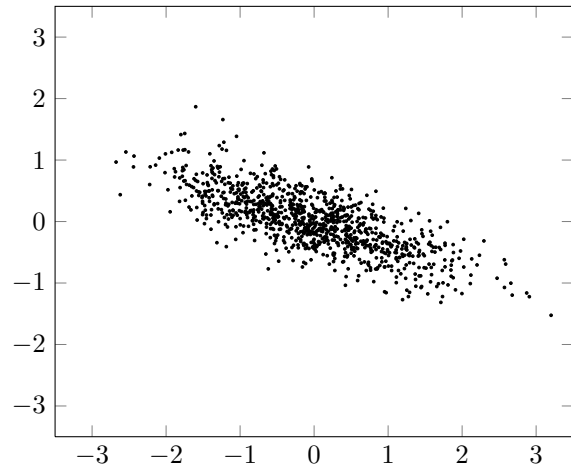


Figure 82. `corrNormal(-0.75, 0, 1, 0, 0.5)`

5.4.5 Correlated Uniform

Input	ρ , correlation coefficient, where $-1 \leq \rho \leq 1$; $[x_{\min}, x_{\max}]$, 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	<ol style="list-style-type: none"> 1. Independently generate $X \sim U(-1, 1)$ and $Z \sim U(-1, 1)$. 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

```

1  std::pair<double,double> corrUniform( double r, double xMin, double xMax,
2                                     double yMin, double yMax ) {
3
4      assert( -1 <= r && r <= 1 );
5      assert( xMin < xMax && yMin < yMax );
6      double x0 = 0.5 * ( xMin + xMax );
7      double y0 = 0.5 * ( yMin + yMax );
8      double a = 0.5 * ( xMax - xMin );
9      double b = 0.5 * ( yMax - yMin );
10     double x, y;
11
12     do {
13         x = uniform( -1, 1 );
14         y = uniform( -1, 1 );
15     } while ( x * x + y * y > 1 );
16
17     y = r * x + sqrt( 1 - r * r ) * y; // correlate variables
18
19     return std::make_pair( x0 + a * x, y0 + b * y );
20 }

```

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.

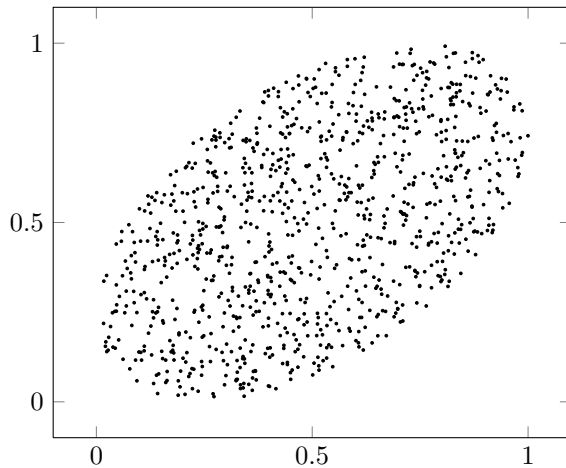


Figure 83. `corrUniform(0.5, 0, 1, 0, 1)`

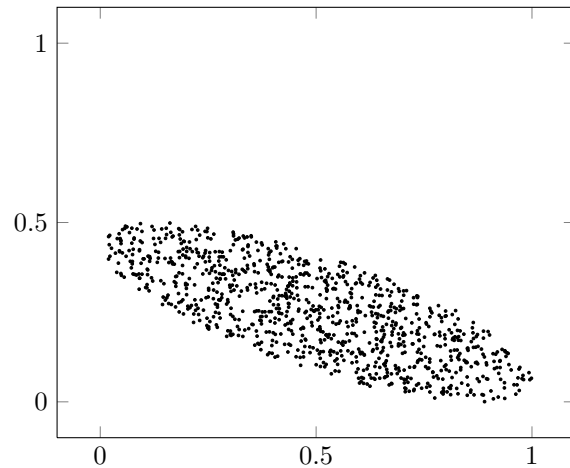


Figure 84. `corrUniform(-0.75, 0, 1, 0, 0.5)`

5.4.6 Spherical Uniform

Density Function	$f(\theta, \phi) = \frac{\sin \theta}{(\phi_{\max} - \phi_{\min})(\cos \theta_{\min} - \cos \theta_{\max})} \text{ for } \begin{cases} 0 \leq \theta_{\min} < \theta < \theta_{\max} \leq \pi \\ 0 \leq \phi_{\min} < \phi < \phi_{\max} \leq 2\pi \end{cases}$
Distribution Function	$F(\theta, \phi) = \frac{(\phi - \phi_{\min})(\cos \theta_{\min} - \cos \theta)}{(\phi_{\max} - \phi_{\min})(\cos \theta_{\min} - \cos \theta_{\max})} \text{ for } \begin{cases} 0 \leq \theta_{\min} < \theta < \theta_{\max} \leq \pi \\ 0 \leq \phi_{\min} < \phi < \phi_{\max} \leq 2\pi \end{cases}$
Input	minimum polar angle, $\theta_{\min} \geq 0$; maximum polar angle, $\theta_{\max} \leq \pi$; minimum azimuthal angle, $\phi_{\min} \geq 0$; maximum azimuthal angle, $\phi_{\max} \leq 2\pi$
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 $U_1 \sim U(\cos \theta_{\max}, \cos \theta_{\min})$ and $U_2 \sim U(\phi_{\min}, \phi_{\max})$. 2. Return $(\Theta, \Phi) = (\cos^{-1}(U_1), U_2)$.

Source Code

```

1 std::pair<double, double> spherical( double thMin, double thMax,
2                                   double phMin, double phMax ) {
3
4     assert( 0 <= thMin && thMin < thMax && thMax <= M_PI &&
5            0 <= phMin && phMin < phMax && phMax <= 2 * M_PI );
6
7     return std::make_pair( acos( uniform( cos( thMax ), cos( thMin ) ) ),
8                           uniform( phMin, phMax ) );
9 }

```

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

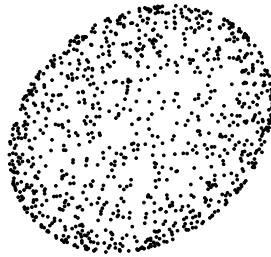


Figure 85. Uniform spherical distribution via calls to spherical()

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., $X_1^2 + X_2^2 + \dots + X_n^2 = 1$)

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

Source Code

```
1 void sphericalND( double x[], // x array returns point
2                 int n ) { // n is number of dimensions
3
4     // generate a point inside the unit n-sphere by normal polar method
5
6     double r2 = 0.;
7     for ( int i = 0; i < n; i++ ) {
8         x[ i ] = normal();
9         r2 += x[ i ] * x[ i ];
10    }
11
12    // project the point onto the surface of the n-sphere by scaling
13
14    const double A = 1. / sqrt( r2 );
15    for ( int i = 0; i < n; i++ ) x[ i ] *= A;
16 }
```

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

$$P_n(x) = (x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0) \pmod{2},$$

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

$$a_n = (x^n + a_{n-1}x^{n-1} + \cdots + a_1x + a_0) \pmod{2}$$

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 tabulated (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 $2^n - 1$.

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

Input

Random number seed (not zero), order n , where $1 \leq n \leq 32$, 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

```
1 void tausworthe( bool* bitvec, unsigned n ) { // returns bit vector of length n
2
3 // It is guaranteed to cycle through all possible combinations of n bits
4 // (except all zeros) before repeating, i.e., cycle is of maximal length 2^n-1.
5 // Ref: Press, W. H., B. P. Flannery, S. A. Teukolsky and W. T. Vetterling,
6 // Numerical Recipes in C, Cambridge Univ. Press, Cambridge, 1988.
7
8 assert( 1 <= n && n <= 32 ); // length of bit vector
9
10 if ( !_seed2 & BIT[ n ] )
11     _seed2 = ( ( !_seed2 ^ MASK[ n ] ) << 1 ) | BIT[ 1 ];
12 else
13     _seed2 <<= 1;
14 for ( int i = 0; i < n; i++ ) bitvec[ i ] = _seed2 & ( BIT[ n ] >> i );
15 }
16
17 bool tausworthe( unsigned n ) // returns a single random bit
18 {
19     assert( 1 <= n && n <= 32 );
20
21     if ( !_seed2 & BIT[ n ] ) {
22         _seed2 = ( ( !_seed2 ^ MASK[ n ] ) << 1 ) | BIT[ 1 ];
23         return true;
24     }
25     else {
26         _seed2 <<= 1;
27         return false;
28     }
29 }
```

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)`.

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.

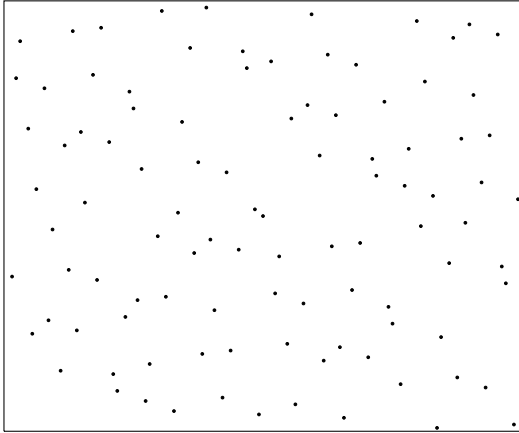


Figure 86. 100 maximal avoidance data points

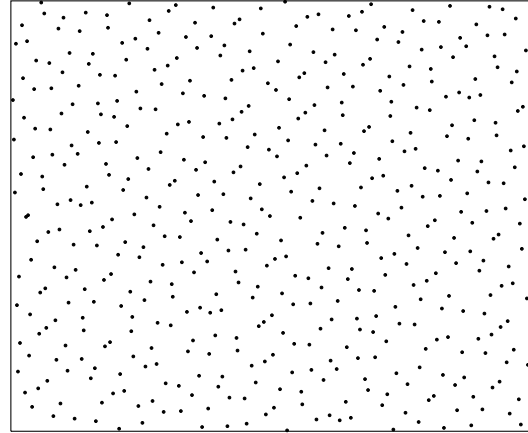


Figure 87. 500 maximal avoidance data points

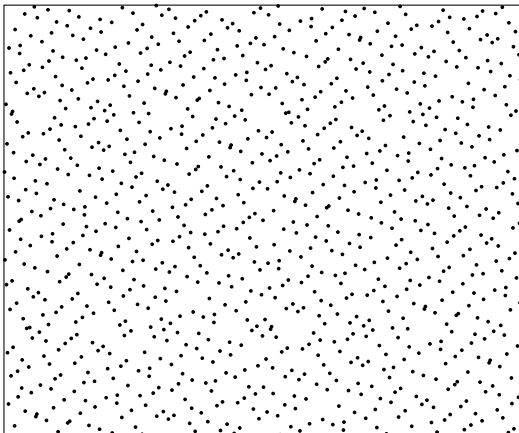


Figure 88. 1000 maximal avoidance data points

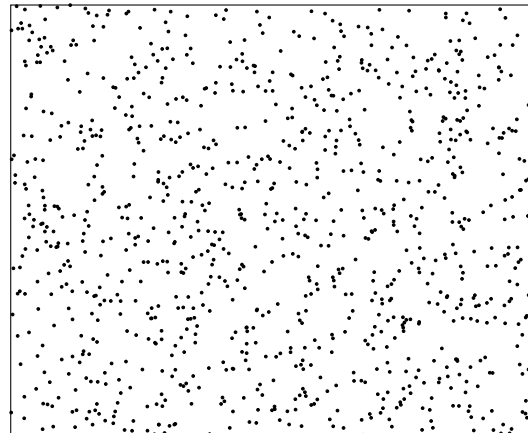


Figure 89. 1000 uniformly distributed data points

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.

```

1 double avoidance( void ) { // 1-dimension (overloaded for convenience)
2
3     double x[ 1 ];
4     avoidance( x, 1 );
5     return x[ 0 ];
6 }
7 void avoidance( double x[], int ndim ) { // multi-dimensional
8
9     static const int MAXBIT = 30;
10    static const int MAXDIM = 6;
11
12    assert( ndim <= MAXDIM );
13    static unsigned long ix[ MAXDIM + 1 ] = { 0 };
14    static unsigned long *u[ MAXBIT + 1 ];
15    static unsigned long mdeg[ MAXDIM + 1 ] = { // degree of primitive polynomial
16        0, 1, 2, 3, 3, 4, 4
17    };
18    static unsigned long p[ MAXDIM + 1 ] = { // decimal encoded interior bits
19        0, 0, 1, 1, 2, 1, 4
20    };
21    static unsigned long v[ MAXDIM * MAXBIT + 1 ] = {
22        0, 1, 1, 1, 1, 1, 1,
23        3, 1, 3, 3, 1, 1,
24        5, 7, 7, 3, 3, 5,
25        15, 11, 5, 15, 13, 9
26    };
27    static double fac;
28    static int in = -1;
29    int j, k;
30    unsigned long i, m, pp;
31
32    if ( in == -1 ) {
33        in = 0;
34        fac = 1. / ( 1L << MAXBIT );
35        for ( j = 1, k = 0; j <= MAXBIT; j++, k += MAXDIM ) u[ j ] = &v[ k ];
36        for ( k = 1, k <= MAXDIM; k++ ) {
37            for ( j = 1; j <= mdeg[ k ]; j++ ) u[ j ][ k ] <= ( MAXBIT - j );
38            for ( j = mdeg[ k ] + 1; j <= MAXBIT; j++ ) {
39                pp = p[ k ];
40                i = u[ j - mdeg[ k ] ][ k ];
41                i ^= ( i >> mdeg[ k ] );
42                for ( int n = mdeg[ k ] - 1; n >= 1; n-- ) {
43                    if ( pp & 1 ) i ^= u[ j - n ][ k ];
44                    pp >>= 1;
45                }
46                u[ j ][ k ] = i;
47            }
48        }
49    }
50    m = in++;
51    for ( j = 0; j < MAXBIT; j++, m >>= 1 ) if ( !( m & 1 ) ) break;
52    if ( j >= MAXBIT ) exit( 1 );
53    m = j * MAXDIM;
54    for ( k = 0; k < ndim; k++ ) {
55        ix[ k + 1 ] ^= v[ m + k + 1 ];
56        x[ k ] = ix[ k + 1 ] * fac;
57    }
58 }

```

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:

```

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

```

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

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

The $\text{negativeBinomial}(s, p)$ represents the number of failures before the s th success in n trials, so it would evaluate as follows:

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

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

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

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

Integrating, we find that the cumulative distribution function is

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

Now, this expression involves transcendental 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.

```
1 double ellipse( double x, double, double ) { // Ellipse Function
2
3     return sqrt( 1. - x * x ) / M_PI_2;
4 }
5
6 double Random::elliptical( void ) { // Elliptical Distribution
7
8     const double X_MIN = -1.;
9     const double X_MAX = 1.;
10    const double Y_MIN = 0.;
11    const double Y_MAX = 1. / M_PI_2;
12
13    return userSpecified( ellipse, X_MIN, X_MAX, Y_MIN, Y_MAX );
14 }
```

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

```
1 #include "Random.h"
2 #include <iostream>
3
4 int main( void ) {
5
6     rng::Random rng;
7     for ( int i = 0; i < 1000; i++ ) std::cout << rng.elliptical() << std::endl;
8
9     return 0;
10 }
```

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
jljkiss	223	168	111	84
jljkiss64	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
jljkiss	All tests were passed	All tests were passed	All tests were passed
jljkiss64	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
jljkiss	$2^{191} \approx 10^{58}$	0.000918	0.038 sec
jljkiss64	$2^{251} \approx 10^{76}$	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 Practitioners*. 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

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

$$x_{i+1} = ax_i + c \pmod{m} \quad (\text{A-1})$$

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

Jump Ahead

If x_0 denotes the seed, then the sequence is

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

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

$$x_n = a^n x_0 + c \sum_{i=0}^{n-1} a^i \pmod{m}, \quad (\text{A-2})$$

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

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

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

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

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

$$\begin{aligned} S_n(a) &= 1 + a + a^2 + a^3 + a^4 + a^5 + a^6 + a^7 + \dots + a^{n-2} + a^{n-1} \\ &= 1 + a + (1 + a)a^2 + (1 + a)a^4 + (1 + a)a^6 + \dots + (1 + a)a^{n-2} \\ &= (1 + a)[1 + a^2 + a^4 + a^6 + \dots + a^{n-2}] \\ &= (1 + a)[1 + (a^2) + (a^2)^2 + (a^2)^3 + \dots + (a^2)^{(n/2-1)}] \\ &= (1 + a)S_{n/2}(a^2). \end{aligned} \quad (\text{A-5})$$

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:

The following C++ code implements this algorithm by making use of the modular functions contained in `mod_math.h`:

```
1 // 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
2 uint64_t gs_mod64( uint64_t a, uintmax_t n, uint64_t m ) {
3
4     if ( n == 0 ) return 0;
```

Algorithm 1 Sum the geometric series $1 + a + a^2 + \dots + a^{n-1} \pmod{m}$ (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  $r$ 

```

```

5
6  uint64_t t = a % m;
7  uint64_t p = 1;
8  uint64_t r = 0;
9
10 while ( n > 1 ) {
11
12     if ( n & 1 ) r = add_mod64( r, mul_mod64( p, pow_mod64( t, n - 1, m ), m ), m );
13     p = mul_mod64( p, add_mod64( 1, t, m ), m );
14     t = mul_mod64( t, t, m );
15     n >>= 1;
16 }
17 r = add_mod64( r, p, m );
18 return r;
19 }

```

The following checks were made

```

1 gs_mod64( 123456789, 0, 4294967296 ) = 0
2 gs_mod64( 123456789, 1, 4294967296 ) = 1
3 gs_mod64( 123456789, 10, 4294967296 ) = 1382346382
4 gs_mod64( 123456789, 1024, 4294967296 ) = 3101645824
5 gs_mod64( 123456789, 1000000, 4294967296 ) = 2009531328
6
7 gs64( 1490024343005336237, 0 ) = 0
8 gs64( 1490024343005336237, 1 ) = 1
9 gs64( 1490024343005336237, 10 ) = 7987679512244350278
10 gs64( 1490024343005336237, 1024 ) = 9396580604419943424
11 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]
Out[1]:= gs[123456789, 0, 2^32]
In[2]:= gs[123456789, 1, 2^32]
Out[2]:= 1
In[3]:= gs[123456789, 10, 2^32]
Out[3]:= 1382346382
In[4]:= gs[123456789, 1024, 2^32]
Out[4]:= 3101645824
In[5]:= gs[123456789, 10^6, 2^32]
Out[5]:= 2009531328
In[6]:= gs[1490024343005336237, 0, 2^64]
Out[6]:= 0
In[7]:= gs[1490024343005336237, 1, 2^64]
Out[7]:= 1
In[8]:= gs[1490024343005336237, 10, 2^64]
Out[8]:= 7987679512244350278
In[9]:= gs[1490024343005336237, 1024, 2^64]
Out[9]:= 9396580604419943424
In[10]:= gs[1490024343005336237, 12345, 2^64]
Out[10]:= 2047449762047247049

```

which sums the first n terms of the geometric series $1 + a + a^2 + \dots + a^{n-1} \pmod{2^{32}}$. Also, note that

```

1 (uint64_t)gs_mod( 123456789, 10, 0, 4294967296 ) = 3101645824
2 (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

$$S_{2^e+c}(a) = \underbrace{1 + a + a^2 + \dots + a^{2^e-1}}_{S_{2^e}(a)} + \underbrace{a^{2^e} + a^{2^e+1} + \dots + a^{2^e+c-1}}_{a^{2^e}(1+a+\dots+a^{c-1})}, \quad (\text{A-6})$$

so that

$$S_{2^e+c}(a) = S_{2^e}(a) + a^{2^e} S_c(a). \quad (\text{A-7})$$

This can be implemented as follows:

```

1 // 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
2 uint64_t gs_mod64( uint64_t a, uint32_t e, uint32_t c, uint64_t m ) {
3
4     if ( e == 0 ) return gs_mod64( a, 1 + c, m );
5
6     uint64_t t = a;
7     uint64_t r = 1;
8
9     for ( uint32_t i = 0; i < e; ++i ) {
10
11         r = mul_mod64( r, add_mod64( 1, t, m ), m );
12         t = mul_mod64( t, t, m );
13     }
14     if ( c == 0 ) return r;
15
16     return add_mod64( r, mul_mod64( t, gs_mod64( a, c, m ), m ), m );
17 }
```

Jump Back

It is also possible to jump backwards. Inverting the equation

$$x_{i+1} = ax_i + c \pmod{m}, \quad (\text{A-8})$$

gives

$$x_{i-1} = a^{-1}(x_i - c) \pmod{m}, \quad (\text{A-9})$$

where we substituted $i \rightarrow i - 1$ and a^{-1} is the multiplicative inverse in the sense that

$$a^{-1}a = aa^{-1} \equiv 1 \pmod{m}.* \quad (\text{A-10})$$

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

$$a^{\phi(m)} \equiv 1 \pmod{m}, \quad (\text{A-11})$$

where $\phi(m)$ is *Euler's Totient* or *Phi function*, which then implies that

$$a^{-1} \pmod{m} = a^{\phi(m)-1} \pmod{m}. \quad (\text{A-12})$$

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

$$x_{-n} = a^{-n}(x - c) + c - c[1 + a^{-1} + a^{-2} + \dots + a^{-(n-1)}] \pmod{m} \quad (\text{A-13})$$

Hence, the *jump back* formula is

$$x_{-n} = a^{-n}(x - c) + c - c \sum_{i=0}^{n-1} a^{-i} \pmod{m}, \quad (\text{A-14})$$

* 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:

```

1 namespace kiss { // period is 2^32 = 4294967296
2 static const uint32_t A = 0x00010dcd; // 69069UL;
3 static const uint32_t C = 0x00003039; // 12345UL;
4 static const uint32_t A_INV = 0xa5e2a705; // 2783094533UL;
5 };
6
7 uint32_t A = kiss::A;
8 uint32_t C = kiss::C;
9 uint32_t A_INV = kiss::A_INV;
10
11 static uint32_t s;
12
13 uint32_t rng( void ) { // random number generator
14
15     s = A * s + C;
16     return s;
17 }
18
19 uint32_t rev( void ) { // random number generator in reverse
20
21     s = A_INV * ( s - C );
22     return s;
23 }

```

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 $m = 2^{64}$,

$$\begin{aligned}
 a^{-1} \pmod{m} &= a^{\phi(m)-1} \pmod{m} = 4294967296 \\
 &= \text{100000000}_{16},
 \end{aligned} \tag{A-18}$$

when $a = 698769069$ and $m = 3001190298811367423$,

$$\begin{aligned}
 a^{-1} \pmod{m} &= a^{\phi(m)-1} \pmod{m} = 4294967296 \\
 &= \text{100000000}_{16},
 \end{aligned} \tag{A-19}$$

when $a = 4294584393$ and $m = 18445099517847011327$, and

$$\begin{aligned} a^{-1} \pmod{m} &= a^{\phi(m)-1} \pmod{m} = 11628298268156854590 \\ &= \text{a16003aa5fc7813e}_{16}, \end{aligned} \quad (\text{A-20})$$

when $a = 4246477509$ and $m = 18445099517847011327$.

We can summarize all these results in Table A-1.

Table 8. Constants for “Jump Back” formula

a	m	$\phi(m)$	$a^{-1} \pmod{m}$
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	a	c	m	$a^{-1} \pmod{m}$
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.

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:

```
1 x ^= ( x << 13 ), x ^= ( x >> 17 ), x ^= ( x << 5 );
```

When this code is applied to 1, represented by the 32-bit bitstring

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 000\mathbf{1}_2 \quad (\text{B-1})$$

it gets transformed into the bitstring

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ \mathbf{1}_2 \Rightarrow 0000\ 0000\ 0000\ 0\mathbf{1}00\ 00\mathbf{1}0\ 0000\ 00\mathbf{1}0\ 000\mathbf{1}_2 \quad (\text{B-2})$$

which is 00042021_{16} in hexadecimal. When the shift register code is applied to 2, it becomes the bit string

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 \Rightarrow 0000\ 0000\ 0000\ 1000\ 0100\ 0000\ 0100\ 0010_2 \quad (\text{B-3})$$

which is 00084042₁₆ in hexadecimal, and so on. Finally, when this code is applied to $2^{32} - 1$, it becomes the bit string

$$\mathbf{1}000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 \Rightarrow \mathbf{1}000\ 0000\ 0000\ \mathbf{1}000\ 0\mathbf{1}00\ 0000\ 0000\ 0000_2 \quad (\text{B-4})$$

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.

```

0000 0000 0000 0000 0000 0000 0000 00012 ⇒ 0000 0000 0000 0100 0010 0000 0010 00012 = 0004202116
0000 0000 0000 0000 0000 0000 0000 00102 ⇒ 0000 0000 0000 1000 0100 0000 0100 00102 = 0008404216
0000 0000 0000 0000 0000 0000 0000 01002 ⇒ 0000 0000 0001 0000 1000 0000 1000 01002 = 0010808416
0000 0000 0000 0000 0000 0000 0000 10002 ⇒ 0000 0000 0010 0001 0000 0001 0000 10002 = 0021010816
0000 0000 0000 0000 0000 0000 0001 00002 ⇒ 0000 0000 0100 0010 0000 0010 0011 00012 = 0042023116
0000 0000 0000 0000 0000 0000 0010 00002 ⇒ 0000 0000 1000 0100 0000 0100 0110 00102 = 0084046216
0000 0000 0000 0000 0000 0000 0100 00002 ⇒ 0000 0001 0000 1000 0000 1000 1100 01002 = 010808c416
0000 0000 0000 0000 0000 0000 1000 00002 ⇒ 0000 0010 0001 0000 0001 0001 1000 10002 = 0210118816
0000 0000 0000 0000 0000 0001 0000 00002 ⇒ 0000 0100 0010 0000 0010 0011 0001 00002 = 0420231016
0000 0000 0000 0000 0000 0010 0000 00002 ⇒ 0000 1000 0100 0000 0100 0110 0010 00002 = 0840462016
0000 0000 0000 0000 0000 0100 0000 00002 ⇒ 0001 0000 1000 0000 1000 1100 0100 00002 = 10808c4016
0000 0000 0000 0000 0000 1000 0000 00002 ⇒ 0010 0001 0000 0001 0001 1000 1000 00002 = 2101188016
0000 0000 0000 0000 0001 0000 0000 00002 ⇒ 0100 0010 0000 0010 0011 0001 0000 00002 = 4202310016
0000 0000 0000 0000 0010 0000 0000 00002 ⇒ 1000 0100 0000 0100 0110 0010 0000 00002 = 8404620016
0000 0000 0000 0000 0100 0000 0000 00002 ⇒ 0000 1000 0000 1000 1100 0100 0000 00002 = 0808c40016
0000 0000 0000 0000 1000 0000 0000 00002 ⇒ 0001 0000 0001 0001 1000 1000 0000 00002 = 1011880016
0000 0000 0000 0001 0000 0000 0000 00002 ⇒ 0010 0000 0010 0011 0001 0000 0000 00002 = 2023100016
0000 0000 0000 0010 0000 0000 0000 00002 ⇒ 0100 0000 0100 0110 0010 0000 0010 00012 = 4046202116
0000 0000 0000 0100 0000 0000 0000 00002 ⇒ 1000 0000 1000 1100 0100 0000 0100 00102 = 808c404216
0000 0000 0000 1000 0000 0000 0000 00002 ⇒ 0000 0001 0000 1000 0000 0000 1000 01002 = 0108008416
0000 0000 0001 0000 0000 0000 0000 00002 ⇒ 0000 0010 0001 0000 0000 0001 0000 10002 = 0210010816
0000 0000 0010 0000 0000 0000 0000 00002 ⇒ 0000 0100 0010 0000 0000 0010 0001 00002 = 0420021016
0000 0000 0100 0000 0000 0000 0000 00002 ⇒ 0000 1000 0100 0000 0000 0100 0010 00002 = 0840042016
0000 0000 1000 0000 0000 0000 0000 00002 ⇒ 0001 0000 1000 0000 0000 1000 0100 00002 = 1080084016
0000 0001 0000 0000 0000 0000 0000 00012 ⇒ 0010 0001 0000 0000 0001 0000 1000 00002 = 2100108016
0000 0010 0000 0000 0000 0000 0000 00002 ⇒ 0100 0010 0000 0000 0010 0001 0000 00002 = 4200210016
0000 0100 0000 0000 0000 0000 0000 00002 ⇒ 1000 0100 0000 0000 0100 0010 0000 00002 = 8400420016
0000 1000 0000 0000 0000 0000 0000 00002 ⇒ 0000 1000 0000 0000 1000 0100 0000 00002 = 0800840016
0001 0000 0000 0000 0000 0000 0000 00002 ⇒ 0001 0000 0000 0001 0000 1000 0000 00002 = 1001080016
0010 0000 0000 0000 0000 0000 0000 00002 ⇒ 0010 0000 0000 0010 0001 0000 0000 00002 = 2002100016
0100 0000 0000 0000 0000 0000 0000 00002 ⇒ 0100 0000 0000 0100 0010 0000 0000 00002 = 4004200016
1000 0000 0000 0000 0000 0000 0000 00002 ⇒ 1000 0000 0000 1000 0100 0000 0000 00002 = 8008400016

```

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

$$0000\ 0000\ 0000\ 01\textcolor{red}{0}0\ 00\textcolor{red}{1}0\ 0000\ 00\textcolor{red}{1}0\ 0001\textcolor{red}{1}_2 \quad (\text{B-5})$$

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

0000 0000 0000 0000 0000 0000 0000 000**1**₂ \Rightarrow 0000 0000 0000 **0**100 00**1**0 0000 00**1**0 000**1**₂
 0000 0000 0000 0000 0000 0000 00**1**0 0000**2**₂ \Rightarrow 0000 0000 **1**000 **0**100 0000 **0**100 **0**1**1**0 00**1**0**2**₂
 0000 0000 0000 0000 00**1**0 0000 0000 0000**2**₂ \Rightarrow **1**000 **0**100 0000 **0**100 **0**1**1**0 00**1**0 0000 0000**2**₂
 0000 0000 0000 **0**100 0000 0000 0000 0000**2**₂ \Rightarrow **1**000 0000 **1**000 **1**100 **0**100 0000 **0**100 00**1**0**2**₂

which are then added together mod 2. In C++ code, mod 2 arithmetic is equivalent to “exclusive-or” (XOR):

$$a + b \pmod{2} \equiv a \oplus b \quad \text{which in C++ code is } a \wedge b, \quad (\text{B-6})$$

where “ \wedge ” is the bitwise XOR. Thus, when these are XORed together, we get

$$\begin{aligned}
& 0000\ 0000\ 0000\ \mathbf{0100}\ 00\mathbf{10}\ 0000\ 00\mathbf{10}\ 000\mathbf{1}_2 \\
\oplus & 0000\ 0000\ \mathbf{1000}\ \mathbf{0100}\ 0000\ 0\mathbf{100}\ 0\mathbf{110}\ 00\mathbf{10}_2 \\
\oplus & \mathbf{1000}\ 0\mathbf{100}\ 0000\ 0\mathbf{100}\ 0\mathbf{110}\ 00\mathbf{10}\ 0000\ 0000_2 \\
\oplus & \mathbf{1000}\ 0000\ \mathbf{1000}\ \mathbf{1100}\ 0\mathbf{100}\ 0000\ 0\mathbf{100}\ 00\mathbf{10}_2 \\
\hline
= & 0000\ 0\mathbf{100}\ 0000\ \mathbf{1000}\ 0000\ 0\mathbf{110}\ 0000\ 000\mathbf{1}_2
\end{aligned} \tag{B-7}$$

Thus, the following C++ code will perform multiplication by the matrix A :

```
1 // multiply a bitmatrix times a vector and return the result
2 uint32_t bitmatrix_mul( const bitmatrix_t& A, uint32_t v ) {
3
4     uint32_t result = 0;
5     for ( size_t i = 0; i < 32; i++, v >= 1 ) if ( v & 1 ) result ^= A.row[i];
6     return result;
7 }
```

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:

[illegible]

The characteristic polynomial is determined by

$$p(\lambda) \equiv \det(A - \lambda I) \pmod{2}. \quad (\text{B-9})$$

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 + λ6 + λ9 + λ14 + λ15 + λ17 + λ18 + λ19 + λ20 + λ21 + λ32
In[2]:= == << FiniteFields*
Out[2]:= IrreduciblePolynomialQ[%1]
In[3]:= True
```

Thus we find that the characteristic polynomial of A , given by

$$p(\lambda) = 1 + \lambda^6 + \lambda^9 + \lambda^{14} + \lambda^{15} + \lambda^{17} + \lambda^{18} + \lambda^{19} + \lambda^{20} + \lambda^{21} + \lambda^{32}, \quad (\text{B-10})$$

is also an *irreducible polynomial*^{*} in the Galois Field $\text{GF}(2^{32})$. Now since every matrix satisfies its own characteristic equation (Cayley-Hamilton theorem), we have

$$1 + A^6 + A^9 + A^{14} + A^{15} + A^{17} + A^{18} + A^{19} + A^{20} + A^{21} + A^{32} = 0 \pmod{2} \quad (\text{B-11})$$

or, since addition and subtraction are equivalent in mod 2 arithmetic,

$$A^6 + A^9 + A^{14} + A^{15} + A^{17} + A^{18} + A^{19} + A^{20} + A^{21} + A^{32} = 1 \pmod{2} \quad (\text{B-12})$$

so that

$$A(A^5 + A^8 + A^{13} + A^{14} + A^{16} + A^{17} + A^{18} + A^{19} + A^{20} + A^{31}) = 1 \pmod{2}, \quad (\text{B-13})$$

and therefore the inverse bitmatrix is given by

$$\begin{aligned} A^{-1} &= A^5 + A^8 + A^{13} + A^{14} + A^{16} + A^{17} + A^{18} + A^{19} + A^{20} + A^{31} \pmod{2} \\ &= A^5 \oplus A^8 \oplus A^{13} \oplus A^{14} \oplus A^{16} \oplus A^{17} \oplus A^{18} \oplus A^{19} \oplus A^{20} \oplus A^{31}. \end{aligned} \quad (\text{B-14})$$

This is easily computed with the functions in `mod_math`, and we get

[illegible]

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.

```
1 Done with shift register
```

```
2 0 0x1234cafe
3 1 0xe602a62b
4 2 0xea347648
5 3 0xfb55c2f6
6 4 0x226f4d13
7 5 0xb2645655
8 6 0x2d73aa42
9 7 0x5f430dbf
10 8 0xe06aaa65
11 9 0x11ec4e36
12 10 0x9d728643
```

```
13
14 Done with bitmatrix, A
```

```
15 0 0x1234cafe
16 1 0xe602a62b
17 2 0xea347648
18 3 0xfb55c2f6
19 4 0x226f4d13
20 5 0xb2645655
21 6 0x2d73aa42
22 7 0x5f430dbf
23 8 0xe06aaa65
24 9 0x11ec4e36
25 10 0x9d728643
```

```
26
27 Done with inverse bitmatrix, A_INV
```

```
28 0 0x9d728643
29 1 0x11ec4e36
30 2 0xe06aaa65
31 3 0x5f430dbf
32 4 0x2d73aa42
33 5 0xb2645655
34 6 0x226f4d13
35 7 0xfb55c2f6
36 8 0xea347648
37 9 0xe602a62b
38 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

$$A = (\mathbf{I} \oplus \mathbf{L}^a)A, \quad A = (\mathbf{I} \oplus \mathbf{R}^b)A, \quad A = (\mathbf{I} \oplus \mathbf{L}^c)A \quad (\text{B-16})$$

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

$$p(x) = 1 + x^6 + x^9 + x^{14} + x^{15} + x^{17} + x^{18} + x^{19} + x^{20} + x^{21} + x^{32}, \quad (\text{B-17})$$

$$A^{-1} = A^5 + A^8 + A^{13} + A^{14} + A^{16} + A^{17} + A^{18} + A^{19} + A^{20} + A^{31}. \quad (\text{B-18})$$

For JKISS, $a = 5$, $b = 7$, $c = 22$, and

$$p(x) = 1 + x^2 + x^8 + x^{10} + x^{11} + x^{12} + x^{14} + x^{20} + x^{21} + x^{22} + x^{23} + x^{24} + x^{32}, \quad (\text{B-19})$$

$$A^{-1} = A + A^7 + A^9 + A^{10} + A^{11} + A^{13} + A^{19} + A^{20} + A^{21} + A^{22} + A^{23} + A^{31}. \quad (\text{B-20})$$

For JLKISS and JLKISS64, $a = 21$, $b = 17$, $c = 30$, and

$$p(x) = 1 + x + x^4 + x^{12} + x^{13} + x^{14} + x^{16} + x^{19} + x^{25} + x^{27} + x^{30} + x^{33} + x^{35} + x^{37} + x^{40} + x^{43} + x^{52} + x^{53} + x^{57} + x^{61} + x^{64}, \quad (\text{B-21})$$

$$A^{-1} = 1 + A^3 + A^{11} + A^{12} + A^{13} + A^{15} + A^{18} + A^{24} + A^{26} + A^{29} + A^{32} + A^{34} + A^{36} + A^{39} + A^{42} + A^{51} + A^{52} + A^{56} + A^{60} + A^{63}. \quad (\text{B-22})$$

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

$$p_1(x) = 1 + x^{13} + x^{19} + x^{25} + x^{31}. \quad (\text{B-23})$$

The second one has $a = 4$, $b = 2$, $c = 25$, and

$$p_2(x) = 1 + x^2 + x^{29}. \quad (\text{B-24})$$

The third one has $a = 17$, $b = 3$, $c = 11$, and

$$p_3(x) = 1 + x^2 + x^3 + x^6 + x^{10} + x^{15} + x^{17} + x^{19} + x^{28}. \quad (\text{B-25})$$

For LFSR113, there are four shift registers, and the characteristic polynomials are

$$p_1(x) = 1 + x^2 + x^4 + x^6 + x^{11} + x^{22} + x^{31} \quad (\text{B-26})$$

$$p_2(x) = 1 + x^2 + x^{29} \quad (\text{B-27})$$

$$p_3(x) = 1 + x^4 + x^8 + x^{12} + x^{13} + x^{16} + x^{20} + x^{24} + x^{28} \quad (\text{B-28})$$

$$p_4(x) = 1 + x^3 + x^4 + x^5 + x^6 + x^{11} + x^{12} + x^{18} + x^{25} \quad (\text{B-29})$$

For LFSR258, there are five shift registers, and the characteristic polynomials are

$$p_1(x) = 1 + x + x^{13} + x^{38} + x^{63} \quad (\text{B-30})$$

$$p_2(x) = 1 + x^{11} + x^{24} + x^{44} + x^{55} \quad (\text{B-31})$$

$$p_3(x) = 1 + x^2 + x^3 + x^6 + x^8 + x^{14} + x^{16} + x^{18} + x^{27} + x^{35} + x^{52} \quad (\text{B-32})$$

$$p_4(x) = 1 + x^4 + x^5 + x^7 + x^{11} + x^{14} + x^{17} + x^{21} + x^{24} + x^{27} + x^{34} + x^{37} + x^{47} \quad (\text{B-33})$$

$$p_5(x) = 1 + x^3 + x^{41} \quad (\text{B-34})$$

The inverse bitmatrix $A^{-1} \pmod{2}$ is easily computed from the characteristic polynomial. For example, consider the characteristic equation

$$p_1(x) = 1 + x + x^{13} + x^{38} + x^{63} = 0 \pmod{2} \quad (\text{B-35})$$

Adding 1 mod 2 to both sides gives

$$x + x^{13} + x^{38} + x^{63} = x(1 + x^{12} + x^{37} + x^{62}) = 1 \pmod{2} \quad (\text{B-36})$$

and since the bitmatrix satisfies its own characteristic polynomial (Cayley-Hamilton theorem), we get

$$A^{-1} = 1 + A^{12} + A^{37} + A^{62} \pmod{2}. \quad (\text{B-37})$$

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 approximately 100 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.

$$\begin{aligned} P_{\text{LFSR88}} &= (2^a - 1)(2^b - 1)(2^c - 1) \\ &= 2^{a+b+c} - 2^{a+b} - 2^{a+c} - 2^{b+c} + 2^a + 2^b + 2^c - 1, \end{aligned} \quad (\text{B-38})$$

where $a = 31$, $b = 29$, and $c = 28$. We can code this as follows:

```
1 virtual void jump_cycle( void ) { // jump ahead a full cycle of lfsr88
2   const uint32_t A = 31, B = 29, C = 28;
3   jump_ahead( A + B + C, 0 );
4   jump_back( A + B, 0 ); jump_back( A + C, 0 ); jump_back( B + C, 0 );
5   jump_ahead( A, 0 ); jump_ahead( B, 0 ); jump_ahead( C, 0 );
6   jump_back( 1 );
7 }
```

LFSR113 consists of four independent shift registers, and its period is

$$\begin{aligned} P_{\text{LFSR113}} &= (2^a - 1)(2^b - 1)(2^c - 1)(2^d - 1) \\ &= 2^{a+b+c+d} - 2^{b+c+d} - 2^{a+c+d} - 2^{a+b+d} - 2^{a+b+c} + \\ &\quad 2^{a+b} + 2^{a+c} + 2^{a+d} + 2^{b+c} + 2^{b+d} + 2^{c+d} - \\ &\quad 2^a - 2^b - 2^c - 2^d + 2^e + 1, \end{aligned} \quad (\text{B-39})$$

where $a = 31$, $b = 29$, $c = 28$, $d = 25$.

Finally, LFSR258 consists of five independent shift registers, and its period is

$$\begin{aligned} P_{\text{LFSR258}} &= (2^a - 1)(2^b - 1)(2^c - 1)(2^d - 1)(2^e - 1), \\ &= 2^{a+b+c+d+e} - 2^{b+c+d+e} - 2^{a+c+d+e} - 2^{a+b+d+e} - 2^{a+b+c+e} - 2^{a+b+c+d} + \\ &\quad 2^{c+d+e} + 2^{b+d+e} + 2^{b+c+e} + 2^{b+c+d} + 2^{a+d+e} + \\ &\quad 2^{a+c+e} + 2^{a+c+d} + 2^{a+b+e} + 2^{a+b+d} + 2^{a+b+c} - \\ &\quad 2^{a+b} - 2^{a+c} - 2^{a+d} - 2^{a+e} - 2^{b+c} - \\ &\quad 2^{b+d} - 2^{b+e} - 2^{c+d} - 2^{c+e} - 2^{d+e} + \\ &\quad 2^a + 2^b + 2^c + 2^d + 2^e - 1, \end{aligned} \quad (\text{B-40})$$

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

$$10^{10} \times 60 \times 60 \times 24 \times 30 = 2.592 \times 10^{16} \quad (\text{B-41})$$

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

$$\begin{aligned} x_{2^{20}} = & x_{30} \oplus x_{27} \oplus x_{26} \oplus x_{25} \oplus x_{24} \oplus x_{23} \oplus x_{21} \oplus \\ & x_{20} \oplus x_{19} \oplus x_{18} \oplus x_{14} \oplus x_{12} \oplus x_9 \oplus x_8 \oplus x_5. \end{aligned} \quad (\text{B-42})$$

Now LFSR88 has three shift registers, with corresponding characteristic polynomials

$$\begin{aligned} p_1(x) &= 1 + x^{13} + x^{19} + x^{25} + x^{31} \\ p_2(x) &= 1 + x^2 + x^{29} \\ p_3(x) &= 1 + x^2 + x^3 + x^6 + x^{10} + x^{15} + x^{17} + x^{19} + x^{28} \end{aligned} \quad (\text{B-43})$$

Using MATHEMATICA, we can verify this result:

MATHEMATICA Session

```
In[1]:= PolynomialMod[PolynomialMod[x220, 1 + x13 + x19 + x25 + x31], 2]
Out[1]:= x5 + x8 + x9 + x12 + x14 + x18 + x19 + x20 + x21 + x23 + x24 + x25 + x26 + x27 + x30
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:

$$b^{25} = b^{16+8+1} = b^{2^4+2^3+2^0} = b^{11001_2}. \quad (\text{B-44})$$

Then the algorithm proceeds as follows:

Initialize $r = 1$, $t = b$.

	$r \leftarrow r \cdot t$	$t \leftarrow t^2$
1100 1 ₂	b	b^2
1100 1 ₂		b^4
11 00 1 ₂		b^8
1 1001 ₂	$b \cdot b^8$	b^{16}
1 1001 ₂	$bb^8 \cdot b^{16}$	b^{32}

Return $r = bb^8b^{16}$.

Another example:

$$b^{62} = b^{32+16+8+4+2} = b^{2^5+2^4+2^3+2^2+2^1} = b^{111110_2}. \quad (\text{B-45})$$

Given base b and exponent n .

Express n in binary form and initialize $r = 1$, $t = b$.

	$r \leftarrow r \cdot t$	$t \leftarrow t^2$
11111 0 ₂	1	b^2
1111 1 0 ₂	b^2	b^4
111 1 10 ₂	$b^2 \cdot b^4$	b^8
11 1 110 ₂	$b^2b^4 \cdot b^8$	b^{16}
1 11110 ₂	$b^2b^4b^8 \cdot b^{16}$	b^{32}
1 11110 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 0 ₂	1	b^2
1111 1 0 ₂	b^2	b^4
111 1 10 ₂	$b^2 \cdot b^4$	b^8
11 1 110 ₂	$b^2 b^4 \cdot b^8$	b^{16}
1 11110 ₂	$b^2 b^4 b^8 \cdot b^{16}$	b^{32}
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 0 ₂	1	x^2
1111 1 0 ₂	x^2	x^4
111 1 10 ₂	$x^2 \cdot x^4$	x^8
11 1 110 ₂	$x^6 \cdot x^8$	x^{16}
1 11110 ₂	$x^{14} \cdot x^{16}$	$x^{26} + x^{20} + x^{14} + x$
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

$$a(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0 \quad \text{and} \quad b(x) = b_{m-1}x^{m-1} + \dots + b_1x + b_0$$

Then, writing $a(x)$ in reverse order, we have

$$\begin{aligned} a(x) \cdot b(x) &= (a_0 + a_1x + a_2x^2 + \dots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1}) \cdot b(x) \\ &= a_0b(x) + xa_1b(x) + x^2a_2b(x) + \dots + x^{n-2}a_{n-2}b(x) + x^{n-1}a_{n-1}b(x) \\ &= a_0b(x) + x(a_1b(x) + x(a_2b(x) + \dots + x(a_{n-2}b(x) + x(a_{n-1}b(x)) \dots))). \end{aligned} \tag{B-51}$$

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:

```

1  const size_t N_BITS = 33; // degree of the irreducible polynomial, p
2
3  // returns a * b mod p
4  bitset<N_BITS> poly_mul( const bitset<N_BITS>& a, const bitset<N_BITS>& b, const bitset<N_BITS>& p ) {
5
6      static size_t msb = 0;
7      bitset<N_BITS> r;
8      int c;
9      if ( msb == 0 ) { // get msb of irreducible polynomial
10         for ( size_t k = p.size()-1; k >= 0; --k ) if ( p.test(k) ) { msb = k; break; }
11     }
12     for ( int i = N_BITS-1; i >= 0; --i ) {
13
14         r <<= 1;
15         c = r[msb];
16         if ( c ) r ^= p;
17         if ( a[i] ) r ^= b;
18     }
19     return r;
20 }
21
22 // returns a^2 mod p
23 bitset<N_BITS> poly_sqr( const bitset<N_BITS>& a, const bitset<N_BITS>& p ) {
24
25     static size_t msb = 0;
26     bitset<N_BITS> r;
27     int c;
28     if ( msb == 0 ) { // get msb of irreducible polynomial
29         for ( size_t k = p.size()-1; k >= 0; --k ) if ( p.test(k) ) { msb = k; break; }
30     }
31     for ( int i = N_BITS-1; i >= 0; --i ) {
32
33         r <<= 1;
34         c = r[msb];
35         if ( c ) r ^= p;
36         if ( a[i] ) r ^= a;
37     }
38     return r;
39 }
40
41 // returns a^n mod p
42 bitset<N_BITS> poly_pow( uintmax_t n, const bitset<N_BITS>& p ) {
43
44     bitset<N_BITS> r( 0x1 ); // 1
45     bitset<N_BITS> t( 0x2 ); // a
46
47     while ( n > 0 ) {
48
49         if ( n & 1 ) r = poly_mul( r, t, p );
50         t = poly_sqr( t, p );
51         n >>= 1;
52     }
53     return r;
54 }
55
56 // returns a^n mod p, where n = 2^e + c
57 bitset<N_BITS> poly_pow( uint32_t e, uint32_t c, const bitset<N_BITS>& p ) {
58
59     bitset<N_BITS> t( 0x2 ); // a
60     if ( e > 0 ) {
61         for ( uint32_t i = 0; i < e; ++i ) t = poly_sqr( t, p );
62     }
63     if ( e == 0 ) c++;
64     bitset<N_BITS> r = poly_pow( c, p );
65     if ( e ) r = poly_mul( r, t, p );
66
67     return r;
68 }

```

C Multiply with Carry Generator

The multiply-with-carry (MWC) method^{*} is defined by the sequence

$$x_{i+1} = ax_i + c \pmod{m} \quad (\text{C-1})$$

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;

```

^{*} Marsaglia, G. Random Number Generators, Journal of Modern Applied Statistical Methods, May 2003, Vol. 2., No. 1, 2-13.

The period of JKISS is

$$\begin{aligned}
P_{\text{JKISS}} &= 2^{32}(2^{32} - 1)(4294584393 \times 2^{31} - 1) \\
&= 170\,126\,015\,070\,303\,082\,434\,102\,628\,274\,311\,004\,160 \\
&= 111111111111101000101000010010000000000000010111 \\
&\quad 010111101101010000000000000000000000000000100000 \\
&\quad 00000000000000000000000000_2 \quad (127 \text{ digits in base } 2).
\end{aligned} \tag{C-5}$$

The period of JLKISS is

$$\begin{aligned}
P_{\text{JLKISS}} &= 2^{64}(2^{64} - 1)(4294584393 \times 2^{31} - 1) \\
&= 3138271061012620924047441856806230331094853687768430673920 \\
&= 1111111111110100010100001001000111111111111111111 \\
&\quad 111111111110100000000000001011101011101101110000 \\
&\quad 0000000000000000000000000000100000000000000000000 \\
&\quad 000_2 \quad (191 \text{ digits in base } 2). \tag{C-6}
\end{aligned}$$

The period of JLKISS64 is

$$\begin{aligned}
P_{\text{JLKISS64}} &= 2^{64}(2^{64} - 1)(4294584393 \times 2^{31} - 1)(698769069 \times 2^{31} - 1) \\
&= 47092743316757674365569961704867972203 \\
&\quad 43483431369386641683085078522096517120 \\
&= 10100110100101011010110101010010100111001011100000 \\
&\quad 01110101001010000011000110011000011010111111010110 \\
&\quad 00110100011111100010110001010100110100000100001101 \\
&\quad 1110101111111111111111111111111111111000000000000 \\
&\quad 000 \\
&\quad 00_2 \quad (252 \text{ digits in base 2}).
\end{aligned} \tag{C-7}$$

These are all handled with `bitset` data structures of size, 127, 191, and 252, respectively.

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

```

1 // mod_math.h: modular math for 32-bit and 64-bit unsigned integers
2 // Ref: https://github.com/cmcqueen/simplerandom
3 // R. Saucier, 14 October 2016
4
5 #include <stdint>
6 #include <cassert>
7
8 static const uint64_t M = 4294967296ULL; // 2^32
9 static const uint64_t TW032 = 4294967296ULL; // 2^32
10 static const long double TW017 = 131072.0L; // 2^17
11 static const long double TW035 = 34359738368.0L; // 2^35
12 static const long double TW053 = 9007199254740992.0L; // 2^53
13 static const long double TW032_INV = 2.328306436538696289962500e-10L; // 2^(-32)
14 static const long double TW064_INV = 5.421010862427522170037264e-20L; // 2^(-64)
15
16 // a + b mod m
17 uint32_t add_mod32( uint32_t a, uint32_t b, uint32_t m ) {
18
19 #ifdef UINT64_C // use the native 64-bit capability
20
21     if ( b <= UINT32_MAX - a )
22         return ( a + b ) % m;
23     else
24         return ( uint64_t( a ) + b ) % m;
25
26 #else // native 64-bit not available, so perform addition using 32-bit
27
28     a %= m;

```

```

29     b %= m;
30     uint32_t t;
31     if ( b <= UINT32_MAX - a )
32         return ( a + b ) % m;
33
34     if ( m <= ( UINT32_MAX >> 1 ) )
35         return ( ( a % m ) + ( b % m ) ) % m;
36
37     t = a + b;
38     if ( t > uint32_t( m * 2 ) ) // m*2 must be truncated before compare
39         t -= m;
40     t -= m;
41     return t % m;
42
43 #endif // UINT64_C
44 }
45
46 // a + b mod 2^32
47 uint32_t add32( uint32_t a, uint32_t b ) {
48
49 #ifdef UINT64_C // use the native 64-bit capability
50
51     if ( b <= UINT32_MAX - a )
52         return ( a + b ) % M;
53     else
54         return ( uint64_t( a ) + b ) % M;
55
56 #else // native 64-bit not available, so perform addition using 32-bit
57
58     a %= M;
59     b %= M;
60     uint32_t t;
61     if ( b <= UINT32_MAX - a )
62         return ( a + b ) % M;
63
64     if ( m <= ( UINT32_MAX >> 1 ) )
65         return ( ( a % M ) + ( b % M ) ) % M;
66
67     t = a + b;
68     if ( t > uint32_t( M * 2 ) ) // m*2 must be truncated before compare
69         t -= M;
70     t -= M;
71     return t % M;
72
73 #endif // UINT64_C
74 }
75
76 // a * b mod m
77 uint32_t mul_mod32( uint32_t a, uint32_t b, uint32_t m ) {
78
79 #ifdef UINT64_C // use the native 64-bit capability
80
81     uint64_t t = uint64_t( a ) * b;
82     return uint32_t( t % m );
83
84 #else // native 64-bit not available, so perform multiplication using 32-bit
85
86     a %= m;
87     b %= m;
88     uint32_t r = 0;
89     uint32_t t;
90
91     if ( b >= m ) {
92
93         if ( m > UINT32_MAX / 2u ) b -= m;
94         else b %= m;
95     }
96
97     while ( a != 0 ) {
98
99         if ( a & 1u ) {
100
101             if ( b >= m - r ) r -= m;
102             r += b;
103         }
104         a >>= 1u;
105
106         t = b;
107         if ( b >= m - t ) t -= m;
108         b += t;
109     }
110     return r;
111
112 #endif // UINT64_C
113 }
114
115 // a * b mod 2^32
116 uint32_t mul32( uint32_t a, uint32_t b ) {
117
118 #ifdef UINT64_C // use the native 64-bit capability
119
120     uint64_t t = uint64_t( a ) * b;
121     return uint32_t( t % M );
122
123 #else // native 64-bit not available, so perform multiplication using 32-bit
124
125     a %= M;
126     b %= M;
127     uint32_t r = 0;
128     uint32_t t;
129
130     if ( b >= m ) {
131
132         if ( m > UINT32_MAX / 2u ) b -= M;
133         else b %= M;
134     }
135

```

```

136 while ( a != 0 ) {
137     if ( a & 1u ) {
138         if ( b >= M - r ) r -= M;
139         r += b;
140     }
141     a >>= 1u;
142     t = b;
143     if ( b >= M - t ) t -= M;
144     b += t;
145 }
146 return r;
147
148 #endif // UINT64_C
149 }
150
151 // 32-bit methods
152
153 // a^n mod m
154 uint32_t pow_mod32( uint32_t a, uintmax_t n, uint32_t m ) {
155     uint32_t r = 1;
156     uint32_t t = a;
157     for (;;) {
158         if ( n & 1 ) r = mul_mod32( r, t, m );
159         n >>= 1;
160         if ( n == 0 ) break;
161         t = mul_mod32( t, t, m );
162     }
163     return r;
164 }
165
166 // a^n mod m, where n = 2^e + c
167 uint32_t pow_mod32( uint32_t a, uintmax_t e, uintmax_t c, uint32_t m ) {
168     if ( e == 0 ) return pow_mod32( a, c + 1, m );
169     uint32_t t = a;
170     for ( uintmax_t i = 0; i < e; ++i ) t = mul_mod32( t, t, m );
171     return mul_mod32( pow_mod32( a, c, m ), t, m );
172 }
173
174 // a^n mod 2^32
175 uint32_t pow32( uint32_t a, uintmax_t n ) {
176     uint32_t r = 1;
177     uint32_t t = a;
178     for (;;) {
179         if ( n & 1 ) r *= t;
180         n >>= 1;
181         if ( n == 0 ) break;
182         t *= t;
183     }
184     return r;
185 }
186
187 // a^n mod 2^32, where n = 2^e + c
188 uint64_t pow32( uint32_t a, uint32_t e, uint32_t c ) {
189     if ( e == 0 ) return pow32( a, c + 1 );
190     uint32_t t = a;
191     for ( uint32_t i = 0; i < e; ++i ) t = mul32( t, t );
192     return mul32( pow32( a, c ), t );
193 }
194
195 // sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
196 uint32_t gs_mod32( uint32_t a, uint32_t n, uint32_t m ) {
197     if ( n == 0 ) return 0;
198     uint32_t t = a % m;
199     uint32_t p = 1;
200     uint32_t r = 0;
201     while ( n > 1 ) {
202         if ( n & 1 ) r = add_mod32( r, mul_mod32( p, pow_mod32( t, n - 1, m ), m ), m );
203         p = mul_mod32( p, add_mod32( 1, t, m ), m );
204         t = mul_mod32( t, t, m );
205         n >>= 1;
206     }
207     r = add_mod32( r, p, m );
208     return r;
209 }
210
211 // sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
212 uint32_t gs_mod32( uint32_t a, uint32_t e, uint32_t c, uint32_t m ) {
213     if ( e == 0 ) return gs_mod32( a, 1 + c, m );
214     uint32_t t = a;
215     uint32_t r = 1;
216     for ( uint32_t i = 0; i < e; ++i ) {
217         r = mul_mod32( r, add_mod32( 1, t, m ), m );
218         t = mul_mod32( t, t, m );
219     }
220     if ( c == 0 ) return r;
221     return add_mod32( r, mul_mod32( t, gs_mod32( a, c, m ), m ), m );
222 }
223

```

```

243
244 // sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^32
245 uint32_t gs32( uint32_t a, uintmax_t n ) {
246
247     if ( n == 0 ) return 0;
248     if ( n == 1 ) return 1;
249
250     uint32_t t = a;
251     uint32_t p = 1;
252     uint32_t r = 0;
253
254     while ( n > 1 ) {
255
256         if ( n & 1 ) r += p * pow32( t, n - 1 );
257         p = ( 1 + t );
258         t *= t;
259         n >>= 1;
260     }
261     r += p;
262     return r;
263 }
264
265 // sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^32, where n = 2^e + c
266 uint32_t gs32( uint32_t a, uint32_t e, uint32_t c ) {
267
268     if ( e == 0 ) return gs32( a, 1 + c );
269
270     uint32_t t = a;
271     uint32_t r = 1;
272
273     for ( uint32_t i = 0; i < e; ++i ) {
274
275         r = mul32( r, add32( 1, t ) );
276         t = mul32( t, t );
277     }
278     if ( c == 0 ) return r;
279     return add32( r, mul32( t, gs32( a, c ) ) );
280 }
281
282 // 64-bit methods
283
284 #ifdef UINT64_C // the following require 64-bit capability
285
286 // 64-bit computation of a + b mod m
287 uint64_t add_mod64( uint64_t a, uint64_t b, uint64_t m ) {
288
289     a %= m;
290     b %= m;
291     uint64_t t;
292     if ( b <= UINT64_MAX - a )
293         return ( a + b ) % m;
294
295     if ( m <= ( UINT64_MAX >> 1 ) )
296         return ( ( a % m ) + ( b % m ) ) % m;
297
298     t = a + b;
299     if ( t > uint64_t( m * 2 ) ) // m*2 must be truncated before compare
300         t -= m;
301     t -= m;
302     return t % m;
303 }
304
305 // 64-bit computation of a + b mod 2^64
306 uint64_t add64( uint64_t a, uint64_t b ) {
307
308     return a + b;
309 }
310
311 // 64-bit computation of a * b mod m
312 uint64_t mul_mod64( uint64_t a, uint64_t b, uint64_t m ) {
313
314     uint64_t r = 0;
315     uint64_t t;
316
317     if ( b >= m ) {
318
319         if ( m > UINT64_MAX / 2u ) b -= m;
320         else b %= m;
321     }
322
323     while ( a != 0 ) {
324
325         if ( a & 1 ) {
326
327             if ( b >= m - r ) r -= m;
328             r += b;
329         }
330         a >>= 1;
331
332         t = b;
333         if ( b >= m - t ) t -= m;
334         b += t;
335     }
336     return r;
337 }
338
339 // 64-bit computation of a * b mod 2^64
340 uint64_t mul64( uint64_t a, uint64_t b ) {
341
342     uint64_t r = 0;
343     uint64_t t;
344
345     while ( a != 0 ) {
346
347         if ( a & 1 ) r += b;
348         a >>= 1;
349

```

```

350     t = b;
351     b += t;
352 }
353 return r;
354 }
355
356 // 64-bit computation of a^n mod m
357 uint64_t pow_mod64( uint64_t a, uintmax_t n, uint64_t m ) {
358
359     if ( n == 0 ) return 1;
360     if ( n == 1 ) return a % m;
361
362     uint64_t r = 1;
363     uint64_t t = a;
364
365     for (;;) {
366         if ( n & 1 ) r = mul_mod64( r, t, m );
367         n >>= 1;
368         if ( n == 0 ) break;
369         t = mul_mod64( t, t, m );
370     }
371     return r;
372 }
373
374 // 64-bit computation of a^n mod m, where n = 2^e + c
375 uint64_t pow_mod64( uint64_t a, uintmax_t e, uintmax_t c, uint64_t m ) {
376
377     if ( e == 0 ) return pow_mod64( a, c + 1, m );
378     uint64_t t = a;
379     for ( uint64_t i = 0; i < e; ++i ) t = mul_mod64( t, t, m );
380     return mul_mod64( pow_mod64( a, c, m ), t, m );
381 }
382
383 // a^n mod 2^64
384 uint64_t pow64( uint64_t a, uintmax_t n ) {
385
386     uint64_t r = 1;
387     uint64_t t = a;
388
389     for (;;) {
390         if ( n & 1 ) r *= t;
391         n >>= 1;
392         if ( n == 0 ) break;
393         t *= t;
394     }
395     return r;
396 }
397
398 // a^n mod 2^64, where n = 2^e + c
399 uint64_t pow64( uint64_t a, uint64_t e, uint64_t c ) {
400
401     if ( e == 0 ) return pow64( a, c + 1 );
402     uint64_t t = a;
403     for ( uint64_t i = 0; i < e; ++i ) t = mul64( t, t );
404     return mul64( pow64( a, c ), t );
405 }
406
407 // 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
408 uint64_t gs_mod64( uint64_t a, uintmax_t n, uint64_t m ) {
409
410     if ( n == 0 ) return 0;
411
412     uint64_t t = a % m;
413     uint64_t p = 1;
414     uint64_t r = 0;
415
416     while ( n > 1 ) {
417         if ( n & 1 ) r = add_mod64( r, mul_mod64( p, pow_mod64( t, n - 1, m ), m ), m );
418         p = mul_mod64( p, add_mod64( 1, t, m ), m );
419         t = mul_mod64( t, t, m );
420         n >>= 1;
421     }
422     r = add_mod64( r, p, m );
423     return r;
424 }
425
426 // 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
427 uint64_t gs_mod64( uint64_t a, uint32_t e, uint32_t c, uint64_t m ) {
428
429     if ( e == 0 ) return gs_mod64( a, 1 + c, m );
430
431     uint64_t t = a;
432     uint64_t r = 1;
433
434     for ( uint32_t i = 0; i < e; ++i ) {
435         r = mul_mod64( r, add_mod64( 1, t, m ), m );
436         t = mul_mod64( t, t, m );
437     }
438     if ( c == 0 ) return r;
439
440     return add_mod64( r, mul_mod64( t, gs_mod64( a, c, m ), m ), m );
441 }
442
443 // 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^64
444 uint64_t gs64( uint64_t a, uintmax_t n ) {
445
446     if ( n == 0 ) return 0;
447     if ( n == 1 ) return 1;
448
449     uint64_t t = a;
450     uint64_t p = 1;
451     uint64_t r = 0;
452
453     while ( n > 1 ) {
454         if ( n & 1 ) r = add_mod64( r, mul_mod64( p, pow64( t, n - 1 ), m ), m );
455         p = mul_mod64( p, add_mod64( 1, t, m ), m );
456         t = mul_mod64( t, t, m );
457         n >>= 1;
458     }
459     r = add_mod64( r, p, m );
460     return r;
461 }

```



```

457 while ( n > 1 ) {
458     if ( n & 1 ) r += mul64( p, pow64( t, n - 1 ) );
459     p = mul64( p, 1 + t );
460     t = mul64( t, t );
461     n >>= 1;
462 }
463 r += p;
464 return r;
465 }
466
467 // 64-bit sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod 2^64, where n = 2^e + c
468 uint64_t gs64( uint64_t a, uint64_t e, uint64_t c ) {
469     if ( e == 0 ) return gs64( a, 1 + c );
470     uint64_t t = a;
471     uint64_t r = 1;
472     for ( uint32_t i = 0; i < e; ++i ) {
473         r = mul64( r, add64( 1, t ) );
474         t = mul64( t, t );
475     }
476     if ( c == 0 ) return r;
477     return add64( r, mul64( t, gs64( a, c ) ) );
478 }
479
480 #endif // UINT64_C
481
482 // compute a + b mod m, where a, b and m must be < 2^35
483 double add_mod( double a, double b, double m ) {
484     assert( a < TW035 && b < TW035 && m < TW035 );
485     double v = a + b;
486     uintmax_t a1;
487     if ( v >= TW053 || v <= -TW053 ) {
488         a1 = static_cast<uintmax_t>( a / TW017 );
489         a -= a1 * TW017;
490         v = a1;
491         a1 = static_cast<uintmax_t>( v / m );
492         v -= a1 * m;
493         v = v * TW017 + a + b;
494     }
495     a1 = static_cast<uintmax_t>( v / m );
496     if ( ( v -= a1 * m ) < 0. ) return v += m;
497     else return v;
498 }
499
500 // a * b mod m, where a, b, and m must be < 2^35
501 double mul_mod( double a, double b, double m ) {
502     assert( a < TW035 && b < TW035 && m < TW035 );
503     double v = a * b;
504     uintmax_t a1;
505     if ( v >= TW053 || v <= -TW053 ) {
506         a1 = static_cast<uintmax_t>( a / TW017 );
507         a -= a1 * TW017;
508         v = a1 * b;
509         a1 = static_cast<uintmax_t>( v / m );
510         v -= a1 * m;
511         v = v * TW017 + a * b;
512     }
513     a1 = static_cast<uintmax_t>( v / m );
514     if ( ( v -= a1 * m ) < 0. ) return v += m;
515     else return v;
516 }
517
518 // compute a^n mod m
519 double pow_mod( double a, uintmax_t n, double m ) {
520     if ( n == 0 ) return 1.;
521     if ( n == 1 ) return a;
522     double r = 1.;
523     double t = a;
524     for (;;) {
525         if ( n & 1 ) r = mul_mod( r, t, m );
526         n >>= 1;
527         if ( n == 0 ) break;
528         t = mul_mod( t, t, m );
529     }
530     return r;
531 }
532
533 // compute a^n mod m, where n = 2^e + c
534 uint64_t pow_mod( double a, uint32_t e, uintmax_t c, double m ) {
535     if ( e == 0 ) return pow_mod( a, c + 1, m );
536     double t = a;
537     for ( uint32_t i = 0; i < e; ++i ) t = mul_mod( t, t, m );
538     return mul_mod( pow_mod( a, c, m ), t, m );
539 }
540
541 // sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m
542 double gs_mod( double a, uintmax_t n, double m ) {
543     if ( n == 0 ) return 0.;
544     if ( n == 1 ) return 1.;
545 }

```

```

564     double t = a;
565     double p = 1.;
566     double r = 0.;
567
568     while ( n > 1 ) {
569         if ( n & 1 ) r = add_mod( r, mul_mod( p, pow_mod( t, n - 1, m ), m ), m );
570         p = mul_mod( p, 1. + t, m );
571         t = mul_mod( t, t, m );
572         n >>= 1;
573     }
574     r = add_mod( r, p, m );
575     return r;
576 }
577
578 // sum first n terms of geometric series: 1 + a + ... + a^(n-1) mod m, where n = 2^e + c
579 double gs_mod( double a, uint32_t e, uint32_t c, double m ) {
580     if ( e == 0 ) return gs_mod( a, 1 + c, m );
581
582     double t = a;
583     double r = 1;
584
585     for ( uint32_t i = 0; i < e; ++i ) {
586         r = mul_mod( r, add_mod( 1, t, m ), m );
587         t = mul_mod( t, t, m );
588     }
589     if ( c == 0 ) return r;
590
591     return add_mod( r, mul_mod( t, gs_mod( a, c, m ), m ), m );
592 }
593
594 }
595

```

Listing D-2. mod_math.cpp

```

1 // mod_math.cpp: modular math for both 32-bit and 64-bit exponentiation and multiply
2
3 #include "mod_math.h"
4 #include <iostream>
5 #include <bitset>
6 #include <cassert>
7 using namespace std;
8
9 static const uint32_t _LC_MULT = 69069ul;
10 static const uint32_t _LC_CONST = 12345ul;
11 static const uint32_t _LC_MULT_INV = 2783094533ul; //0xa5e2a705
12
13 static const uint64_t LC_MULT = 1490024343005336237ULL;
14 static const uint64_t LC_CONST = 123456789ULL;
15 static const uint64_t A_INV = 14241175500494512421ULL;
16
17 // jump back
18 uint32_t jump_back( uint32_t s1, uintmax_t n, double m ) {
19     double a = mul_mod( pow_mod( _LC_MULT_INV, n, m ), add_mod( s1, -_LC_CONST, m ), m );
20     double b = mul_mod( -_LC_CONST, gs_mod( _LC_MULT_INV, n, m ), m );
21     double c = add_mod( a, b, m );
22     return static_cast<uint32_t>( add_mod( _LC_CONST, c, m ) );
23 }
24
25 // jump back 64-bit
26 uint64_t jump_back64( uint64_t s1, uintmax_t n ) {
27     uint64_t p = mul64( pow64( A_INV, n ), s1 - LC_CONST );
28     uint64_t q = mul64( -LC_CONST, gs64( A_INV, n ) );
29     s1 = p + q + LC_CONST;
30     return s1;
31 }
32
33 // jump ahead n numbers
34 uint32_t jump_ahead( uint32_t s1, uintmax_t n, double m ) {
35     double a = mul_mod( pow_mod( _LC_MULT, n, m ), s1, m );
36     double b = mul_mod( _LC_CONST, gs_mod( _LC_MULT, n, m ), m );
37     return static_cast<uint32_t>( add_mod( a, b, m ) );
38 }
39
40 // jump ahead n numbers, where n = 2^e + c
41 uint32_t jump_ahead( uint32_t s1, uint32_t e, uint32_t c, double m ) {
42     double a = mul_mod( pow_mod( _LC_MULT, e, c, m ), s1, m );
43     double b = mul_mod( _LC_CONST, gs_mod( _LC_MULT, e, c, m ), m );
44     return static_cast<uint32_t>( add_mod( a, b, m ) );
45 }
46
47 int main( void ) {
48
49 // test cases from Mathematica:
50 assert( jump_ahead( 123456789, 1024, TW032 ) == 3928303893 );
51 assert( jump_ahead( 123456789, 10, 0, TW032 ) == 3928303893 );
52 assert( jump_ahead( 123456789, 1000000, TW032 ) == 410693845 );
53 assert( jump_ahead( 123456789, 19, 475712, TW032 ) == 410693845 );
54 assert( jump_ahead( 123456789, 1000000000, TW032 ) == 4013060885 );
55 assert( jump_ahead( 123456789, 29, 463129088, TW032 ) == 4013060885 );
56 assert( jump_ahead( 123456789, 1073741824, TW032 ) == 3344682261 );
57 assert( jump_ahead( 123456789, 30, 0, TW032 ) == 3344682261 );
58 assert( jump_ahead( 0x7db7b9e0, 1024, TW032 ) == 3683370464 );
59 assert( jump_ahead( 0x7db7b9e0, 10, 0, TW032 ) == 3683370464 );
60 assert( jump_ahead( 0x7db7b9e0, 1000000, TW032 ) == 3408994464 );
61 assert( jump_ahead( 0x7db7b9e0, 19, 475712, TW032 ) == 3408994464 );
62 assert( jump_ahead( 0x7db7b9e0, 4294967295, TW032 ) == 1473225027 );
63 assert( jump_ahead( 0x7db7b9e0, 31, 2147483647, TW032 ) == 1473225027 );
64 assert( jump_ahead( 0x7db7b9e0, 4294967296ULL, TW032 ) == 2109192672 );
65 assert( jump_ahead( 0x7db7b9e0, 32, 0, TW032 ) == 2109192672 );
66 assert( jump_ahead( 0x7db7b9e0, 1099511627776ULL, TW032 ) == 2109192672 );
67
68 }
69

```

```

71 assert( jump_ahead( 0x7db7b9e0, 40, 0, TW032 ) == 2109192672 );
72
73 cout << "Test jump_ahead two different ways..." << endl;
74 uint32_t s0 = 0xcafe1234;
75 uint32_t s1 = s0; // initialize
76 uint32_t N = 1050;
77 uint32_t e = 10, c = 26;
78
79 for ( uint32_t i = 1; i <= N; i++ ) s1 = _LC_MULT * s1 + _LC_CONST;
80
81 assert( jump_ahead( s0, N, TW032 ) == s1 );
82 assert( jump_ahead( s0, e, c, TW032 ) == s1 );
83 cout << "Passed jump_ahead." << endl;
84 cout << "Test jump_back two different ways..." << endl;
85 assert( jump_back( s1, N, TW032 ) == s0 );
86 //assert( jump_back( s1, e, c, TW032 ) == s0 );
87 cout << "Passed jump_back." << endl;
88
89 cout << dec;
90
91 assert( gs_mod64( 123456789, 0, 4294967296 ) == 0 );
92 assert( gs_mod64( 123456789, 1, 4294967296 ) == 1 );
93 assert( gs_mod64( 123456789, 10, 4294967296 ) == 1382346382 );
94 assert( gs_mod64( 123456789, 1024, 4294967296 ) == 3101645824 );
95 assert( gs_mod64( 123456789, 1000000, 4294967296 ) == 2009531328 );
96
97 assert( gs64( 1490024343005336237, 0 ) == 0 );
98 assert( gs64( 1490024343005336237, 1 ) == 1 );
99 assert( gs64( 1490024343005336237, 10 ) == 7987679512244350278 );
100 assert( gs64( 1490024343005336237, 1024 ) == 9396580604419943424ULL );
101 assert( gs64( 1490024343005336237, 12345 ) == 2047449762047247049 );
102
103 // For a = 69069 and m = 2^32, a^(-1) is given by pow_mod64( 69069, 2147483647, 4294967296 )
104 assert( mul_mod64( 69069, pow_mod64( 69069, 2147483647, 4294967296 ), 4294967296 ) == 1 );
105 assert( pow_mod64( 69069, 2147483648, 4294967296 ) == 1 );
106
107 // For a = 314527869 and m = 2^32, a^(-1) is given by pow_mod64( 314527869, 2147483647, 4294967296 )
108 assert( mul_mod64( 314527869, pow_mod64( 314527869, 2147483647, 4294967296 ), 4294967296 ) == 1 );
109 assert( pow_mod64( 314527869, 2147483648, 4294967296 ) == 1 );
110
111 // For a = 1490024343005336237 and m = 2^64, a^(-1) is given by pow64( 1490024343005336237, 9223372036854775807 )
112 assert( mul64( 1490024343005336237, pow64( 1490024343005336237, 9223372036854775807 ) ) == 1 );
113 assert( pow64( 1490024343005336237, 9223372036854775808ULL ) == 1 );
114
115 // For a = 698769069 and m = 3001190298811367423, a^(-1) is given by pow_mod64( 698769069, 3001190298811367421, 3001190298811367423 )
116 assert( mul_mod64( 698769069, pow_mod64( 698769069, 3001190298811367421, 3001190298811367423 ), 3001190298811367423 ) == 1 );
117 assert( pow_mod64( 698769069, 3001190298811367422, 3001190298811367423 ) == 1 );
118
119 // For a = 4294584393 and m = 18445099517847011327, a^(-1) is given by pow_mod64( 4294584393, 18445099517847011325ULL, 18445099517847011327
120 // ULL )
121 assert( mul_mod64( 4294584393, pow_mod64( 4294584393, 18445099517847011325ULL, 18445099517847011327ULL ), 18445099517847011327ULL ) == 1 );
122 assert( pow_mod64( 4294584393, 18445099517847011326ULL, 18445099517847011327ULL ) == 1 );
123
124 // For a = 4246477509 and m = 18445099517847011327, a^(-1) is given by pow_mod64( 4246477509, 18445099517847011325ULL, 18445099517847011327
125 // ULL )
126 assert( mul_mod64( 4246477509, pow_mod64( 4246477509, 18445099517847011325ULL, 18445099517847011327ULL ), 18445099517847011327ULL ) == 1 );
127 assert( pow_mod64( 4246477509, 18445099517847011326ULL, 18445099517847011327ULL ) == 1 );
128
129 cout << dec;
130 cout << "Start tests..." << endl;
131 assert( add_mod32( 123456789, 987654321, 919 ) == 593 );
132 assert( add_mod32( 123456789, 987654321, 4294967295 ) == 1111111110 );
133 assert( add_mod64( 8446744073709551615ULL, 446744073709551615ULL, 18446744073709551ULL ) == 2157503891099648ULL );
134 assert( add_mod64( 8446744073709551615ULL, 446744073709551615ULL, 987654321 ) == 147440801 );
135 assert( add32( 4294967295ULL, 1 ) == 0 );
136 assert( add32( 4294967295ULL, 4294967295ULL ) == 4294967294 );
137 assert( add64( 18446744073709551615ULL, 1 ) == 0 );
138 assert( add64( 18446744073709551615ULL, 18446744073709551615ULL ) == 18446744073709551614ULL );
139
140 assert( mul_mod32( 123456789, 987654321, 919 ) == 379 );
141 assert( mul_mod32( 123456789, 987654321, 4294967295 ) == 4256203929 );
142 assert( mul_mod64( 8446744073709551615ULL, 446744073709551615ULL, 18446744073709551ULL ) == 714732902007009ULL );
143 assert( mul_mod64( 8446744073709551615ULL, 446744073709551615ULL, 987654321 ) == 906633840 );
144 assert( mul32( 4294967295ULL, 4294967295ULL ) == 1 );
145 assert( mul32( 123456789ULL, 987654321ULL ) == 4227814277 );
146 assert( mul64( 18446744073709551615ULL, 18446744073709551615ULL ) == 1 );
147 assert( mul64( 18446744073709551615ULL, 8446744073709551615ULL ) == 1000000000000000000ULL );
148
149 assert( pow_mod32( 123456789, 0, 123456 ) == 1 );
150 assert( pow_mod32( 123456789, 1, 123456 ) == 789 );
151 assert( pow_mod32( 123456789, 10, 123456 ) == 54681 );
152 assert( pow_mod32( 123456789, 100, 123456 ) == 30705 );
153 assert( pow_mod32( 123456789, 1000, 123456 ) == 18273 );
154 assert( pow_mod32( 123456789, 4294967296ULL, 123456 ) == 513 );
155 assert( pow_mod32( 123456789, 1, 987654321 ) == 123456789 );
156 assert( pow_mod32( 123456789, 0, 0, 987654321 ) == 123456789 );
157 assert( pow_mod32( 123456789, 2, 987654321 ) == 478395063 );
158 assert( pow_mod32( 123456789, 0, 1, 987654321 ) == 478395063 );
159
160 assert( pow_mod64( 446744073709551616, 1048576, 123456 ) == 34624 );
161 assert( pow_mod64( 446744073709551616, 20, 0, 123456 ) == 34624 );
162 assert( pow_mod64( 123456789, 1, 987654321 ) == 123456789 );
163 assert( pow_mod64( 123456789, 0, 0, 987654321 ) == 123456789 );
164 assert( pow_mod64( 123456789, 2, 987654321 ) == 478395063 );
165 assert( pow_mod64( 123456789, 0, 1, 987654321 ) == 478395063 );
166
167 assert( pow32( 123456789, 0 ) == 1 );
168 assert( pow32( 123456789, 1 ) == 123456789 );
169 assert( pow32( 123456789, 0, 0 ) == 123456789 );
170 assert( pow32( 123456789, 100 ) == 3584311345 );
171 assert( pow32( 123456789, 6, 36 ) == 3584311345 );
172 assert( pow32( 987654321, 317 ) == 133802993 );
173 assert( pow32( 123456789, 1048576 ) == 2092957697 );
174 assert( pow32( 123456789, 20, 0 ) == 2092957697 );
175 assert( pow32( 123456789, 19, 524288 ) == 2092957697 );
176
177 assert( pow64( 123456789, 1048576 ) == 16544794250596843521ULL );

```

```

176     assert( pow64( 123456789, 20, 0 ) == 16544794250596843521ULL );
177     assert( pow64( 123456789, 19, 524288 ) == 16544794250596843521ULL );
178
179     assert( gs_mod32( 123456789, 1000, 12345 ) == 4060 );
180     assert( gs32( 123456789, 1000 ) == 3030896216 );
181
182     assert( gs_mod64( 446744073709551616, 512, 987654321 ) == 852835532 );
183     assert( gs_mod64( 446744073709551616, 9, 0, 987654321 ) == 852835532 );
184
185     assert( gs64( 446744073709551616, 512 ) == 3452140635857354753ULL );
186     assert( gs64( 446744073709551616, 9, 0 ) == 3452140635857354753ULL );
187
188     assert( gs_mod64( 123456789, 64, 12345 ) == 3370 );
189     assert( gs_mod64( 123456789, 6, 0, 12345 ) == 3370 );
190     assert( gs_mod64( 123456789, 0, 63, 12345 ) == 3370 );
191
192     assert( gs_mod64( 123456789, 67, 12345 ) == 5446 );
193     assert( gs_mod64( 123456789, 6, 3, 12345 ) == 5446 );
194     assert( gs_mod64( 123456789, 0, 66, 12345 ) == 5446 );
195
196     assert( gs_mod64( 446744073709551616, 1024, 987654321 ) == 608654230 );
197     assert( gs_mod64( 446744073709551616, 10, 0, 987654321 ) == 608654230 );
198     assert( gs_mod64( 446744073709551616, 0, 1023, 987654321 ) == 608654230 );
199     assert( gs_mod64( 446744073709551616, 9, 512, 987654321 ) == 608654230 );
200     assert( gs_mod64( 446744073709551616, 8, 768, 987654321 ) == 608654230 );
201
202     assert( gs64( 446744073709551616, 1024 ) == 3452140635857354753ULL );
203     assert( gs64( 446744073709551616, 10, 0 ) == 3452140635857354753ULL );
204     assert( gs64( 446744073709551616, 0, 1023 ) == 3452140635857354753ULL );
205     assert( gs64( 446744073709551616, 9, 512 ) == 3452140635857354753ULL );
206     assert( gs64( 446744073709551616, 8, 768 ) == 3452140635857354753ULL );
207
208     assert( gs_mod32( 123456789, 1024, 12345 ) == 7165 );
209     assert( gs_mod32( 123456789, 10, 0, 12345 ) == 7165 );
210     assert( gs_mod32( 123456789, 0, 1023, 12345 ) == 7165 );
211     assert( gs_mod32( 123456789, 9, 512, 12345 ) == 7165 );
212     assert( gs_mod32( 123456789, 8, 768, 12345 ) == 7165 );
213
214     assert( gs32( 123456789, 1024 ) == 3101645824 );
215     assert( gs32( 123456789, 10, 0 ) == 3101645824 );
216     assert( gs32( 123456789, 0, 1023 ) == 3101645824 );
217     assert( gs32( 123456789, 9, 512 ) == 3101645824 );
218     assert( gs32( 123456789, 8, 768 ) == 3101645824 );
219
220     assert( gs_mod( 123456789, 64, 12345 ) == gs_mod64( 123456789, 64, 12345 ) );
221     assert( gs_mod( 123456789, 6, 0, 12345 ) == gs_mod64( 123456789, 6, 0, 12345 ) );
222     assert( gs_mod( 123456789, 0, 63, 12345 ) == gs_mod( 123456789, 0, 63, 12345 ) );
223
224     assert( pow_mod( 123456789, 1048576, 123456 ) == pow_mod64( 123456789, 1048576, 123456 ) );
225     assert( pow_mod( 123456789, 20, 0, 123456 ) == pow_mod64( 123456789, 20, 0, 123456 ) );
226
227     cout << "All tests passed." << endl;
228
229     return 0;
230 }

```

Listing D-3. Bitmatrix.h

```

1  // Bitmatrix.h: template class for 32 x 32 or 64 x 64 matrices using mod 2 arithmetic
2  // R. Saucier, August 2016
3
4  #ifndef BITMATRIX_H
5  #define BITMATRIX_H
6
7  #include <stdint> // for uint32_t and uint64_t
8  #include <climits> // for CHAR_BIT, the number of bits per byte
9
10 typedef struct { uint32_t matrix[32]; } bitmatrix32_t;
11 typedef struct { uint64_t matrix[64]; } bitmatrix64_t;
12
13 template <class T>
14 class Bitmatrix {
15
16 public:
17
18     static const unsigned int N.BITS = CHAR_BIT * sizeof( T ); // number of bits
19
20 public:
21
22     Bitmatrix( void ) { // default constructor
23     }
24
25     Bitmatrix( const bitmatrix32_t& A ) { // constructor from array of 32-bit constants
26
27         for ( T i = 0; i < N.BITS; i++ ) _matrix[i] = A.matrix[i];
28     }
29
30     Bitmatrix( const bitmatrix64_t& A ) { // constructor from array of 64-bit constants
31
32         for ( T i = 0; i < N.BITS; i++ ) _matrix[i] = A.matrix[i];
33     }
34
35     ~Bitmatrix( void ) { // default destructor
36     }
37
38     Bitmatrix( const Bitmatrix& A ) { // copy constructor
39
40         for ( T i = 0; i < N.BITS; i++ ) _matrix[i] = A._matrix[i];
41     }
42
43     Bitmatrix& operator=( const Bitmatrix& A ) { // assignment operator
44
45         if ( this != &A ) for ( T i = 0; i < N.BITS; i++ ) _matrix[i] = A._matrix[i];
46         return *this;
47     }

```

```

48
49 void identity( Bitmatrix& A ) { // create an identity matrix
50
51     T v = T(1);
52     for ( T i = 0; i < N.BITS; i++, v <<= 1 ) A._matrix[i] = v;
53 }
54
55 T matrix( T i ) { // return the ith vector of the bitmatrix
56
57     return _matrix[i];
58 }
59
60 // overloaded operators
61
62 friend T operator*( const Bitmatrix<T>& A, T v ) { // matrix multiplication of a vector
63
64     T r = T(0);
65     T b = T(1);
66     for ( T i = 0; i < N.BITS; i++, v >>= 1 ) if ( v & b ) r ^= A._matrix[i];
67     return r;
68 }
69
70 friend Bitmatrix operator*( Bitmatrix<T>& A, const Bitmatrix<T>& B ) { // multiplication of two Bitmatrices
71
72     Bitmatrix<T> C;
73
74     for ( T i = 0; i < N.BITS; i++ ) C._matrix[i] = A * B._matrix[i];
75     return C;
76 }
77
78 Bitmatrix& operator*=( const Bitmatrix<T>& A ) { // multiplication assignment of two Bitmatrices
79
80     return *this = *this * A;
81 }
82
83 Bitmatrix operator^( T n ) { // return A^n, Bitmatrix A to the power n
84
85     Bitmatrix<T> B, A = *this;
86
87     identity( B );
88     T b = T(1);
89
90     while ( n > 0 ) { // Knuth's "exponentiation by squaring" algorithm
91
92         if ( n & b ) B *= A;
93         A *= A;
94         n >>= 1;
95     }
96     return B;
97 }
98
99 friend Bitmatrix pow( Bitmatrix<T>& A, T e, T c ) { // return A^n, Bitmatrix A to the power n, where n = 2^e + c
100
101     Bitmatrix<T> B;//, A = *this;
102     if ( e > 0 ) {
103         B = A;
104         for ( T i = 0; i < e; i++ ) B *= B;
105     }
106     A = A^c;
107     if ( e ) A *= B;
108     return A;
109 }
110
111 private:
112     T _matrix[N.BITS];
113 };
114
115 // declaration of friends
116 //void identity( Bitmatrix<uint32_t>& A );
117 uint32_t operator*( const Bitmatrix<uint32_t>& A, uint32_t v );
118 Bitmatrix<uint32_t> operator*( Bitmatrix<uint32_t>& A, const Bitmatrix<uint32_t>& B );
119 Bitmatrix<uint32_t> pow( Bitmatrix<uint32_t>& A, uint32_t e, uint32_t c );
120
121 //void identity( Bitmatrix<uint64_t>& A );
122 uint64_t operator*( const Bitmatrix<uint64_t>& A, uint32_t v );
123 Bitmatrix<uint64_t> operator*( Bitmatrix<uint64_t>& A, const Bitmatrix<uint64_t>& B );
124
125 #endif // BITMATRIX_H
126

```

Listing D-4. lfsr88.h

```

1 // lfsr88.h: 32-bit Random number generator U[0,1): lfsr88
2 // Cycle length is (2^31 - 1)(2^29 - 1)(2^28 - 1) = 309485007947847626691444735 or approximately 2^88
3 // Author: Pierre L'Ecuier,
4 // Source: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tausme2.ps
5 // R. Saucier, December 2016
6
7 #ifndef LFSR88_H
8 #define LFSR88_H
9
10 namespace LFSR88 {
11
12 static const uint32_t N_SEEDS = 3;
13 static const bitmatrix32_t MATRIX[N_SEEDS] = {
14     {
15         {
16             0x00000000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040001, 0x00080002,
17             0x00100004, 0x00200008, 0x00400010, 0x00800020, 0x01000040, 0x02000080, 0x04000100, 0x08000200,
18             0x10000400, 0x20000800, 0x40001000, 0x80000001, 0x00000002, 0x00000004, 0x00000008, 0x00000010,
19             0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800, 0x00001000
20         },
21     },
22     {
23

```

```

24         0x00000000, 0x00000000, 0x00000000, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800,
25         0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000, 0x00080000,
26         0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x04000000, 0x08000001,
27         0x10000002, 0x20000005, 0x4000000A, 0x80000014, 0x00000028, 0x00000050, 0x00000020, 0x00000040
28     },
29 },
30 {
31     {
32         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00200000, 0x00400000, 0x00800000, 0x01000000,
33         0x02000001, 0x04000002, 0x08000004, 0x10000009, 0x20000012, 0x40000024, 0x80000048, 0x00000090,
34         0x00000120, 0x00000240, 0x00000480, 0x00000900, 0x00001200, 0x00002400, 0x00004800, 0x00009000,
35         0x00012000, 0x00024000, 0x00048000, 0x00090000, 0x00120000, 0x00040000, 0x00080000, 0x00100000
36     }
37 }
38 };
39 static const bitmatrix32_t MATRIX_INV[N_SEEDS] = {
40 {
41     {
42         0x00000000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x04000000,
43         0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000001, 0x00000002, 0x00000004, 0x00000008,
44         0x00000010, 0x00000020, 0x00000040, 0x00100080, 0x00200100, 0x00400200, 0x00800400, 0x01000800,
45         0x02001000, 0x04002000, 0x08004000, 0x10008000, 0x20010000, 0x40020000, 0x80040000, 0x00080000
46     }
47 },
48 {
49     {
50         0x00000000, 0x00000000, 0x00000000, 0x50000000, 0xa0000001, 0x40000002, 0x80000004, 0x00000008,
51         0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800,
52         0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000, 0x00080000,
53         0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x54000000, 0xa8000000
54     }
55 },
56 {
57     {
58         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x49248000, 0x92490000, 0x24920000, 0x49240000,
59         0x92480000, 0x24900000, 0x49200000, 0x92400000, 0x24800000, 0x49000000, 0x92000000, 0x24000000,
60         0x48000000, 0x90000001, 0x20000002, 0x40000004, 0x80000008, 0x00000010, 0x00000020, 0x00000040,
61         0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800, 0x49249000, 0x92492000, 0x24924000
62     }
63 }
64 };
65 static const uint32_t C0 = 0xfffffffful; // 4294967295ul
66 static const uint32_t C1 = 0xfffffffful; // 4294967294ul
67 static const uint32_t C2 = 0xfffffffful; // 4294967288ul
68 static const uint32_t C3 = 0xfffffffful; // 4294967280ul
69
70 class lfsr88 : public Generator<uint32_t> {
71
72 public:
73     lfsr88 ( void ) { // default constructor
74     }
75
76     lfsr88 ( std::vector<uint32_t> seed ) { // constructor from vector seed
77
78         setState( seed );
79     }
80
81     virtual ~lfsr88() { // default destructor
82
83         std::cout << "deleting lfsr88" << std::endl;
84     }
85
86     virtual void setState( std::vector<uint32_t> seed ) { // set the seeds
87
88         assert( seed.size() >= N_SEEDS );
89
90         // VERY IMPORTANT: The initial seeds _s[0], _s[1], _s[2] MUST be larger than 1, 7, and 15 respectively
91         _s[0] = seed[0]; if ( _s[0] < 2 ) _s[0] += 2;
92         _s[1] = seed[1]; if ( _s[1] < 8 ) _s[1] += 8;
93         _s[2] = seed[2]; if ( _s[2] < 16 ) _s[2] += 16;
94     }
95
96     virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector
97
98         assert( seed.size() >= N_SEEDS );
99         for ( size_t i = 0; i < N_SEEDS; i++ ) seed[i] = _s[i];
100     }
101
102     virtual void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers
103
104         for ( size_t i = 0; i < N_SEEDS; i++ ) {
105
106             Bitmatrix<uint32_t> A( MATRIX[i] );
107             _s[i] = ( A^n ) * _s[i];
108         }
109     }
110
111     virtual void jump_ahead( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
112
113         if ( e == 0 && c == 0 ) return jump_ahead( 1 );
114
115         Bitmatrix<uint32_t> A, B;
116
117         for ( size_t i = 0; i < N_SEEDS; i++ ) {
118
119             if ( e ) {
120                 B = MATRIX[i];
121                 for ( uintmax_t j = 0; j < e; j++ ) B *= B;
122             }
123             A = MATRIX[i];
124             A = A^c;
125             if ( e ) A *= B;
126             _s[i] = A * _s[i];
127         }
128     }
129
130     virtual void jump_back( uintmax_t n ) { // jump ahead the next n random numbers

```

```

131
132     for ( size_t i = 0; i < N_SEEDS; i++ ) {
133
134         Bitmatrix<uint32_t> A( MATRIX_INV[i] );
135         _s[i] = ( A^n ) * _s[i];
136     }
137 }
138
139 virtual void jump_back( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
140
141     if ( e == 0 && c == 0 ) return jump_back( 1 );
142
143     Bitmatrix<uint32_t> A, B;
144
145     for ( size_t i = 0; i < N_SEEDS; i++ ) {
146
147         if ( e ) {
148             B = MATRIX_INV[i];
149             for ( uintmax_t j = 0; j < e; j++ ) B *= B;
150         }
151         A = MATRIX_INV[i];
152         A = A^c;
153         if ( e ) A *= B;
154         _s[i] = A * _s[i];
155     }
156 }
157
158 virtual void jump_cycle( void ) {
159
160     const uint32_t A = 31, B = 29, C = 28;
161     jump_ahead( A + B + C, 0 );
162     jump_back( A + B, 0 );
163     jump_back( A + C, 0 );
164     jump_back( B + C, 0 );
165     jump_ahead( A, 0 );
166     jump_ahead( B, 0 );
167     jump_ahead( C, 0 );
168     jump_back( 1 );
169 }
170
171 virtual uint32_t rng32( void ) { // returns 32-bit integer
172
173     _s[0] = ( ( _s[0] & C1 ) << 12 ) ^ ( ( ( _s[0] << 13 ) ^ _s[0] ) >> 19 );
174     _s[1] = ( ( _s[1] & C2 ) << 4 ) ^ ( ( ( _s[1] << 2 ) ^ _s[1] ) >> 25 );
175     _s[2] = ( ( _s[2] & C3 ) << 17 ) ^ ( ( ( _s[2] << 3 ) ^ _s[2] ) >> 11 );
176
177     return ( _s[0] ^ _s[1] ^ _s[2] ) & C0;
178 }
179
180 virtual uint64_t rng64( void ) { // returns 64-bit integer
181
182     uint64_t low = rng32();
183     uint64_t high = rng32();
184     return low | ( high << 32 );
185 }
186
187 virtual double rng32_01( void ) { // returns a double in [0,1)
188
189     return double( rng32() ) * TWO32_INV;
190 }
191
192 virtual long double rng64_01( void ) { // returns a long double in [0,1)
193
194     return double( rng64() ) * TWO64_INV;
195 }
196
197 private:
198     uint32_t _s[ N_SEEDS ];
199 }; // end lfsr88 class
200 } // end namespace LFSR88
201
202 #endif // LFSR88_H
203
204

```

Listing D-5. lfsr113.h

```

1 // 32-bit Random number generator U[0,1): lfsr113
2 // Period is (2^31 - 1)(2^29 - 1)(2^28 - 1)(2^25 - 1) = 1038459334472050478831840650870785 or approximately 2^113
3 // Author: Pierre L'Ecuier,
4 // Source: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/tausme2.ps
5 // R. Saucier, December 2016
6
7 #ifndef LFSR113_H
8 #define LFSR113_H
9
10 namespace LFSR113 {
11
12     static const uint32_t N_SEEDS = 4;
13     static const bitmatrix32_t MATRIX[N_SEEDS] = {
14     {
15         {
16             0x00000000, 0x00080000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000001,
17             0x04000002, 0x08000004, 0x10000008, 0x20000010, 0x40000020, 0x80000041, 0x00000082, 0x00000104,
18             0x00000208, 0x00000410, 0x00000820, 0x00001040, 0x00002080, 0x00004100, 0x00008200, 0x00010400,
19             0x00020800, 0x00041000, 0x00082000, 0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000
20         },
21     },
22     {
23         {
24             0x00000000, 0x00000000, 0x00000000, 0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200,
25             0x00000400, 0x00000800, 0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x00010000, 0x00020000,
26             0x00040000, 0x00080000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000, 0x02000000,
27             0x04000000, 0x08000001, 0x10000002, 0x20000005, 0x4000000a, 0x80000014, 0x00000008, 0x00000010
28         }
29     }
30 }

```

```

29     },
30     {
31     {
32         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00001000, 0x00002000, 0x00004000,
33         0x00008001, 0x00010002, 0x00020004, 0x00040008, 0x00080010, 0x00100020, 0x00200040, 0x00400080,
34         0x00800100, 0x01000200, 0x02000400, 0x04000800, 0x08000000, 0x10000001, 0x20000002, 0x40000004,
35         0x80000008, 0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400
36     }
37     },
38     {
39     {
40         0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00100000,
41         0x00200000, 0x00400001, 0x00800002, 0x01000004, 0x02000009, 0x04000012, 0x08000024, 0x10000048,
42         0x20000090, 0x40000120, 0x80000240, 0x00000480, 0x00000900, 0x00001200, 0x00002400, 0x00004800,
43         0x00009000, 0x00012000, 0x00024000, 0x00048000, 0x00090000, 0x00200000, 0x00400000, 0x00800000
44     }
45     }
46 };
47 static const bitmatrix32_t MATRIX_INV[N_SEEDS] = {
48 {
49 {
50     0x00000000, 0x04104000, 0x08208000, 0x10410000, 0x20820000, 0x41040000, 0x82080000, 0x04100000,
51     0x08200000, 0x10400000, 0x20800000, 0x41000000, 0x82000000, 0x04000000, 0x08000000, 0x10000000,
52     0x20000000, 0x40000000, 0x80000001, 0x00000002, 0x00000004, 0x00000008, 0x00000010, 0x00000020,
53     0x00000040, 0x00000080, 0x04104100, 0x08208200, 0x10410400, 0x20820800, 0x41041000, 0x82082000
54 },
55 },
56 {
57 {
58     0x00000000, 0x00000000, 0x00000000, 0x40000002, 0x80000004, 0x00000008, 0x00000010, 0x00000020,
59     0x00000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400, 0x00000800, 0x00001000, 0x00002000,
60     0x00004000, 0x00008000, 0x00010000, 0x00020000, 0x00040000, 0x00080000, 0x00100000, 0x00200000,
61     0x00400000, 0x00800000, 0x01000000, 0x02000000, 0x04000000, 0x08000001, 0x50000000, 0xa0000001
62 },
63 },
64 {
65 {
66     0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x02000000, 0x04000000, 0x08000000, 0x10000001,
67     0x20000002, 0x40000004, 0x80000008, 0x00000010, 0x00000020, 0x00000040, 0x00000080, 0x00000100,
68     0x00000200, 0x00000400, 0x00000800, 0x00010000, 0x00020000, 0x00040000, 0x00080000, 0x00100000,
69     0x00200000, 0x00400000, 0x00800000, 0x00100000, 0x00200000, 0x00400000, 0x00800000, 0x01000000
70 },
71 },
72 {
73 {
74     0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x00000000, 0x92480000,
75     0x24900000, 0x49200000, 0x92400000, 0x24800000, 0x49000000, 0x92000001, 0x24000002, 0x48000004,
76     0x90000008, 0x20000010, 0x40000020, 0x80000040, 0x00000080, 0x00000100, 0x00000200, 0x00000400,
77     0x00000800, 0x00001000, 0x00002000, 0x00004000, 0x00008000, 0x92490000, 0x24920000, 0x49240000
78 },
79 }
80 };
81 static const uint32_t C0 = 0xfffffffful; // 4294967295ul
82 static const uint32_t C1 = 0xfffffffful; // 4294967294ul
83 static const uint32_t C2 = 0xfffffffful; // 4294967288ul
84 static const uint32_t C3 = 0xfffffffful; // 4294967280ul
85 static const uint32_t C4 = 0xfffffffful; // 4294967168ul
86
87 class lfsr113 : public Generator<uint32_t> {
88
89 public:
90     lfsr113 ( void ) { // default constructor
91     }
92
93     lfsr113 ( std::vector<uint32_t> seed ) { // constructor from vector seed
94
95         setState( seed );
96     }
97
98     virtual ~lfsr113() { // default destructor
99
100         std::cout << "deleting lfsr113" << std::endl;
101     }
102
103     virtual void setState( std::vector<uint32_t> seed ) { // set the seeds
104
105         assert( seed.size() >= N_SEEDS );
106
107         // VERY IMPORTANT: The initial seeds _s1, _s2, _s3 MUST be larger than 1, 7, 15, and 127 respectively
108         _s[0] = seed[0]; if ( _s[0] < 2 ) _s[0] += 2;
109         _s[1] = seed[1]; if ( _s[1] < 8 ) _s[1] += 8;
110         _s[2] = seed[2]; if ( _s[2] < 16 ) _s[2] += 16;
111         _s[3] = seed[3]; if ( _s[3] < 128 ) _s[3] += 128;
112     }
113
114     virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector
115
116         assert( seed.size() >= N_SEEDS );
117         for ( size_t i = 0; i < N_SEEDS; i++ ) seed[i] = _s[i];
118     }
119
120     virtual void jump_ahead( uintmax_t n ) { // jumps ahead the next n random numbers
121
122         for ( size_t i = 0; i < N_SEEDS; i++ ) {
123
124             Bitmatrix<uint32_t> A( MATRIX[i] );
125             _s[i] = ( A^n ) * _s[i];
126         }
127     }
128
129     virtual void jump_ahead( uintmax_t e, uintmax_t c ) { // jumps ahead the next n random numbers, where n = 2^e + c
130
131         if ( e == 0 && c == 0 ) jump_ahead( 1 );
132
133         Bitmatrix<uint32_t> A, B;
134
135         for ( size_t i = 0; i < N_SEEDS; i++ ) {

```



```

136
137     if ( e ) {
138         B = MATRIX[i];
139         for ( uintmax_t j = 0; j < e; j++ ) B *= B;
140     }
141     A = MATRIX[i];
142     A = A^c;
143     if ( e ) A *= B;
144     _s[i] = A * _s[i];
145 }
146 }
147
148 virtual void jump_back( uintmax_t n ) { // jumps ahead the next n random numbers
149
150     for ( size_t i = 0; i < N_SEEDS; i++ ) {
151
152         Bitmatrix<uint32_t> A( MATRIX_INV[i] );
153         _s[i] = ( A^n ) * _s[i];
154     }
155 }
156
157 virtual void jump_back( uintmax_t e, uintmax_t c ) { // jumps ahead the next n random numbers, where n = 2^e + c
158
159     if ( e == 0 && c == 0 ) return jump_back( 1 );
160
161     Bitmatrix<uint32_t> A, B;
162
163     for ( size_t i = 0; i < N_SEEDS; i++ ) {
164
165         if ( e ) {
166             B = MATRIX_INV[i];
167             for ( uintmax_t j = 0; j < e; j++ ) B *= B;
168         }
169         A = MATRIX_INV[i];
170         A = A^c;
171         if ( e ) A *= B;
172         _s[i] = A * _s[i];
173     }
174 }
175
176 virtual void jump_cycle( void ) { // jump ahead an entire cycle of lfsr113
177
178     const uint32_t A = 31, B = 29, C = 28, D = 25;
179     jump_ahead( A + B + C + D, 0 );
180     jump_back( A + B + C, 0 );
181     jump_back( A + B + D, 0 );
182     jump_back( A + C + D, 0 );
183     jump_back( B + C + D, 0 );
184     jump_ahead( A + B, 0 );
185     jump_ahead( A + C, 0 );
186     jump_ahead( A + D, 0 );
187     jump_ahead( B + C, 0 );
188     jump_ahead( B + D, 0 );
189     jump_ahead( C + D, 0 );
190     jump_back( A, 0 );
191     jump_back( B, 0 );
192     jump_back( C, 0 );
193     jump_back( D, 0 );
194     jump_ahead( 1 );
195 }
196
197 virtual uint32_t rng32( void ) { // returns the next number (a 32-bit unsigned int)
198
199     _s[0] = ( ( _s[0] & C1 ) << 18 ) ^ ( ( ( _s[0] << 6 ) ^ _s[0] ) >> 13 );
200     _s[1] = ( ( ( _s[1] & C2 ) << 2 ) ^ ( ( _s[1] << 2 ) ^ _s[1] ) >> 27 );
201     _s[2] = ( ( _s[2] & C3 ) << 7 ) ^ ( ( ( _s[2] << 13 ) ^ _s[2] ) >> 21 );
202     _s[3] = ( ( _s[3] & C4 ) << 13 ) ^ ( ( ( _s[3] << 3 ) ^ _s[3] ) >> 12 );
203
204     return ( _s[0] ^ _s[1] ^ _s[2] ^ _s[3] ) & C0;
205 }
206
207 virtual uint64_t rng64( void ) { // returns 64-bit integer
208
209     uint64_t low = rng32();
210     uint64_t high = rng32();
211     return low | ( high << 32 );
212 }
213
214 virtual double rng32_01( void ) { // returns a double int the half-open interval [0,1)
215
216     return double( rng32() ) * TW032_INV;
217 }
218
219 virtual long double rng64_01( void ) { // returns a long double in [0,1)
220
221     return double( rng64() ) * TW064_INV;
222 }
223
224 private:
225     uint32_t _s[ N_SEEDS ];
226
227 }; // end lfsr113 class
228 } // end namespace LFSR113
229
230 #endif // LFSR113_H

```

Listing D-6. kiss.h

```

1 // kiss.h: Marsaglia's Keep It Simple Stupid RNG,
2 // which consists of a combination of linear congruential, 3-shift-register, and multiply with carry.
3 // Period is (2^32)(2^32-1)(698769069(2^31)-1) = 27681094672891588090390813844460011520, approximately 2^124.
4 // R. Saucier, December 2016
5
6 #ifndef KISS_H

```

```

7 #define KISS.H
8 #include <bitset>
9
10 namespace KISS {
11
12     static const bitmatrix32_t MATRIX = {
13     {
14         0x00042021, 0x00084042, 0x00108084, 0x00210108, 0x00420231, 0x00840462, 0x010808C4, 0x02101188,
15         0x04202310, 0x08404620, 0x10808C40, 0x21011880, 0x42023100, 0x84046200, 0x0808C400, 0x10118800,
16         0x20231000, 0x40462021, 0x808C4042, 0x01080084, 0x02100108, 0x04200210, 0x08400420, 0x10800840,
17         0x21001080, 0x42002100, 0x84004200, 0x08008400, 0x10010800, 0x20021000, 0x40042000, 0x80084000
18     }
19     };
20     static const bitmatrix32_t MATRIX_INV = {
21     {
22         0xf2b58529, 0xe56b0a52, 0xd6b4a5, 0xbdad694a, 0x7b5ad294, 0xf6b5a528, 0xed6b4a50, 0xcd634a1,
23         0x9dac6942, 0x3b58d284, 0x76b1a508, 0xed634a10, 0xcec63421, 0x9d8c6842, 0x3b18d084, 0x7631a108,
24         0xec634210, 0xcc62421, 0x998c4842, 0x33189084, 0x66312108, 0xcc624210, 0x88c40420, 0x11880840,
25         0x23101080, 0x46202100, 0x8c404200, 0x08800400, 0x11000800, 0x22002100, 0x44004200, 0x88008400
26     }
27     };
28     static const uint32_t LC_MULT = 0x00010dcd; // 69069UL
29     static const uint32_t LC_CONST = 0x00003039; // 12345UL
30     static const uint32_t LC_MULT_INV = 0xa5e2a705; // 2783094533UL
31     static const uint64_t MWC_MULT = 0x0000000029a65ead; // 698769069ULL;
32     static const uint64_t MWC_MOD = 0x29a65eacffffffff; // 3001190298811367423ULL
33     static const uint64_t MWC_MULT_INV = 0x0000000010000000; // 4294967296ULL;
34     static const uint32_t N_SEEDS = 4; // requires four 32-bit seeds
35
36     class kiss : public Generator<uint32_t> {
37     public:
38         kiss( void ) { // default constructor
39         }
40
41         kiss( std::vector<uint32_t> seed ) { // constructor from seed vector
42
43             setState( seed );
44         }
45
46         virtual ~kiss() { // default destructor
47
48             std::cout << "deleting kiss" << std::endl;
49         }
50
51         virtual void setState( std::vector<uint32_t> seed ) { // set the seeds
52
53             assert( seed.size() >= N_SEEDS );
54             _s1 = seed[0];
55             _s2 = seed[1];
56             _s3 = seed[2];
57             _s4 = seed[3];
58         }
59
60         virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector
61
62             assert( seed.size() >= N_SEEDS );
63             seed[0] = _s1;
64             seed[1] = _s2;
65             seed[2] = _s3;
66             seed[3] = _s4;
67         }
68
69         virtual void jump_ahead( uintmax_t n ) { // jumps ahead the next n random numbers
70
71         #ifdef UINT64_C // use the native 64-bit capability
72
73             uint64_t p = mul_mod64( pow_mod64( LC_MULT, n, M ), _s1, M );
74             uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, n, M ), M );
75             _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );
76
77         #else // native 64-bit not available, so use double instead
78
79             double p = mul_mod( pow_mod( LC_MULT, n, M ), _s1, M );
80             double q = mul_mod( LC_CONST, gs_mod( LC_MULT, n, M ), M );
81             _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );
82
83         #endif // UINT64_C
84
85             Bitmatrix<uint32_t> A, B( MATRIX );
86             A = B^n;
87             _s2 = A * _s2;
88
89             uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
90             a = mul_mod64( pow_mod64( MWC_MULT, n, MWC_MOD ), a, MWC_MOD );
91             _s4 = ( uint32_t )( a >> 32u );
92             _s3 = ( uint32_t )( a );
93         }
94
95         virtual void jump_ahead( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
96
97             if ( e == 0 && c == 0 ) return jump_ahead( 1 );
98
99         #ifdef UINT64_C // use the native 64-bit capability
100
101             uint64_t p = mul_mod64( pow_mod64( LC_MULT, e, c, M ), _s1, M );
102             uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, e, c, M ), M );
103             _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );
104
105         #else // native 64-bit not available, so use double instead
106
107             double p = mul_mod( pow_mod( LC_MULT, e, c, M ), _s1, M );
108             double q = mul_mod( LC_CONST, gs_mod( LC_MULT, e, c, M ), M );
109             _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );
110
111         #endif // UINT64_C
112
113     };

```



```

221     return low | ( high << 32 );
222 }
223
224 virtual double rng32_01( void ) { // returns a random number in the half-open interval [0,1)
225
226     return double( rng32() ) * TW032_INV;
227 }
228
229 virtual long double rng64_01( void ) { // returns a long double in [0,1)
230
231     return double( rng64() ) * TW064_INV;
232 }
233
234 private:
235     uint32_t _s1, _s2, _s3, _s4;
236
237 }; // end kiss class
238 } // end namespace KISS
239
240
241 #endif // KISS_H

```

Listing D-7. jkiss.h

```

1 // jkiss.h: Based upon Marsaglia's Keep It Simple Stupid RNG
2 // Ref: Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications
3 // David Jones, UCL Bioinformatics Group (d.jones@cs.ucl.ac.uk), May 7, 2010
4 // Period is (2^32)(2^32-1)(4294584393(2^31)-1) = 170126015070308243410262827431100416 or approximately 2^127
5 // Period of MMC is 4294584393(2^31)-1 = 9222549758923505663
6 // R. Saucier, December 2016
7
8 #ifndef JKISS_H
9 #define JKISS_H
10
11 namespace JKISS {
12
13     static const bitmatrix32_t MATRIX = { // 32x32 bitmatrix
14     {
15         0x08400021, 0x10800042, 0x21400085, 0x4280010a, 0x85000214, 0x0a000428, 0x14000850, 0x284010a1,
16         0x50802142, 0xa1004284, 0x42008508, 0x84010a10, 0x08021420, 0x10042840, 0x20085080, 0x4010a100,
17         0x80214200, 0x00428400, 0x00850800, 0x010a1000, 0x02142000, 0x04284000, 0x08508000, 0x10a10000,
18         0x21420000, 0x42840000, 0x85080000, 0x08100000, 0x10200000, 0x20400000, 0x40800000, 0x81000000
19     }
20 };
21 // 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
22 static const bitmatrix32_t MATRIX_INV = {
23 {
24     0x9ce52d63, 0x39ca5ac6, 0x7394b58c, 0xe7296b18, 0xce52d630, 0x9ca5ac60, 0xb5bdce1, 0xb4a73de3,
25     0x694e7bc6, 0xd29cf78c, 0x5294a508, 0xa5294a10, 0x4a529420, 0x94a52840, 0x6b5ad4a1, 0xd6b5a942,
26     0xad6b5284, 0x5ad6a508, 0xb5ad4a10, 0x6b5a9420, 0xd6b52840, 0xef7ad4a1, 0xdef5a942, 0xbdeb5284,
27     0x7bdeb508, 0xf7ad4a10, 0xef5a9420, 0xbdeb5284, 0xff7ad4a1, 0xfef5a942, 0xfdeb5284, 0xfdb6a508
28 }
29 };
30 static const uint32_t LC_MULT = 0x12bf507dul; // 314527869ul;
31 static const uint32_t LC_CONST = 0x0012d687ul; // 1234567ul;
32 static const uint32_t LC_MULT_INV = 0x6200a8d5ul; // 1644210389ul;
33 static const uint64_t MMC_MULT = 0x00000000fffa2849ull; // 4294584393ull;
34 static const uint64_t MMC_MOD = 0xffffa2848ffffffffull; // 18445099517847011327ull;
35 static const uint64_t MMC_MULT_INV = 0x0000000100000000ull; // 4294967296ull;
36 static const uint64_t LC_PERIOD = 0x0000000100000000ull; // 4294967296ull;
37 static const uint64_t SR_PERIOD = 0x00000000ffffffffull; // 4294967295ull;
38 static const uint64_t MMC_PERIOD = 0x7ffd14247ffffffffull; // 9222549758923505663ull;
39 static const uint32_t N_SEEDS = 4; // requires four 32-bit words
40
41 class jkiss : public Generator<uint32_t> {
42
43 public:
44     jkiss( void ) { // default constructor
45     }
46
47     jkiss( std::vector<uint32_t> seed ) { // constructor from seed vector
48
49         setState( seed );
50     }
51
52     virtual ~jkiss() { // default destructor
53
54         std::cout << "deleting jkiss" << std::endl;
55     }
56
57     virtual void setState( std::vector<uint32_t> seed ) { // set the seeds
58
59         assert( seed.size() >= N_SEEDS );
60         _s1 = seed[0];
61         _s2 = seed[1];
62         _s3 = seed[2];
63         _s4 = seed[3];
64     }
65
66     virtual void getState( std::vector<uint32_t>& seed ) { // get the seed vector
67
68         assert( seed.size() >= N_SEEDS );
69         seed[0] = _s1;
70         seed[1] = _s2;
71         seed[2] = _s3;
72         seed[3] = _s4;
73     }
74
75     virtual void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers
76
77 #ifdef UINT64_C // use the native 64-bit capability
78
79         uint64_t p = mul_mod64( pow_mod64( LC_MULT, n, M ), _s1, M );
80         uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, n, M ), M );
81         _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );

```

```

82
83 //else // native 64-bit not available, so use double instead
84
85     double p = mul_mod( pow_mod( LC_MULT, n, M ), _s1, M );
86     double q = mul_mod( LC_CONST, gs_mod( LC_MULT, n, M ), M );
87     _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );
88
89 #endif // UINT64_C
90
91     Bitmatrix<uint32_t> A, B( MATRIX );
92     A = B^n;
93     _s2 = A * _s2;
94
95     uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
96     a = mul_mod64( pow_mod64( MWC_MULT, n, MWC_MOD ), a, MWC_MOD );
97     _s4 = ( uint32_t )( a >> 32u );
98     _s3 = ( uint32_t )( a );
99 }
100
101 virtual void jump_ahead( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
102
103     if ( e == 0 && c == 0 ) return jump_ahead( 1 );
104
105 #ifdef UINT64_C // use the native 64-bit capability
106
107     uint64_t p = mul_mod64( pow_mod64( LC_MULT, e, c, M ), _s1, M );
108     uint64_t q = mul_mod64( LC_CONST, gs_mod64( LC_MULT, e, c, M ), M );
109     _s1 = static_cast<uint32_t>( add_mod64( p, q, M ) );
110
111 #else // native 64-bit not available, so use double instead
112
113     double p = mul_mod( pow_mod( LC_MULT, e, c, M ), _s1, M );
114     double q = mul_mod( LC_CONST, gs_mod( LC_MULT, e, c, M ), M );
115     _s1 = static_cast<uint32_t>( add_mod( p, q, M ) );
116
117 #endif // UINT64_C
118
119     Bitmatrix<uint32_t> A, B;
120
121     if ( e ) {
122         B = MATRIX;
123         for ( size_t i = 0; i < e; i++ ) B *= B;
124     }
125     A = MATRIX;
126     A = A^c;
127     if ( e ) A *= B;
128     _s2 = A * _s2;
129
130     uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
131     a = mul_mod64( pow_mod64( MWC_MULT, e, c, MWC_MOD ), a, MWC_MOD );
132     _s4 = ( uint32_t )( a >> 32u );
133     _s3 = ( uint32_t )( a );
134 }
135
136 virtual void jump_back( uintmax_t n ) { // jump back the next n random numbers
137
138 #ifdef UINT64_C // use the native 64-bit capability
139
140     uint64_t p = mul_mod64( pow_mod64( LC_MULT_INV, n, M ), add_mod64( _s1, -LC_CONST, M ), M );
141     uint64_t q = mul_mod64( -LC_CONST, gs_mod64( LC_MULT_INV, n, M ), M );
142     uint64_t r = add_mod64( p, q, M );
143     _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );
144
145 #else // native 64-bit not available, so use double instead
146
147     double p = mul_mod( pow_mod( LC_MULT_INV, n, M ), add_mod( _s1, -LC_CONST, M ), M );
148     double q = mul_mod( -LC_CONST, gs_mod( LC_MULT_INV, n, M ), M );
149     double r = add_mod( p, q, M );
150     _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );
151
152 #endif // UINT64_C
153
154     Bitmatrix<uint32_t> A( MATRIX_INV );
155     A = A^n;
156     _s2 = A * _s2;
157
158     uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
159     a = mul_mod64( pow_mod64( MWC_MULT_INV, n, MWC_MOD ), a, MWC_MOD );
160     _s4 = ( uint32_t )( a >> 32u );
161     _s3 = ( uint32_t )( a );
162 }
163
164 virtual void jump_back( uintmax_t e, uintmax_t c ) { // jump back the next n random numbers, where n = 2^e + c
165
166     if ( e == 0 && c == 0 ) return jump_back( 1 );
167
168 #ifdef UINT64_C // use the native 64-bit capability
169
170     uint64_t p = mul_mod64( pow_mod64( LC_MULT_INV, e, c, M ), add_mod64( _s1, -LC_CONST, M ), M );
171     uint64_t q = mul_mod64( -LC_CONST, gs_mod64( LC_MULT_INV, e, c, M ), M );
172     uint64_t r = add_mod64( p, q, M );
173     _s1 = static_cast<uint32_t>( add_mod64( LC_CONST, r, M ) );
174
175 #else // native 64-bit not available, so use double instead
176
177     double p = mul_mod( pow_mod( LC_MULT_INV, e, c, M ), add_mod( _s1, -LC_CONST, M ), M );
178     double q = mul_mod( -LC_CONST, gs_mod( LC_MULT_INV, e, c, M ), M );
179     double r = add_mod( p, q, M );
180     _s1 = static_cast<uint32_t>( add_mod( LC_CONST, r, M ) );
181
182 #endif // UINT64_C
183
184     Bitmatrix<uint32_t> A, B;
185
186     if ( e ) {
187         B = MATRIX_INV;
188         for ( size_t i = 0; i < e; i++ ) B *= B;

```


[illegible]

[illegible]


```

153     setState( seed );
154 }
155
156 virtual ~lfsr258() { // default destructor
157
158     std::cout << "deleting lfsr258" << std::endl;
159 }
160
161 virtual void setState( std::vector<uint64_t> seed ) { // set the seeds
162
163     assert( seed.size() >= N_SEEDS );
164
165     // VERY IMPORTANT: The initial seeds s1, s2, s3 MUST be larger than 1, 511, 4095, 131071, and 8388607 respectively
166     s[0] = seed[0]; if ( s[0] < 2 ) s[0] += 2;
167     s[1] = seed[1]; if ( s[1] < 512 ) s[1] += 512;
168     s[2] = seed[2]; if ( s[2] < 4096 ) s[2] += 4096;
169     s[3] = seed[3]; if ( s[3] < 131072 ) s[3] += 131072;
170     s[4] = seed[4]; if ( s[4] < 8388608 ) s[4] += 8388608;
171 }
172
173 virtual void getState( std::vector<uint64_t>& seed ) { // get the seed vector
174
175     assert( seed.size() >= N_SEEDS );
176     for ( size_t i = 0; i < N_SEEDS; i++ ) seed[i] = s[i];
177 }
178
179 virtual void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers
180
181     for ( size_t i = 0; i < N_SEEDS; i++ ) {
182
183         Bitmatrix<uint64_t> A( MATRIX[i] );
184         s[i] = ( A^n ) * s[i];
185     }
186 }
187
188 virtual void jump_ahead( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
189
190     if ( e == 0 && c == 0 ) return jump_ahead( 1 );
191
192     Bitmatrix<uint64_t> A;
193
194     for ( size_t i = 0; i < N_SEEDS; i++ ) {
195
196         if ( e ) {
197             B = MATRIX[i];
198             for ( uintmax_t j = 0; j < e; j++ ) B *= B;
199         }
200         A = MATRIX[i];
201         A = A^c;
202         if ( e ) A *= B;
203         s[i] = A * s[i];
204     }
205 }
206
207 virtual void jump_back( uintmax_t n ) { // jump ahead the next n random numbers
208
209     for ( size_t i = 0; i < N_SEEDS; i++ ) {
210
211         Bitmatrix<uint64_t> A( MATRIX_INV[i] );
212         s[i] = ( A^n ) * s[i];
213     }
214 }
215
216 virtual void jump_back( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
217
218     if ( e == 0 && c == 0 ) return jump_back( 1 );
219
220     Bitmatrix<uint64_t> A;
221
222     for ( size_t i = 0; i < N_SEEDS; i++ ) {
223
224         if ( e ) {
225             B = MATRIX_INV[i];
226             for ( uintmax_t j = 0; j < e; j++ ) B *= B;
227         }
228         A = MATRIX_INV[i];
229         A = A^c;

```

```

230         if ( e ) A *= B;
231         _s[1] = A * _s[1];
232     }
233 }
234
235 virtual void jump_cycle( void ) { // jump ahead an entire cycle of lfsr258
236
237     const uint32_t A = 63, B = 55, C = 52, D = 47, E = 41;
238     jump_ahead( A + B + C + D + E, 0 );
239     jump_back( B + C + D + E, 0 );
240     jump_back( A + C + D + E, 0 );
241     jump_back( A + B + D + E, 0 );
242     jump_back( A + B + C + E, 0 );
243     jump_back( A + B + C + D, 0 );
244     jump_ahead( C + D + E, 0 );
245     jump_ahead( B + D + E, 0 );
246     jump_ahead( B + C + E, 0 );
247     jump_ahead( B + C + D, 0 );
248     jump_ahead( A + D + E, 0 );
249     jump_ahead( A + C + E, 0 );
250     jump_ahead( A + C + D, 0 );
251     jump_ahead( A + B + E, 0 );
252     jump_ahead( A + B + D, 0 );
253     jump_ahead( A + B + C, 0 );
254     jump_back( A + B, 0 );
255     jump_back( A + C, 0 );
256     jump_back( A + D, 0 );
257     jump_back( A + E, 0 );
258     jump_back( B + C, 0 );
259     jump_back( B + D, 0 );
260     jump_back( B + E, 0 );
261     jump_back( C + D, 0 );
262     jump_back( C + E, 0 );
263     jump_back( D + E, 0 );
264     jump_ahead( A, 0 );
265     jump_ahead( B, 0 );
266     jump_ahead( C, 0 );
267     jump_ahead( D, 0 );
268     jump_ahead( E, 0 );
269     jump_back( 1 );
270 }
271
272 uint32_t rng32( void ) { // returns the next random number as a 32-bit integer
273
274     return uint32_t( rng64() );
275 }
276
277 uint64_t rng64( void ) { // returns the next random number as a 64-bit integer
278
279     _s[0] = ( ( _s[0] & C1 ) << 10 ) ^ ( ( _s[0] << 1 ) ^ _s[0] ) >> 53 );
280     _s[1] = ( ( _s[1] & C2 ) << 5 ) ^ ( ( _s[1] << 24 ) ^ _s[1] ) >> 56 );
281     _s[2] = ( ( _s[2] & C3 ) << 29 ) ^ ( ( _s[2] << 3 ) ^ _s[2] ) >> 23 );
282     _s[3] = ( ( _s[3] & C4 ) << 23 ) ^ ( ( _s[3] << 5 ) ^ _s[3] ) >> 24 );
283     _s[4] = ( ( _s[4] & C5 ) << 8 ) ^ ( ( _s[4] << 3 ) ^ _s[4] ) >> 33 );
284     return ( _s[0] ^ _s[1] ^ _s[2] ^ _s[3] ^ _s[4] ) & C0;
285 }
286
287 double rng32_01( void ) { // returns a random number in the half-open interval [0,1)
288
289     return rng32() * TWO32_INV;
290 }
291
292 long double rng64_01( void ) { // returns a random number in the half-open interval [0,1)
293
294     return rng64() * TWO64_INV;
295 }
296
297 private:
298
299     uint64_t _s[ N_SEEDS ];
300 }; // end lfsr258 class
301 } // end namespace LFSR258
302 #endif // LFSR258_H

```

Listing D-9. ilkiss.h

```

1 // jkiss.h: Based upon Marsaglia's Keep It Simple Stupid RNG
2 // Ref: Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications
3 // David Jones, UCL Bioinformatics Group (d.jones@ucl.ac.uk), May 7, 2010
4 // Cycle length is (2^64-1)/(4294584393*(2^31)-1) = 313871016102609240474418568623031094853687768430673920,
5 // or approximately 2^91
6 // Period of MWC is 4294584393*(2^31)-1 = 9222549758923505653
7 // R. Saucier, December 2016
8
9 #ifndef JUKTSS_H
10 #define JUKTSS_H
11
12 namespace JUKTSS {
13
14     static const bitmatrix64_t MATRIX = {
15         {
16             0x0608000440270011, 0x0610000880400022, 0x0070001100800044, 0x0640007201000088, 0x0680004402700110, 0x0100088040002700, 0x0270001100800044, 0x0640007201000088,
17             0x0680004402700110, 0x10088040002700, 0x10088040002700, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
18             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
19             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
20             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
21             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
22             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
23             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
24             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
25         },
26     };
27
28     static const bitmatrix64_t MATRIX_INV = {
29         {
30             0x0608000440270011, 0x0610000880400022, 0x0070001100800044, 0x0640007201000088, 0x0680004402700110, 0x0100088040002700, 0x0270001100800044, 0x0640007201000088,
31             0x0680004402700110, 0x10088040002700, 0x10088040002700, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
32             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
33             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
34             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
35             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
36             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
37             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
38             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
39             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
40             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
41             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
42             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
43             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
44             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
45             0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001, 0x06800044027001,
46             0x06800044027001, 0x06800044027001, 
```

```

76 seed[0] = _s1;
77 seed[1] = _s2;
78 seed[2] = ( uint64_t( _s3 ) << 32 ) + _s4;
79 }
80
81 void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers
82
83     _s1 = mul64( pow64( LC.MULT, n ), _s1 ) + mul64( LC.CONST, gs64( LC.MULT, n ) );
84
85     Bitmatrix<uint64_t> A, B( MATRIX );
86     A = B^n;
87     _s2 = A * _s2;
88
89     uint64_t a = _s3 + ( (uint64_t)s4 << 32u );
90     a = mul_mod64( pow_mod64( MMC.MULT, n, MMC.MOD ), a, MMC.MOD );
91     _s4 = ( uint32_t ) ( a >> 32u );
92     _s3 = ( uint32_t ) ( a );
93 }
94
95 void jump_ahead( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
96
97     if ( e == 0 && c == 0 ) return jump_ahead( 1 );
98
99     _s1 = mul64( pow64( LC.MULT, e, c ), _s1 ) + mul64( LC.CONST, gs64( LC.MULT, e, c ) );
100
101     Bitmatrix<uint64_t> A, B;
102
103     if ( e ) {
104         B = MATRIX;
105         for ( uint64_t i = 0; i < e; i++ ) B *= B;
106     }
107     A = MATRIX;
108     A = A^c;
109     if ( e ) A *= B;
110     _s2 = A * _s2;
111
112     uint64_t a = _s3 + ( (uint64_t)s4 << 32u );
113     a = mul_mod64( pow_mod64( MMC.MULT, e, c, MMC.MOD ), a, MMC.MOD );
114     _s4 = ( uint32_t ) ( a >> 32u );
115     _s3 = ( uint32_t ) ( a );
116 }
117
118 void jump_back( uintmax_t n ) { // jump back the next n random numbers
119
120     _s1 = mul64( pow64( LC.MULT_INV, n ), _s1 - LC.CONST ) + LC.CONST - mul64( LC.CONST, gs64( LC.MULT_INV, n ) );
121
122     Bitmatrix<uint64_t> A, B( MATRIX_INV );
123     A = B^n;
124     _s2 = A * _s2;
125
126     uint64_t a = _s3 + ( (uint64_t)s4 << 32u );
127     a = mul_mod64( pow_mod64( MMC.MULT_INV, n, MMC.MOD ), a, MMC.MOD );
128     _s4 = ( uint32_t ) ( a >> 32u );
129     _s3 = ( uint32_t ) ( a );
130 }
131
132 void jump_back( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
133
134     if ( e == 0 && c == 0 ) return jump_back( 1 );
135
136     _s1 = mul64( pow64( LC.MULT_INV, e, c ), _s1 - LC.CONST ) + LC.CONST - mul64( LC.CONST, gs64( LC.MULT_INV, e, c ) );
137
138     Bitmatrix<uint64_t> A, B;
139
140     if ( e ) {
141         B = MATRIX_INV;
142         for ( uint64_t i = 0; i < e; i++ ) B *= B;
143     }
144     A = MATRIX_INV;
145     A = A^c;
146     if ( e ) A *= B;
147     _s2 = A * _s2;
148
149     uint64_t a = _s3 + ( (uint64_t)s4 << 32u );
150     a = mul_mod64( pow_mod64( MMC.MULT_INV, e, c, MMC.MOD ), a, MMC.MOD );
151     _s4 = ( uint32_t ) ( a >> 32u );
152     _s3 = ( uint32_t ) ( a );

```

```

153 }
154
155 virtual void jump_cycle( void ) { // jump ahead a full cycle of jkiss
156
157     std::bitset<191> p( std::string( "1111111111110100001010000100100011111111111111111" ) +
158     std::string( "11111111111101000000000000101101011101011010110000" ) +
159     std::string( "000000000000000000000010000000000000000000000000000" ) +
160     std::string( "00000000000000000000000000000000000000000000000000" ) );
161     for ( size_t i = 0; i < p.size(); ++i ) if ( p.test(i) ) jump_ahead( i, 0 );
162 }
163
164 uint32_t rng32( void ) {
165     _s1 = LC_MULT * _s1 + LC_CONST;
166
167     _s2 ^= ( _s2 << 21 ), _s2 ^= ( _s2 >> 17 ), _s2 ^= ( _s2 << 30 );
168
169     uint64_t a = MMC_MULT * _s3 + _s4;
170     _s4 = ( a >> 32u );
171     _s3 = uint32_t( a );
172
173     //return (uint32_t)_s1 + (uint32_t)_s2 + _s3; // this didn't work
174     //return (uint32_t)( _s1 >> 32 ) + (uint32_t)_s2 + _s3; // this doesn't seem to work
175     return _s1 + _s2 + _s3; // this passes all the tests, small crush, and big crush
176 }
177
178 uint64_t rng64( void ) {
179     _s1 = LC_MULT * _s1 + LC_CONST;
180
181     _s2 ^= ( _s2 << 21 ), _s2 ^= ( _s2 >> 17 ), _s2 ^= ( _s2 << 30 );
182
183     uint64_t a = MMC_MULT * _s3 + _s4;
184     _s4 = ( a >> 32u );
185     _s3 = uint32_t( a );
186
187     return _s1 + _s2 + ( (uint64_t)_s4 << 32 ) + _s3;
188 }
189
190 double rng32_01( void ) { // returns a random number in the half-open interval [0,1)
191     return double( rng32() ) * TW032_INV;
192 }
193
194 long double rng64_01( void ) { // returns a random number in the half-open interval [0,1)
195     return ( long double ) ( rng64() ) * TW064_INV;
196 }
197
198 private:
199     uint64_t _s1, _s2;
200     uint32_t _s3, _s4;
201
202     }; // end jkiss class
203     }; // end namespace JKISS
204
205 #endif // JKISS_H

```

Listing D-10. jkiss64.h

```

1 // jkiss64.h: Based upon Marsaglia's Keep It Simple Stupid RNG
2 // Ref: Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications
3 // David Jones, UCL Bioinformatics Group (d.jones@cs.ucl.ac.uk), May 7, 2010
4 // Period is (2^64)^(2^64-1) (4294584393 (2^31)-1) (698769069 (2^31)-1) = 470927433167567436556996170486797220343483431369386641683085078522096517120
5 // or approximately 2^251
6 // Requires four 64-bit seeds to initialize.
7 // R. Saucier, December 2016
8
9 #ifndef JKISS64_H
10 #define JKISS64_H
11
12 #include <iostream>
13 #include <iomanip>
14 using namespace std;

```

```

15 namespace jkKiss64 {
16
17
18
19     static const bitmatrix64_t MATRIX = {
20     {
21         0x0008000440200011, 0x0010000800400022, 0x0020001000800044, 0x0040002010000088, 0x0080004020000100, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
22         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
23         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
24         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
25         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
26         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
27         0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
28     }
29 };
30
31 static const bitmatrix64_t MATRIX_INV = {
32 {
33     0x90808c0404202201, 0x2101180808040402, 0x4202301010080804, 0x8404602021011008, 0x8808444402202001, 0x1100888004400220, 0x2201110088800440, 0x4402220111008880,
34     0x8804440220011008, 0x1008880440220011, 0x1100888044022001, 0x1100888044022001, 0x1100888044022001, 0x1100888044022001, 0x1100888044022001, 0x1100888044022001, 0x1100888044022001,
35     0x4402202101008880, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110,
36     0x4402202101008880, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110, 0x88040402200110,
37     0x2001100888000400, 0x40022001100888, 0x40022001100888, 0x40022001100888, 0x40022001100888, 0x40022001100888, 0x40022001100888, 0x40022001100888, 0x40022001100888,
38     0x01100888000400, 0x02200110088800, 0x02200110088800, 0x02200110088800, 0x02200110088800, 0x02200110088800, 0x02200110088800, 0x02200110088800, 0x02200110088800,
39     0x11008880040002, 0x22011100888004, 0x22011100888004, 0x22011100888004, 0x22011100888004, 0x22011100888004, 0x22011100888004, 0x22011100888004, 0x22011100888004,
40     }
41 };
42
43 static const uint64_t LC_MULT
44     = 0x14ad3ed78492ad; // 1490024343005336237ULL;
45 static const uint64_t LC_CONST
46     = 0x0000000075bcd15;
47 static const uint64_t LC_MULT_INV
48     = 0xc5a2d1aa2a78a125;
49 static const uint64_t MWC_MULT1
50     = 0x00000000ffffa2849; // 4294584393ULL;
51 static const uint64_t MWC_MULT1_INV
52     = 0xffffa2849fffff; // 1844509517847011327ULL;
53 static const uint64_t MWC_MULT2
54     = 0x0000000100000000; // 4294967296ULL;
55 static const uint64_t MWC_MULT2_INV
56     = 0x0000000029ab5e4d; // 69876969ULL;
57 static const uint64_t MWC_MULT2_INV
58     = 0x29ab5e4dffff; // 3801199298811367423ULL;
59 static const uint64_t MWC_MULT2_INV
60     = 0x0000000100000000; // 4294967296ULL;
61 static const uint32_t N_SEEDS
62     = 4; // requires 4 64-bit words
63
64 class jkKiss64 : public Generator<uint64_t> {
65 public:
66     jkKiss64( void ) { // default constructor
67     }
68     virtual ~jkKiss64() { // default destructor
69     }
70     std::cout << "deleting jkKiss64" << std::endl;
71
72     virtual void setState( std::vector<uint64_t> seed ) { // set the seeds from four 64-bit words
73     }
74     assert( seed.size() >= N_SEEDS );
75     -s1 = seed[0];
76     -s2 = seed[1];
77     -s3 = uint32_t( seed[2] >> 32 );
78     -s4 = uint32_t( seed[2] );
79     -s5 = uint32_t( seed[3] >> 32 );
80     -s6 = uint32_t( seed[3] );
81
82     virtual void getState( std::vector<uint64_t>& seed ) { // get the seed vector
83     }
84     assert( seed.size() >= N_SEEDS );
85     seed[0] = -s1;
86     seed[1] = -s2;
87     seed[2] = ( uint64_t( -s3 ) << 32 ) + -s4;
88     seed[3] = ( uint64_t( -s5 ) << 32 ) + -s6;
89
90     void jump_ahead( uintmax_t n ) { // jump ahead the next n random numbers
91     }
92     -s1 = mul64( pow64( LC_MULT, n ), -s1 ) + mul64( LC_CONST, gs64( LC_MULT, n ) );
93

```

```

92 Bitmatrix<uint64_t> A, B( MATRIX );
93 A = B^n;
94 _s2 = A * _s2;
95
96 uint64_t a = _s3 + ( (uint64_t)s4 << 32u );
97 a = mul_mod64( pow_mod64( MMC_MULT1, n, MMC_MOD1 ), a, MMC_MOD1 );
98 _s4 = ( (uint32_t) ( a >> 32u ) );
99 _s3 = ( (uint32_t) ( a ) );
100
101 a = _s5 + ( (uint64_t)s6 << 32u );
102 a = mul_mod64( pow_mod64( MMC_MULT2, n, MMC_MOD2 ), a, MMC_MOD2 );
103 _s6 = ( (uint32_t) ( a >> 32u ) );
104 _s5 = ( (uint32_t) ( a ) );
105 }
106
107 void jump_ahead( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
108
109
110 if ( e == 0 && c == 0 ) return jump_ahead( 1 );
111
112 _s1 = mul64( pow64( LC_MULT, e, c ), _s1 ) + mul64( LC_CONST, gs64( LC_MULT, e, c ) );
113
114 Bitmatrix<uint64_t> A, B;
115
116 if ( e ) {
117     B = MATRIX;
118     for ( uint64_t i = 0; i < e; i++ ) B *= B;
119
120     A = MATRIX;
121     A = A^c;
122     if ( e ) A *= B;
123     _s2 = A * _s2;
124
125     uint64_t a = _s3 + ( (uint64_t)s4 << 32u );
126     a = mul_mod64( pow_mod64( MMC_MULT1, e, c, MMC_MOD1 ), a, MMC_MOD1 );
127     _s4 = ( (uint32_t) ( a >> 32u ) );
128     _s3 = ( (uint32_t) ( a ) );
129
130     a = _s5 + ( (uint64_t)s6 << 32u );
131     a = mul_mod64( pow_mod64( MMC_MULT2, e, c, MMC_MOD2 ), a, MMC_MOD2 );
132     _s6 = ( (uint32_t) ( a >> 32u ) );
133     _s5 = ( (uint32_t) ( a ) );
134 }
135
136 void jump_back( uintmax_t n ) { // jump back the next n random numbers
137
138
139 _s1 = mul64( pow64( LC_MULT_INV, n ), _s1 - LC_CONST ) + LC_CONST - mul64( LC_CONST, gs64( LC_MULT_INV, n ) );
140
141 Bitmatrix<uint64_t> A, B( MATRIX_INV );
142 A = B^n;
143 _s2 = A * _s2;
144
145 uint64_t a = _s3 + ( (uint64_t)s4 << 32u );
146 a = mul_mod64( pow_mod64( MMC_MULT1_INV, n, MMC_MOD1 ), a, MMC_MOD1 );
147 _s4 = ( (uint32_t) ( a >> 32u ) );
148 _s3 = ( (uint32_t) ( a ) );
149
150 a = _s5 + ( (uint64_t)s6 << 32u );
151 a = mul_mod64( pow_mod64( MMC_MULT2_INV, n, MMC_MOD2 ), a, MMC_MOD2 );
152 _s6 = ( (uint32_t) ( a >> 32u ) );
153 _s5 = ( (uint32_t) ( a ) );
154 }
155
156 void jump_back( uintmax_t e, uintmax_t c ) { // jump ahead the next n random numbers, where n = 2^e + c
157
158
159 if ( e == 0 && c == 0 ) return jump_back( 1 );
160
161 _s1 = mul64( pow64( LC_MULT_INV, e, c ), _s1 - LC_CONST ) + LC_CONST - mul64( LC_CONST, gs64( LC_MULT_INV, e, c ) );
162
163 Bitmatrix<uint64_t> A, B;
164
165 if ( e ) {
166     B = MATRIX_INV;
167     for ( uint64_t i = 0; i < e; i++ ) B *= B;
168
169     A = MATRIX_INV;
170     A = A^c;
171 }

```

```

169 if ( e ) A *= B;
170 _s2 = A * _s2;
171
172 uint64_t a = _s3 + ( (uint64_t)_s4 << 32u );
173 a = mul_mod64( pow_mod64( MMC_MULT1_INV, e, c, MMC_MOD1 ), a, MMC_MOD1 );
174 _s4 = ( (uint32_t) ( a >> 32u ) );
175 _s3 = ( (uint32_t) ( a ) );
176
177 a = _s5 + ( (uint64_t)_s6 << 32u );
178 a = mul_mod64( pow_mod64( MMC_MULT2_INV, e, c, MMC_MOD2 ), a, MMC_MOD2 );
179 _s6 = ( (uint32_t) ( a >> 32u ) );
180 _s5 = ( (uint32_t) ( a ) );
181 }
182
183 virtual void jump_cycle( void ) { // jump ahead a full cycle of jlkiss64
184
185     std::bitset<252> p( std::string( "101001101001010101010101010101011001011100000" ) +
186         std::string( "011101010101000011000100110000101011111010110" ) +
187         std::string( "001101000111111000101000101001010000100001101" ) +
188         std::string( "1110101111111111111111111111111110000000000000" ) +
189         std::string( "00000000000000000000000000000000000000000000" ) +
190         std::string( "00" ) );
191     for ( size_t i = 0; i < p.size(); ++i ) if ( p.test(i) ) jump_ahead( i, 0 );
192 }
193
194 uint32_t rng32( void ) { // returns the next random number (as a 32-bit unsigned int)
195
196     return uint32_t( rng64() );
197 }
198
199 uint64_t rng64( void ) { // returns the next random number (as a 64-bit unsigned integer)
200
201     _s1 = LC_MULT * _s1 + LC_CONST;
202
203     _s2 ^= ( _s2 << 21 ), _s2 ^= ( _s2 >> 17 ), _s2 ^= ( _s2 << 30 );
204
205     uint64_t a = MMC_MULT1 * _s3 + _s4;
206     _s4 = ( a >> 32u ); // upper 32 bits of a
207     _s3 = uint32_t( a ); // lower 32 bits of a
208
209     a = MMC_MULT2 * _s5 + _s6;
210     _s6 = ( a >> 32u ); // upper 32 bits of a
211     _s5 = uint32_t( a ); // lower 32 bits of a
212
213     // use _s3 for lower 32 bits and _s5 << 32 for upper 32 bits of 64-bit word
214     return _s1 + _s2 + _s3 + ( (uint64_t)_s5 << 32 );
215 }
216
217 double rng32_01( void ) { // returns a random number in the half-open interval [0,1)
218
219     return double( rng32() ) * TW032_INV;
220 }
221
222 long double rng64_01( void ) { // returns a random number in the half-open interval [0,1)
223
224     return ( long double )( rng64() ) * TW064_INV;
225 }
226
227 private:
228
229     uint64_t _s1, _s2; // two 64-bit
230     uint32_t _s3, _s4, _s5, _s6; // important that these be 32-bit and not 64-bit
231 }; // end jlkiss64 class
232 } // end namespace JLKISS64
233
234 #endif // JLKISS64_H
235

```


Listing D-11. Generator.h

```

1 // Generator.h: template class file for random number generators
2 // R. Saucier, July 2016
3
4 #ifndef GENERATOR_H
5 #define GENERATOR_H
6
7 #include "Bitmatrix.h"
8 #include "mod_math.h"
9 #include <vector>
10 #include <bitset>
11 #include <iostream>
12
13 template <class T> // for 32-bit and 64-bit generators
14 class Generator {
15
16 public:
17
18     virtual ~Generator() {};// std::cout << "deleting Generator" << std::endl; }
19     virtual void setState( std::vector<T> seed ) = 0;
20     virtual void getState( std::vector<T>& seed ) = 0;
21     virtual void jump_ahead( uintmax_t ) = 0;
22     virtual void jump_ahead( uintmax_t, uintmax_t ) = 0;
23     virtual void jump_back( uintmax_t ) = 0;
24     virtual void jump_back( uintmax_t, uintmax_t ) = 0;
25     virtual void jump_cycle( void ) = 0;
26
27     virtual uint32_t rng32( void ) = 0; // returns 32-bit integer
28     virtual uint64_t rng64( void ) = 0; // returns 64-bit integer
29     virtual double rng32_01( void ) = 0; // returns double in [0,1)
30     virtual long double rng64_01( void ) = 0; // returns long double in [0,1)
31
32     inline double u32( double a = 0., double b = 1. ) { return a + ( b - a ) * this->rng32_01(); }
33     inline double u64( double a = 0., double b = 1. ) { return a + ( b - a ) * this->rng64_01(); }
34 };
35
36 // 32-bit generators
37 #include "kiss.h"
38 #include "jkiss.h"
39 #include "lfsr88.h"
40 #include "lfsr113.h"
41
42 // 64-bit generators
43 #include "jlkiss.h"
44 #include "jlkiss64.h"
45 #include "lfsr258.h"
46
47 #endif

```

Listing D-12. Random.h

```

1 // Random.h: Definition and Implementation of Random Number Distribution Class
2 // This rewrite of the following reference decouples the distributions from the generators
3 // Ref: Richard Saucier, "Computer Generation of Statistical Distributions," ARL-TR-2168,
4 // US Army Research Laboratory, Aberdeen Proving Ground, MD, 21005-5068, March 2000.
5 // R. Saucier, December 2016
6
7 #ifndef RANDOM_H
8 #define RANDOM_H
9
10 #include "Generator.h"
11 #include <iostream>
12 #include <fstream>
13 #include <vector>
14 #include <algorithm>
15 #include <functional>
16 #include <cassert>
17 #include <cmath>
18 #include <climits>
19 #include <cfloat> // for FLT_MIN and FLT_MAX
20 #include <unistd.h> // for getpid
21 #include <map>
22
23 namespace rnd { // rnd namespace
24
25 // for convenience, define some data structures for bivariate distributions
26
27 typedef std::pair<double,double> cartesianCoord; // x = first, y = second
28 typedef std::pair<double,double> sphericalCoord; // theta = first, phi = second
29 typedef std::pair<double,double> polarCoord; // r = first, theta = second
30
31 struct point2d { // cartesian coordinates in 2-D for use in stochasticInterpolation
32
33     double x, y;
34     point2d& operator+=( const point2d& p ) {
35         x += p.x;
36         y += p.y;
37         return *this;
38     }
39     point2d& operator-=( const point2d& p ) {
40         x -= p.x;
41         y -= p.y;
42         return *this;
43     }
44     point2d& operator+=( double scalar ) {
45         x += scalar;
46         y += scalar;
47         return *this;
48     }
49     point2d& operator/=( double scalar ) {
50         x /= scalar;
51         y /= scalar;
52         return *this;
53     }
54 }
55

```

```

54 };
55
56 // comparison functor for determining the neighborhood of a data point
57 struct dSquared : public std::binary_function< point2d, point2d, bool > {
58
59     bool operator()( point2d p, point2d q ) { return p.x * p.x + p.y * p.y < q.x * q.x + q.y * q.y; }
60 };
61
62 template <class Typename>    // for 32-bit and 64-bit generators
63 class Random {
64
65 public:
66
67     Random( Generator<Typename> *gen ) { _gen = gen; }
68     ~Random( void ) {} // default destructor
69
70     // Continuous Distributions
71
72     double arcsine( double xMin = 0., double xMax = 1. ) { // Arc Sine
73
74         double q = sin( M_PI_2 * _u() );
75         return xMin + ( xMax - xMin ) * q * q;
76     }
77
78     double beta( double v, double w, // Beta
79                 double xMin = 0., double xMax = 1. ) { // (v > 0. and w > 0.)
80
81         if ( v < w ) return xMax - ( xMax - xMin ) * beta( w, v );
82         double y1 = gamma( 0., 1., v );
83         double y2 = gamma( 0., 1., w );
84         return xMin + ( xMax - xMin ) * y1 / ( y1 + y2 );
85     }
86
87     double cauchy( double a = 0., double b = 1. ) { // Cauchy (or Lorentz)
88
89         // a is the location parameter and b is the scale parameter
90         // b is the half width at half maximum (HWHM) and variance doesn't exist
91
92         assert( b > 0. );
93
94         return a + b * tan( M_PI * uniform( -0.5, 0.5 ) );
95     }
96
97     double chiSquare( int df ) { // Chi-Square
98
99         assert( df >= 1 );
100
101         return gamma( 0., 2., 0.5 * double( df ) );
102     }
103
104     double cosine( double xMin = 0., double xMax = 1. ) { // Cosine
105
106         assert( xMin < xMax );
107
108         double a = 0.5 * ( xMin + xMax ); // location parameter
109         double b = ( xMax - xMin ) / M_PI; // scale parameter
110
111         return a + b * asin( uniform( -1., 1. ) );
112     }
113
114     double doubleLog( double xMin = -1., double xMax = 1. ) { // Double Log
115
116         assert( xMin < xMax );
117
118         double a = 0.5 * ( xMin + xMax ); // location parameter
119         double b = 0.5 * ( xMax - xMin ); // scale parameter
120
121         if ( bernoulli( 0.5 ) ) return a + b * _u();
122         else return a - b * _u();
123     }
124
125     double erlang( double b, int c ) { // Erlang (b > 0. and c >= 1)
126
127         assert( b > 0. && c >= 1 );
128
129         double prod = 1.;
130         for ( int i = 0; i < c; i++ ) prod *= _u();
131
132         return -b * log( prod );
133     }
134
135     double exponential( double a = 0., double c = 1. ) { // Exponential
136
137         // location a, shape c
138         assert( c > 0.0 );
139
140         return a - c * log( _u() );
141     }
142
143     double extremeValue( double a = 0., double c = 1. ) { // Extreme Value
144
145         // location a, shape c
146         assert( c > 0. );
147
148         return a + c * log( -log( _u() ) );
149     }
150
151     double fRatio( int v, int w ) { // F Ratio (v and w >= 1)
152
153         assert( v >= 1 && w >= 1 );
154
155         return ( chiSquare( v ) / v ) / ( chiSquare( w ) / w );
156     }
157
158     double gamma( double a, double b, double c ) { // Gamma
159
160         // location a, scale b, shape c
161         assert( b > 0. && c > 0. );
162
163         if ( c < 1. ) {

```

```

161     const double C = 1. + c / M.E;
162     while ( true ) {
163         double p = C * _u();
164         if ( p > 1. ) {
165             double y = -log( ( C - p ) / c );
166             if ( _u() <= pow( y, c - 1. ) ) return a + b * y;
167         }
168         else {
169             double y = pow( p, 1. / c );
170             if ( _u() <= exp( -y ) ) return a + b * y;
171         }
172     }
173 }
174 else if ( c == 1.0 ) return exponential( a, b );
175 else {
176     const double A = 1. / sqrt( 2. * c - 1. );
177     const double B = c - log( 4. );
178     const double Q = c + 1. / A;
179     const double T = 4.5;
180     const double D = 1. + log( T );
181     while ( true ) {
182         double p1 = _u();
183         double p2 = _u();
184         double v = A * log( p1 / ( 1. - p1 ) );
185         double y = c * exp( v );
186         double z = p1 * p1 * p2;
187         double w = B + Q * v - y;
188         if ( w + D - T * z >= 0. || w >= log( z ) ) return a + b * y;
189     }
190 }
191 }
192
193 double laplace( double a = 0., double b = 1. ) { // Laplace
194     // (or double exponential)
195     assert( b > 0. );
196
197     // composition method
198
199     if ( bernoulli( 0.5 ) ) return a + b * log( _u() );
200     else return a - b * log( _u() );
201 }
202
203 double logarithmic( double xMin = 0., double xMax = 1. ) { // Logarithmic
204
205     assert( xMin < xMax );
206
207     double a = xMin; // location parameter
208     double b = xMax - xMin; // scale parameter
209
210     // use convolution formula for product of two IID uniform variates
211
212     return a + b * _u() * _u();
213 }
214
215 double logistic( double a = 0., double c = 1. ) { // Logistic
216
217     assert( c > 0. );
218
219     return a - c * log( 1. / _u() - 1. );
220 }
221
222 double lognormal( double a, double mu, double sigma ) { // Lognormal
223
224     return a + exp( normal( mu, sigma ) );
225 }
226
227 double normal( double mu = 0., double sigma = 1. ) { // Normal
228
229     assert( sigma > 0. );
230
231     static bool f = true;
232     static double p2, q;
233     double p1, p;
234
235     if ( f ) {
236         do {
237             p1 = uniform( -1., 1. );
238             p2 = uniform( -1., 1. );
239             p = p1 * p1 + p2 * p2;
240         } while ( p >= 1. );
241         f = false;
242         q = sqrt( -2. * log( p ) / p );
243         return mu + sigma * p1 * q;
244     }
245     f = true;
246     return mu + sigma * p2 * q;
247 }
248
249 double parabolic( double xMin = 0., double xMax = 1. ) { // Parabolic
250
251     assert( xMin < xMax );
252
253     double a = 0.5 * ( xMin + xMax ); // location parameter
254     double yMax = _parabola( a, xMin, xMax ); // maximum function range
255
256     return userSpecified( _parabola, xMin, xMax, 0., yMax );
257 }
258
259 double pareto( double c ) { // Pareto
260     // shape c
261     assert( c > 0. );
262
263     return pow( _u(), -1. / c );
264 }
265
266 double pearson5( double b, double c ) { // Pearson Type 5
267     // scale b, shape c

```

```

268     assert( b > 0. && c > 0. );
269     return 1. / gamma( 0., 1. / b, c );
270 }
271
272 double pearson6( double b, double v, double w ) { // Pearson Type 6
273     // scale b, shape v & w
274     assert( b > 0. && v > 0. && w > 0. );
275     return gamma( 0., b, v ) / gamma( 0., b, w );
276 }
277
278 double power( double c ) { // Power
279     // shape c
280     assert( c > 0. );
281     return pow( _u(), 1. / c );
282 }
283
284 double rayleigh( double a, double b ) { // Rayleigh
285     // location a, scale b
286     assert( b > 0. );
287     return a + b * sqrt( -log( _u() ) );
288 }
289
290 double studentT( int df ) { // Student's T
291     // degrees of freedom df
292     assert( df >= 1 );
293     return normal() / sqrt( chiSquare( df ) / df );
294 }
295
296 double triangular( double xMin = 0., // Triangular
297                   double xMax = 1., // with default interval [0,1]
298                   double c = 0.5 ) { // and default mode 0.5
299     assert( xMin < xMax && xMin <= c && c <= xMax );
300     double p = _u(), q = 1. - p;
301     if ( p <= ( c - xMin ) / ( xMax - xMin ) )
302         return xMin + sqrt( ( xMax - xMin ) * ( c - xMin ) * p );
303     else
304         return xMax - sqrt( ( xMax - xMin ) * ( xMax - c ) * q );
305 }
306
307 double uniform( double xMin = 0., double xMax = 1. ) { // Uniform
308     // on [xMin,xMax]
309     assert( xMin < xMax );
310     return xMin + ( xMax - xMin ) * _u();
311 }
312
313 double userSpecified( // User-Specified Distribution
314     double (*usf)() { // pointer to user-specified function
315         double, // x
316         double, // xMin
317         double, // xMax
318         double xMin, double xMax, // function domain
319         double yMin, double yMax ) { // function range
320     assert( xMin < xMax && yMin < yMax );
321     double x, y, areaMax = ( xMax - xMin ) * ( yMax - yMin );
322     // acceptance-rejection method
323     do {
324         x = uniform( 0.0, areaMax ) / ( yMax - yMin ) + xMin;
325         y = usf( x );
326     } while ( y > areaMax );
327     return x;
328 }
329
330 double weibull( double a, double b, double c ) { // Weibull
331     // location a, scale b,
332     // shape c
333     assert( b > 0. && c > 0. );
334     return a + b * pow( -log( _u() ), 1. / c );
335 }
336
337 // Discrete Distributions
338
339 bool bernoulli( double p = 0.5 ) { // Bernoulli Trial
340     assert( 0. <= p && p <= 1. );
341     return _u() < p;
342 }
343
344 int binomial( int n, double p ) { // Binomial
345     assert( n >= 1 && 0. <= p && p <= 1. );
346     int sum = 0;
347     for ( int i = 0; i < n; i++ ) sum += bernoulli( p );
348     return sum;
349 }
350
351 int geometric( double p ) { // Geometric
352     assert( 0. < p && p <= 1. );
353     return int( log( _u() ) / log( 1. - p ) );
354 }

```

```

375 }
376
377 int hypergeometric( int n, int N, int K ) { // Hypergeometric
378     // trials n, size N,
379     assert( 0 <= n && n <= N && N >= 1 && K >= 0 ); // successes K
380
381     int count = 0;
382     for ( int i = 0; i < n; i++, N-- ) {
383         double p = double( K ) / double( N );
384         if ( bernoulli( p ) ) { count++; K--; }
385     }
386     return count;
387 }
388
389 void multinomial( int n, // Multinomial
390                 double p[], // trials n, probability vector p,
391                 int count[], // success vector count,
392                 int m ) { // number of disjoint events m
393
394     assert( m >= 2 ); // at least 2 events
395     double sum = 0.;
396     for ( int bin = 0; bin < m; bin++ ) sum += p[ bin ]; // probabilities
397     assert( sum == 1. ); // must sum to 1
398
399     for ( int bin = 0; bin < m; bin++ ) count[ bin ] = 0; // initialize
400
401     // generate n uniform variates in the interval [0,1] and bin the results
402
403     for ( int i = 0; i < n; i++ ) {
404         double lower = 0., upper = 0., u = _u();
405         for ( int bin = 0; bin < m; bin++ ) {
406             // locate subinterval, which is of length p[ bin ],
407             // that contains the variate and increment the corresponding counter
408
409             lower = upper;
410             upper += p[ bin ];
411             if ( lower <= u && u < upper ) { count[ bin ]++; break; }
412         }
413     }
414 }
415
416 int negativeBinomial( int s, double p ) { // Negative Binomial
417     // successes s, probability p
418     assert( s >= 1 && 0. < p && p < 1. );
419
420     int sum = 0;
421     for ( int i = 0; i < s; i++ ) sum += geometric( p );
422     return sum;
423 }
424
425 int pascal( int s, double p ) { // Pascal
426     // successes s, probability p
427     return negativeBinomial( s, p ) + s;
428 }
429
430 int poisson( double mu ) { // Poisson
431     assert( mu > 0. );
432
433     double a = exp( -mu );
434     double b = 1.;
435
436     int i;
437     for ( i = 0; b >= a; i++ ) b *= _u();
438     return i - 1;
439 }
440
441 int uniformDiscrete( int i, int j ) { // Uniform Discrete
442     // inclusive i to j
443     assert( i < j );
444     return i + int( ( j - i + 1 ) * _u() );
445 }
446
447 // Empirical and Data-Driven Distributions
448
449 double empirical( void ) { // Empirical Continuous
450
451     static std::vector< double > x, cdf;
452     static int n;
453     static bool init = false;
454
455     if ( !init ) {
456         std::ifstream in( "empiricalDistribution" );
457         if ( !in ) {
458             std::cerr << "Cannot open \"empiricalDistribution\" input file" << std::endl;
459             exit( 1 );
460         }
461         double value, prob;
462         while ( in >> value >> prob ) { // read in empirical distribution
463             x.push_back( value );
464             cdf.push_back( prob );
465         }
466         n = x.size();
467         init = true;
468
469         // check that this is indeed a cumulative distribution
470
471         assert( 0. == cdf[ 0 ] && cdf[ n - 1 ] == 1. );
472         for ( int i = 1; i < n; i++ ) assert( cdf[ i - 1 ] < cdf[ i ] );
473     }
474
475     double p = _u();

```

```

482     for ( int i = 0; i < n - 1; i++ )
483         if ( cdf[ i ] <= p && p < cdf[ i + 1 ] )
484             return x[ i ] + ( x[ i + 1 ] - x[ i ] ) *
485                 ( p - cdf[ i ] ) / ( cdf[ i + 1 ] - cdf[ i ] );
486     return x[ n - 1 ];
487 }
488
489 int empiricalDiscrete( void ) { // Empirical Discrete
490
491     static std::vector< int > k;
492     static std::vector< double > f[ 2 ]; // f[ 0 ] is pdf and f[ 1 ] is cdf
493     static double max;
494     static int n;
495     static bool init = false;
496
497     if ( !init ) {
498         std::ifstream in ( "empiricalDiscrete" );
499         if ( !in ) {
500             std::cerr << "Cannot open \"empiricalDiscrete\" input file" << std::endl;
501             exit( 1 );
502         }
503         int value;
504         double freq;
505         while ( in >> value >> freq ) { // read in empirical data
506             k.push_back( value );
507             f[ 0 ].push_back( freq );
508         }
509         n = k.size();
510         init = true;
511
512         // form the cumulative distribution
513         f[ 1 ].push_back( f[ 0 ][ 0 ] );
514         for ( int i = 1; i < n; i++ )
515             f[ 1 ].push_back( f[ 1 ][ i - 1 ] + f[ 0 ][ i ] );
516
517         // check that the integer points are in ascending order
518         for ( int i = 1; i < n; i++ ) assert( k[ i - 1 ] < k[ i ] );
519
520         max = f[ 1 ][ n - 1 ];
521     }
522
523     // select a uniform variate between 0 and the max value of the cdf
524     double p = uniform( 0., max );
525
526     // locate and return the corresponding index
527     for ( int i = 0; i < n; i++ ) if ( p <= f[ 1 ][ i ] ) return k[ i ];
528     return k[ n - 1 ];
529 }
530
531 double sample( bool replace = true ) { // Sample w or w/o replacement from a
532     // distribution of 1-D data in a file
533     static std::vector< double > v; // vector for sampling with replacement
534     static bool init = false; // flag that file has been read in
535     static int n; // number of data elements in the file
536     static int index = 0; // subscript in the sequential order
537
538     if ( !init ) {
539         std::ifstream in( "sampleData" );
540         if ( !in ) {
541             std::cerr << "Cannot open \"sampleData\" file" << std::endl;
542             exit( 1 );
543         }
544         double d;
545         while ( in >> d ) v.push_back( d );
546         in.close();
547         n = v.size();
548         init = true;
549         if ( replace == false ) { // sample without replacement
550             // shuffle contents of v once and for all
551             // Ref: Knuth, D. E., The Art of Computer Programming, Vol. 2:
552             // Seminumerical Algorithms. London: Addison-Wesley, 1969.
553             for ( int i = n - 1; i > 0; i-- ) {
554                 int j = int( ( i + 1 ) * _u() );
555                 std::swap( v[ i ], v[ j ] );
556             }
557         }
558     }
559
560     // return a random sample
561     if ( replace ) // sample w/ replacement
562         return v[ uniformDiscrete( 0, n - 1 ) ];
563     else { // sample w/o replacement
564         assert( index < n ); // retrieve elements
565         return v[ index++ ]; // in sequential order
566     }
567 }
568
569 void sample( double x[], int ndim ) { // Sample from a given distribution
570     // of multi-dimensional data
571     static const int N_DIM = 6;
572     assert( ndim <= N_DIM );
573
574     static std::vector< double > v[ N_DIM ];
575     static bool init = false;
576     static int n;
577
578     if ( !init ) {
579         std::ifstream in( "sampleData" );
580         if ( !in ) {
581             std::cerr << "Cannot open \"sampleData\" file" << std::endl;
582         }
583     }

```

```

589         exit( 1 );
590     }
591     double d;
592     while ( !in.eof() ) {
593         for ( int i = 0; i < ndim; i++ ) {
594             in >> d;
595             v[ i ].push_back( d );
596         }
597     }
598     in.close();
599     n = v[ 0 ].size();
600     init = true;
601 }
602 int index = uniformDiscrete( 0, n - 1 );
603 for ( int i = 0; i < ndim; i++ ) x[ i ] = v[ i ][ index ];
604 }
605
606 point2d stochasticInterpolation( void ) { // Stochastic Interpolation
607
608     // Refs: Taylor, M. S. and J. R. Thompson, Computational Statistics & Data Analysis, Vol. 4, pp. 93-101, 1986;
609     //       Thompson, J. R., Empirical Model Building, pp. 108-114, Wiley, 1989;
610     //       Bodt, B. A. and M. S. Taylor, A Data Based Random Number Generator for A Multivariate Distribution -
611     //       A User's Manual, ARBRL-TR-02439, BRL, APG, MD, Nov. 1982.
612
613     static std::vector<point2d> data;
614     static point2d min, max;
615     static int m;
616     static double lower, upper;
617     static bool init = false;
618
619     if ( !init ) {
620         std::ifstream in( "stochasticData" );
621         if ( !in ) {
622             std::cerr << "Cannot open \"stochasticData\" input file" << std::endl;
623             exit( 1 );
624         }
625
626         // read in the data and set min and max values
627
628         min.x = min.y = FLT_MAX;
629         max.x = max.y = FLT_MIN;
630         point2d p;
631         while ( in >> p.x >> p.y ) {
632
633             min.x = ( p.x < min.x ? p.x : min.x );
634             min.y = ( p.y < min.y ? p.y : min.y );
635             max.x = ( p.x > max.x ? p.x : max.x );
636             max.y = ( p.y > max.y ? p.y : max.y );
637
638             data.push_back( p );
639         }
640         in.close();
641         init = true;
642
643         // scale the data so that each dimension will have equal weight
644
645         for ( unsigned int i = 0; i < data.size(); i++ ) {
646
647             data[ i ].x = ( data[ i ].x - min.x ) / ( max.x - min.x );
648             data[ i ].y = ( data[ i ].y - min.y ) / ( max.y - min.y );
649         }
650
651         // set m, the number of points in a neighborhood of a given point
652
653         m = data.size() / 20; // 5% of all the data points
654         if ( m < 5 ) m = 5; // but no less than 5
655         if ( m > 20 ) m = 20; // and no more than 20
656
657         lower = ( 1. - sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
658         upper = ( 1. + sqrt( 3. * ( double( m ) - 1. ) ) ) / double( m );
659     }
660
661     // uniform random selection of a data point (with replacement)
662
663     point2d origin = data[ uniformDiscrete( 0, data.size() - 1 ) ];
664
665     // make this point the origin of the coordinate system
666
667     for ( unsigned int n = 0; n < data.size(); n++ ) data[ n ] -= origin;
668
669     // sort the data with respect to its distance (squared) from this origin
670
671     std::sort( data.begin(), data.end(), dSquared() );
672
673     // find the mean value of the data in the neighborhood about this point
674
675     point2d mean;
676     mean.x = mean.y = 0.;
677     for ( int n = 0; n < m; n++ ) mean += data[ n ];
678     mean /= double( m );
679
680     // select a random linear combination of the points in this neighborhood
681
682     point2d p;
683     p.x = p.y = 0.;
684     for ( int n = 0; n < m; n++ ) {
685
686         double rn;
687         if ( m == 1 ) rn = 1.;
688         else rn = uniform( lower, upper );
689         p.x += rn * ( data[ n ].x - mean.x );
690         p.y += rn * ( data[ n ].y - mean.y );
691     }
692
693     // restore the data to its original form
694
695     for ( unsigned int n = 0; n < data.size(); n++ ) data[ n ] += origin;

```

```

696
697 // use mean and original point to translate the randomly-chosen point
698
699 p += mean;
700 p += origin;
701
702 // scale randomly-chosen point to the dimensions of the original data
703
704 p.x = p.x * ( max.x - min.x ) + min.x;
705 p.y = p.y * ( max.y - min.y ) + min.y;
706
707 return p;
708 }
709
710 // Multivariate Distributions
711
712 cartesianCoord bivariateNormal( double muX = 0.,
713                                double sigmaX = 1.,
714                                double muY = 0.,
715                                double sigmaY = 1. ) { // Bivariate Gaussian
716
717     assert( sigmaX > 0. && sigmaY > 0. );
718
719     return make_pair( normal( muX, sigmaX ), normal( muY, sigmaY ) );
720 }
721
722 cartesianCoord bivariateUniform( double xMin = -1.,
723                                 double xMax = 1.,
724                                 double yMin = -1.,
725                                 double yMax = 1. ) { // Bivariate Uniform
726
727     assert( xMin < xMax && yMin < yMax );
728
729     double x0 = 0.5 * ( xMin + xMax );
730     double y0 = 0.5 * ( yMin + yMax );
731     double a = 0.5 * ( xMax - xMin );
732     double b = 0.5 * ( yMax - yMin );
733     double x, y;
734
735     do {
736         x = uniform( -1., 1. );
737         y = uniform( -1., 1. );
738     } while( x * x + y * y > 1. );
739
740     return make_pair( x0 + a * x, y0 + b * y );
741 }
742
743 polarCoord circularUniform( double rMin = 0.,
744                             double rMax = 1.,
745                             double thMin = 0.,
746                             double thMax = 2. * M_PI ) { // Circular Uniform
747
748     assert( 0 <= rMin && rMin <= rMax && thMin <= thMax );
749
750     double r = sqrt( uniform( rMin * rMin, rMax * rMax ) );
751     double th = uniform( thMin, thMax );
752
753     return make_pair( r, th );
754 }
755
756 cartesianCoord corrNormal( double r,
757                           double muX = 0.,
758                           double sigmaX = 1.,
759                           double muY = 0.,
760                           double sigmaY = 1. ) { // Correlated Normal
761
762     assert( -1. <= r && r <= 1. && // bounds on correlation coeff
763            sigmaX > 0. && sigmaY > 0. ); // positive std dev
764
765     double x = normal();
766     double y = normal();
767
768     y = r * x + sqrt( 1. - r * r ) * y; // correlate the variables
769
770     // translate and scale
771     return make_pair( muX + sigmaX * x, muY + sigmaY * y );
772 }
773
774 cartesianCoord corrUniform( double r,
775                            double xMin = 0.,
776                            double xMax = 1.,
777                            double yMin = 0.,
778                            double yMax = 1. ) { // Correlated Uniform
779
780     assert( -1. <= r && r <= 1. && // bounds on correlation coeff
781            xMin < xMax && yMin < yMax ); // bounds on domain
782
783     double x0 = 0.5 * ( xMin + xMax );
784     double y0 = 0.5 * ( yMin + yMax );
785     double a = 0.5 * ( xMax - xMin );
786     double b = 0.5 * ( yMax - yMin );
787     double x, y;
788
789     do {
790         x = uniform( -1., 1. );
791         y = uniform( -1., 1. );
792     } while ( x * x + y * y > 1. );
793
794     y = r * x + sqrt( 1. - r * r ) * y; // correlate the variables
795
796     // translate and scale
797     return make_pair( x0 + a * x, y0 + b * y );
798 }
799
800 private:
801
802

```



```

803
804 // function returns an associative array where each element of the vector is the key and the rank is the value
805 inline std::map<double,int> _rank( std::vector<double> v ) { // NB: pass a copy of the vector, not a reference
806
807     std::sort( v.begin(), v.end() ); // sort the vector in ascending order
808     std::map<double,int> r;
809     for ( unsigned int i = 0; i < v.size(); i++ ) r[v[i]] = i; // map the element to its rank
810     return r;
811 }
812
813 public:
814
815 // correlate two distributions without changing the marginal distributions
816 // (Thanks to Dr. Joseph Collins for describing this technique. For an understanding of the theory,
817 // and more general cases of any number of distributions, see "Inducing Dependence in Multivariate Random Samples,"
818 // unpublished paper, August 25, 2011, and references cited therein.)
819 void corrDist( std::vector<double>& dist1, std::vector<double>& dist2, double rankCorr ) { // the two distributions are reordered to induce
    the correlation
820
821     std::vector<double> t1( dist1 ), t2( dist2 ); // copy the two distributions
822     std::sort( t1.begin(), t1.end() ); // sort the copies in ascending order
823     std::sort( t2.begin(), t2.end() );
824
825     double x, y, c = 2. * sin( rankCorr * M_PI / 6. ), s = sqrt( 1. - c * c );
826     const int N = dist1.size();
827     std::vector<double> u(N), v(N);
828     for ( int i = 0; i < N; i++ ) { // generate two correlated vectors with the given rank correlation
829
830         x = normal();
831         y = normal();
832         y = c * x + s * y; // perform a rotation to induce the correlation
833         u[i] = x;
834         v[i] = y;
835     }
836     std::map<double,int> rank_u = _rank( u ); // generate maps from the values to the corresponding ranks
837     std::map<double,int> rank_v = _rank( v );
838
839     for ( int i = 0; i < N; i++ ) { // apply these maps as index functions to the sorted distributions
840
841         dist1[i] = t1[ rank_u[ u[i] ] ];
842         dist2[i] = t2[ rank_v[ v[i] ] ];
843     }
844 }
845
846 sphericalCoord spherical( double thMin = 0.,
847                          double thMax = M_PI,
848                          double phMin = 0.,
849                          double phMax = 2. * M_PI ) { // Uniform Spherical
850
851     assert( 0. <= thMin && thMin < thMax && thMax <= M_PI &&
852            0. <= phMin && phMin < phMax && phMax <= 2. * M_PI );
853
854     // return polar angle and azimuth
855     return make_pair( acos( uniform( cos( thMax ), cos( thMin ) ) ), uniform( phMin, phMax ) );
856 }
857
858 void sphericalND( double x[], int n ) { // Uniform over the surface of an n-dimensional unit sphere
859
860     // generate a point inside the unit n-sphere by normal polar method
861
862     double r2 = 0.;
863     for ( int i = 0; i < n; i++ ) {
864         x[ i ] = normal();
865         r2 += x[ i ] * x[ i ];
866     }
867
868     // project the point onto the surface of the unit n-sphere by scaling
869
870     const double A = 1. / sqrt( r2 );
871     for ( int i = 0; i < n; i++ ) x[ i ] *= A;
872 }
873
874 // Number Theoretic Distributions
875
876 double avoidance( void ) { // Maximal Avoidance (1-D), overloaded for convenience
877
878     double x[ 1 ];
879     avoidance( x, 1 );
880     return x[ 0 ];
881 }
882
883 void avoidance( double x[], unsigned int ndim ) { // Maximal Avoidance (N-D)
884
885     static const unsigned int MAXBIT = 30;
886     static const unsigned int MAXDIM = 6;
887
888     assert( ndim <= MAXDIM );
889
890     static unsigned long ix[ MAXDIM + 1 ] = { 0 };
891     static unsigned long *u[ MAXBIT + 1 ];
892     static unsigned long mdeg[ MAXDIM + 1 ] = { // degree of
893         0, // primitive polynomial
894         1, 2, 3, 3, 4, 4
895     };
896     static unsigned long p[ MAXDIM + 1 ] = { // decimal encoded
897         0, // interior bits
898         0, 1, 1, 2, 1, 4
899     };
900     static unsigned long v[ MAXDIM * MAXBIT + 1 ] = {
901         0,
902         1, 1, 1, 1, 1, 1,
903         3, 1, 3, 3, 1, 1,
904         5, 7, 7, 3, 3, 5,
905         15, 11, 5, 15, 13, 9
906     };
907
908     static double fac;

```

```

909     static int in = -1;
910     unsigned int j, k;
911     unsigned long i, m, pp;
912
913     if ( in == -1 ) {
914         in = 0;
915         fac = 1. / ( 1L << MAXBIT );
916         for ( j = 1, k = 0; j <= MAXBIT; j++, k += MAXDIM ) u[ j ] = &v[ k ];
917         for ( k = 1; k <= MAXDIM; k++ ) {
918             for ( j = 1; j <= mdeg[ k ]; j++ ) u[ j ][ k ] <= ( MAXBIT - j );
919             for ( j = mdeg[ k ] + 1; j <= MAXBIT; j++ ) {
920                 pp = p[ k ];
921                 i = u[ j - mdeg[ k ] ][ k ];
922                 i ^= ( i >> mdeg[ k ] );
923                 for ( int n = mdeg[ k ] - 1; n >= 1; n-- ) {
924                     if ( pp & 1 ) i ^= u[ j - n ][ k ];
925                     pp >>= 1;
926                 }
927                 u[ j ][ k ] = i;
928             }
929         }
930     }
931     m = in++;
932     for ( j = 0; j < MAXBIT; j++, m >>= 1 ) if ( !( m & 1 ) ) break;
933     if ( j >= MAXBIT ) exit( 1 );
934     m = j * MAXDIM;
935     for ( k = 0; k < ndim; k++ ) {
936         ix[ k + 1 ] ^= v[ m + k + 1 ];
937         x[ k ] = ix[ k + 1 ] * fac;
938     }
939 }
940
941 private:
942     Generator<Typename> *_gen;
943     static const unsigned int N_BITS = CHAR_BIT * sizeof( Typename ); // number of bits (32 or 64)
944
945     long double _u( void ) {
946         if ( N_BITS == 32 ) return _gen->rng32_01();
947         else return _gen->rng64_01();
948     }
949
950
951     static double _parabola( double x, double xMin, double xMax ) { // parabola
952         if ( x < xMin || x > xMax ) return 0.0;
953
954         double a = 0.5 * ( xMin + xMax ); // location parameter
955         double b = 0.5 * ( xMax - xMin ); // scale parameter
956         double yMax = 0.75 / b;
957
958         return yMax * ( 1. - ( x - a ) * ( x - a ) / ( b * b ) );
959     }
960 }; // Random class
961 } // rnd namespace
962
963 #endif // RANDOM_H

```