ARMY RESEARCH LABORATORY

Computer Generation of Statistical Distributions

by Richard Saucier

ARL-TR-2168 March 2000

ABSTRACT

This report presents a collection of computer–generated statistical distributions which are useful for performing Monte Carlo simulations. The distributions are encapsulated into a C++ class, called ‘‘Random,’’ so that they can be used with any C++ program. The class currently contains 27 continuous distributions, 9 discrete distributions, data–driven distributions, bivariate distributions, and number–theoretic distributions. The class is designed to be flexible and extensible, and this is supported in two ways: (1) a function pointer is provided so that the user–programmer can specify an arbitrary probability density function, and (2) new distributions can be easily added by coding them directly into the class. The format of the report is designed to provide the practitioner of Monte Carlo simulations with a handy reference for generating statistical distributions. However, to be self–contained, various techniques for generating distributions are also discussed, as well as procedures for estimating distribution parameters from data. Since most of these distributions rely upon a good underlying uniform distribution of random numbers, several candidate generators are presented along with selection criteria and test results. Indeed, it is noted that one of the more popular generators is probably overused and under what conditions it should be avoided.

ACKNOWLEDGMENTS

The author would like to thank Linda L. C. Moss and Robert Shnidman for correcting a number of errors and suggesting improvements on an earlier version of this report. The author is especially indebted to Richard S. Sandmeyer for generously sharing his knowledge of this subject area, suggesting generalizations for a number of the distributions, testing the random number distributions against their analytical values, as well as carefully reviewing the entire manuscript. Needless to say, any errors that remain are not the fault of the reviewers—nor the author—but rather are to be blamed on the computer.

INTENTIONALLY LEFT BLANK.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTENTIONALLY LEFT BLANK.

1. SUMMARY

This report presents a collection of various distributions of random numbers, suitable for performing Monte Carlo simulations. They have been organized into a C++ class, called ‘‘Random,’’ which is callable from any C++ program. Using the Random class is very simple. For example, the following is source code to print 1,000 normal variates with a mean of zero and a variance of one.

```
// Sample program for using the Random class

#include <iostream.h>
#include "Random.h"                  ⇐   include the definition of the Random class

void main( void )
{
   Random rv;                        ⇐   declare a random variate

   for ( int i = 0; i < 1000; i++ )
      cout << rv.normal()            ⇐   reference the normal distribution (with default parameters)
           << endl;
}
```

There are various aspects that the programmer will wish to know at this point, such as how the random number seed is set and how to compile and link the sample program. These aspects are discussed later (see Appendix B). The point to be emphasized here is that the Random class is very easy and straightforward to use. The class itself is quite comprehensive, currently containing 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 very easy to extend the collection to include new distributions.

2. INTRODUCTION

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

- *To Supplement Theory*
  While the underlying process or physical law may be understood, an analytical solution—or even a solution by numerical methods—may not be available. In addition, even in the cases where we possess a deterministic solution, we may be unable to obtain the initial conditions or other information necessary to apply it.
- *To Supplement Experiment*
  Experiments can be very costly or we may be unable to perform the measurements required for a particular mathematical model.
- *Computing Power has Increased while Cost has Decreased*
  In 1965, when writing an article for *Electronics* magazine, Gordon Moore formulated what has since been named Moore’s Law: the number of components that could be squeezed onto a silicon chip would double every year. Moore updated this prediction in 1975 from doubling every year to doubling every two years. These observations proved remarkably accurate; the processing technology of 1996, for example, was some eight million times more powerful than that of 1966 [Helicon Publishing 1999].

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

3. METHODS FOR GENERATING RANDOM NUMBER DISTRIBUTIONS

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

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

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.

(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 Figure 1.

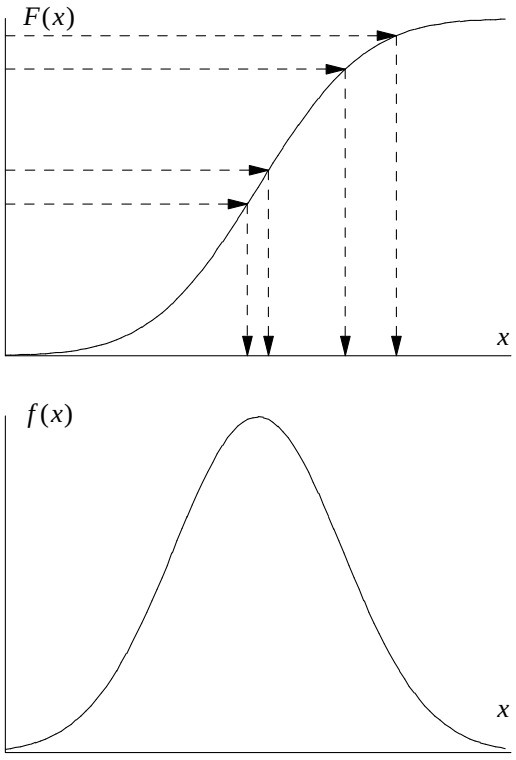


Figure 1. Inverse Transform Method.

We take uniformly distributed samples along the y axis between 0 and 1. We see that, where the distribution function $F(x)$ is relatively steep, 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, there will result in a corresponding lower density of points along the x axis (giving a smaller value of $f(x)$). More formally, if

* 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.’’

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

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

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

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

3.2 Composition

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

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

where

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

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

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

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

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

This can also be written as

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

where

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

Now each of these has an indefinite integral, namely

---

* 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.

† 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.

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

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

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

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

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

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

$$F_{X+Y}(u) = \int\int_{x+y \leq u} f(x,y)\, dx\, dy \tag{11}$$

$$= \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{u-x} f(x,y)\, dy \right) dx\,. \tag{12}$$

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

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

where we used Leibniz's rule (see Glossary) 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

$$f_{X+Y}(u) = \int_{-\infty}^{\infty} f(x) f(u-x)\, dx\,. \tag{14}$$

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

$$f_{X+Y}(u) = \int_0^1 f(u-x)\, dx = \int_{u-1}^{u} f(t)\, dt = \begin{cases} u & u \leq 1 \\ 2-u & 1 < u \leq 2 \end{cases}, \tag{15}$$

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

$$f_{X+Y}(z) = \int_0^z f_X(x) f_Y(z-x)\, dx = \int_0^z \frac{1}{b} e^{-x/b} \frac{1}{b} e^{-(z-x)/b}\, dx = \frac{1}{b^2} z e^{-z/b}\,. \tag{16}$$

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

$$f_{X_1+\cdots+X_n}(x) = \frac{x^{n-1}e^{-x/b}}{(n-1)!\,b^n} = \text{gamma}\,(0, b, n), \tag{17}$$

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 $i = 1, 2, \cdots, n$.
(2) Set $X = X_1 + X_2 + \cdots + X_n$.

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

$$X = -b \ln X_1 - \cdots - b \ln X_c = -b \ln (X_1 \cdots X_c). \tag{18}$$

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

$$X = -b \ln \prod_{i=1}^{c} X_i. \tag{19}$$

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

$$F_{XY}(u) = \int\int_{xy \le u} f(x, y)\, dx\, dy \tag{20}$$

$$= \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{u/x} f(x, y) dy \right) dx\ . \tag{21}$$

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

$$f_{XY}(u) = \int_{-\infty}^{\infty} f(x, u/x)\, \frac{1}{x}\, dx\ . \tag{22}$$

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

$$f_{XY}(u) = \int_{-\infty}^{\infty} f(x)\, f(u/x)\, \frac{1}{x}\, dx\ . \tag{23}$$

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

$$f_{XY}(u) = \int_{u}^{1} \frac{1}{x}\, dx = -\ln u. \tag{24}$$

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:

$$F_{Y/X}(u) = \int\int_{y/x \le u} f(x, y)\, dx\, dy \tag{25}$$

$$= \int_{-\infty}^{\infty} \left(\int_{y=-\infty}^{ux} f(x, y) dy \right) dx\ . \tag{26}$$

Differentiating this to get the density,

$$f_{Y/X}(u) = \int_{-\infty}^{\infty} f(x, ux)\,|x|\,dx\ . \tag{27}$$

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

$$f_{Y/X}(u) = \int_{-\infty}^{\infty} f(x)\,f(ux)\,|x|\,dx = \frac{1}{2\pi}\int_{-\infty}^{\infty} e^{-x^2/2}e^{-u^2x^2/2}\,|x|\,dx\ , \tag{28}$$

and we find that

$$f_{Y/X}(u) = \frac{1}{\pi}\int_{0}^{\infty} e^{-(1+u^2)x^2/2}\,x\,dx = \frac{1}{\pi(1+u^2)}\ . \tag{29}$$

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 Figure 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.

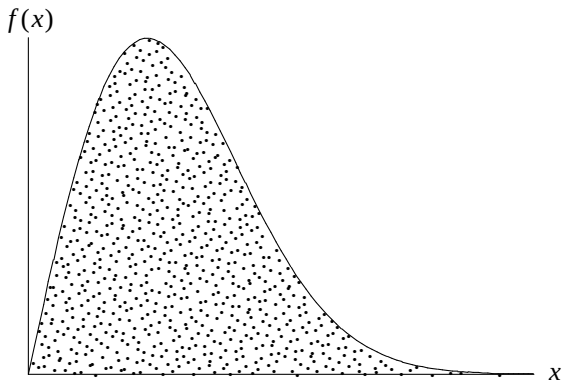


Figure 2. Probability Density Generated From Uniform Areal Density.

A simple way to do this is as follows.

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

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 $\hat{f}(x) \ge f(x)$ for all x in the domain, and, at the same time, the integral of $\hat{f}$ is invertible. Thus, let

$$\hat{F}(x) = \int_{x_{\min}}^{x} \hat{f}(x)\,dx \quad \text{and define} \quad A_{\max} = \int_{x_{\min}}^{x_{\max}} \hat{f}(x)\,dx\ . \tag{30}$$

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 $f(x) = 10,296\, x^5(1-x)^7$. 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 2.97187 at $x = 5/12$. Therefore, the function $\hat{f}(x) = 2.97187$ majorizes $f(x)$. So, with $A_{\max} = 2.97187$, $F(x) = 2.97187\, x$, and $y_{\max} = 2.97187$, the algorithm is as follows.

(1) Select $A \sim U(0, 2.97187)$.
(2) Compute $X = A / 2.97187$.
(3) Independently select $Y \sim U(0, 2.97187)$.
(4) Accept X if and only if $Y \leq f(X)$.

3.5 Sampling and Data–Driven Techniques

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

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

3.6 Techniques Based on Number Theory

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

1. *Primitive Polynomials Modulo Two*
   These are useful for generating random bits of 1's and 0's that cycle through all possible combinations (excluding all zeros) before repeating. This is discussed in section 5.5.1.
2. *Prime Number Theory*
   This has been exploited to produce sequences of quasi-random numbers that are self-avoiding. This is discussed in section 5.5.2.

3.7 Monte Carlo Simulation

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

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

$$A_p = \gamma (m/\rho)^{3/2}, \tag{31}$$

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

$$\gamma(\theta, \phi) = \sin\theta \cos\phi + \sin\theta \sin\phi + \cos\theta. \tag{32}$$

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 $f(\gamma)$.

```
Midpt    Freq
1.008       0
1.025       1 *
1.041       0
1.057       2 **
1.074       1 *
1.090       2 **
1.106       5 *****
1.123       2 **
1.139       5 *****
1.155       6 ******
1.172       5 *****
1.188      11 ***********
1.205      14 **************
1.221      12 ************
1.237       8 ********
1.254      10 **********
1.270      11 ***********
1.286      15 ***************
1.303      13 *************
1.319      21 *********************
1.335      19 *******************
1.352      19 *******************
1.368      26 **************************
1.384      34 **********************************
1.401      25 *************************
1.417      34 **********************************
1.434      40 ****************************************
1.450      39 ***************************************
1.466      42 ******************************************
1.483      43 *******************************************
1.499      33 *********************************
1.515      41 *****************************************
1.532      32 ********************************
1.548      29 *****************************
1.564      37 *************************************
1.581      34 **********************************
1.597      39 ***************************************
1.614      43 *******************************************
1.630      29 *****************************
1.646      45 *********************************************
1.663      43 *******************************************
1.679      41 *****************************************
1.695      30 ******************************
1.712      35 ***********************************
1.728      24 ************************
```

Figure 3. Shape Factor Probability Density of a Randomly Oriented Cube via Monte Carlo Simulation.

Incidently, we find that

Output	=	$\gamma \in [1, \sqrt{3})$,
Mean	=	3/2, and
Variance	=	$4/\pi - 5/4$.

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.

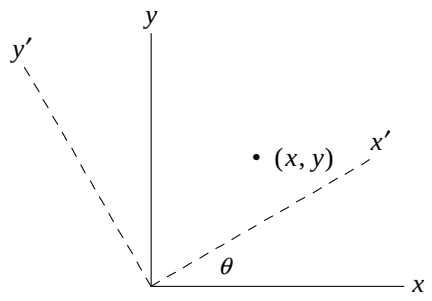


Figure 4. Coordinate Rotation to Induce Correlations.

The transformation between the two coordinate systems is given by

$$x' = x\cos\theta + y\sin\theta \qquad \text{and} \qquad y' = -x\sin\theta + y\cos\theta\ . \tag{33}$$

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

$$x' = \rho x + \sqrt{1-\rho^2}\, y \tag{34}$$

induces the desired correlation. To check this,

$$\text{corr}(x, x') = \rho\,\text{corr}(x, x) + \sqrt{1-\rho^2}\,\text{corr}(x, y) = \rho\,(1) + \sqrt{1-\rho^2}\,(0) = \rho, \tag{35}$$

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

Here are some special cases:

$$\begin{cases} \theta = 0 & \rho = 1 & x' = x \\ \theta = \pi/2 & \rho = 0 & x' \text{ is independent of } x \\ \theta = \pi & \rho = -1 & x' = -x \end{cases}\ . \tag{36}$$

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 and suppose that we want to truncate the distribution to the subinterval $[a, b]$. This can be accomplished by defining a truncated density:

$$\tilde{f}(x) \equiv \begin{cases} \dfrac{f(x)}{F(b) - F(a)} & a \le x \le b \\ \\ 0 & \text{otherwise} \end{cases}\ , \tag{37}$$

which has corresponding truncated distribution

$$\tilde{F}(x) \equiv \begin{cases} 0 & x < a \\ \dfrac{F(x) - F(a)}{F(b) - F(a)} & a \le x \le b \\ 1 & x > b \end{cases} . \tag{38}$$

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

(1) Generate $U \sim U(0, 1)$.
(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 (38), or if we do not have an explicit formula for F^{-1}, then a less efficient but nevertheless correct method of producing the truncated distribution is the following algorithm.

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

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

4. PARAMETER ESTIMATION

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

4.1 Linear Regression (Least–Squares Estimate)

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

$$F(x) = 1 - \exp[-(x/b)^c] . \tag{39}$$

We first sort the data x_i in accending order:

$$x_1 \le x_2 \le x_3 \le \cdots \le x_N . \tag{40}$$

The corresponding cumulative probability is $F(x_i) = F_i = i/N$. Rearranging eq. (39) so that the parameters appear linearly, we have

$$\ln[-\ln(1 - F_i)] = c \ln x_i - c \ln b . \tag{41}$$

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

4.2 Maximum Likelihood Estimation

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

$$L(\beta) = f_\beta(X_1) f_\beta(X_2) \cdots f_\beta(X_N) . \tag{42}$$

* 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.

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

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

$$f_{\mu,\sigma^2}(x) = \begin{cases} \dfrac{1}{\sqrt{2\pi}\,\sigma x} \exp\left[-\dfrac{(\ln x - \mu)^2}{2\sigma^2} \right] & x > 0 \\ 0 & \text{otherwise} \end{cases} . \tag{43}$$

The log-likelihood function is

$$\ln L(\mu, \sigma^2) = \ln \prod_{i=1}^{N} f_{\mu,\sigma^2}(x_i) = \sum_{i=1}^{N} \ln f_{\mu,\sigma^2}(x_i) \tag{44}$$

and, in this case,

$$\ln L(\mu, \sigma^2) = \sum_{i=1}^{N} \left[\ln(\sqrt{2\pi\sigma^2} x_i) + \frac{(\ln x_i - \mu)^2}{2\sigma^2} \right]. \tag{45}$$

This is a maximuum when both

$$\frac{\partial \ln L(\mu, \sigma^2)}{\partial \mu} = 0 \quad \text{and} \quad \frac{\partial \ln L(\mu, \sigma^2)}{\partial \sigma^2} = 0 \tag{46}$$

and we find

$$\mu = \frac{1}{N} \sum_{i=1}^{N} \ln x_i \quad \text{and} \quad \sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (\ln x_i - \mu)^2 . \tag{47}$$

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 practioner 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*
Arcsin	x_{min} and x_{max}	yes
Beta	x_{min}, x_{max}, and shape v and w	only when v and w are equal
Cauchy (Lorentz)	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 (Double Exponential)	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 (Inverted Gamma)	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 v (degrees of freedom)	yes
Triangular	x_{min}, x_{max}, and shape c	only when $c = (x_{min} + x_{max})/2$
Uniform	x_{min} and x_{max}	yes
User–Specified	x_{min}, x_{max} and y_{min}, y_{max}	depends upon the function
Weibull	location a, scale b, and shape c	no

5.1.1 Arcsine

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

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

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

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

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 $i = 1, 2, \cdots, N$

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

Source Code:

```
double arcsine( double xMin, double xMax )
{
   assert( xMin < xMax );

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

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

Examples of the probability density function and the cumulative distribution function are shown in Figures 5 and 6, respectively.

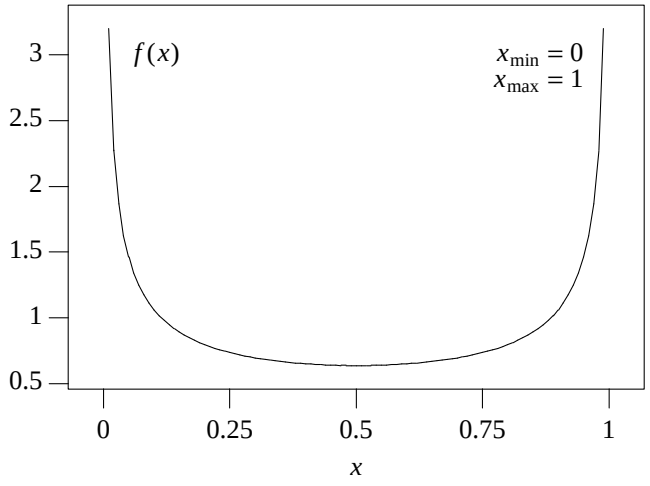


Figure 5. Arcsine Density Function.

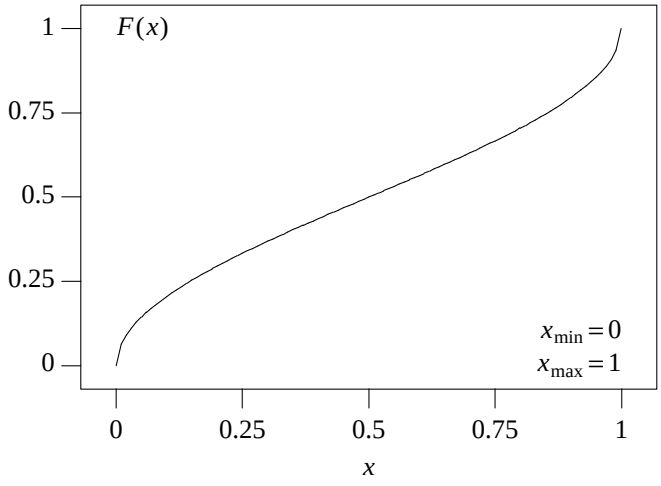


Figure 6. Arcsine Distribution Function.

5.1.2 Beta

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

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

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

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

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

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

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

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

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

Algorithm: (1) Generate two IID gamma variates, $Y_1 \sim \text{gamma}(1, v)$ and $Y_2 \sim \text{gamma}(1, w)$

(2) Return $X = \begin{cases} x_{\min} + (x_{\max} - x_{\min})\,Y_1/(Y_1+Y_2) & \text{if } v \ge w \\ x_{\max} - (x_{\max} - x_{\min})\,Y_2/(Y_1+Y_2) & \text{if } v < w \end{cases}$

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

Notes: (1) $X \sim \text{Beta}(v, w)$ if and only if $1 - X \sim \text{Beta}(w, v)$.
(2) When $v = w = 1/2$, this reduces to the *arcsine* distribution.
(3) When $v = w = 1$, this reduces to the *uniform* distribution.

Examples of probability density functions and cumulative distribution functions are shown in Figures 7 and 8, respectively.

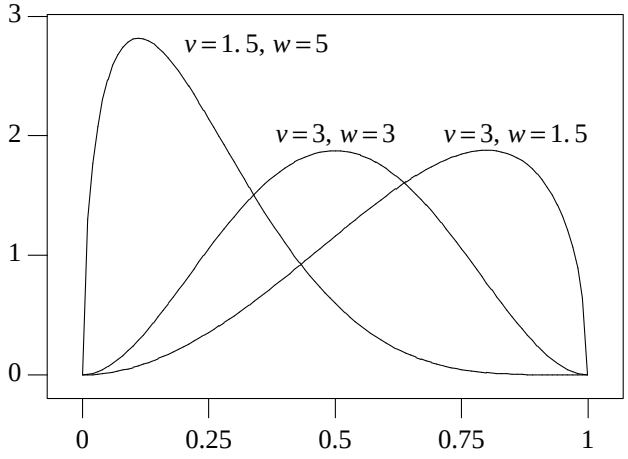


Figure 7. Beta Density Functions.

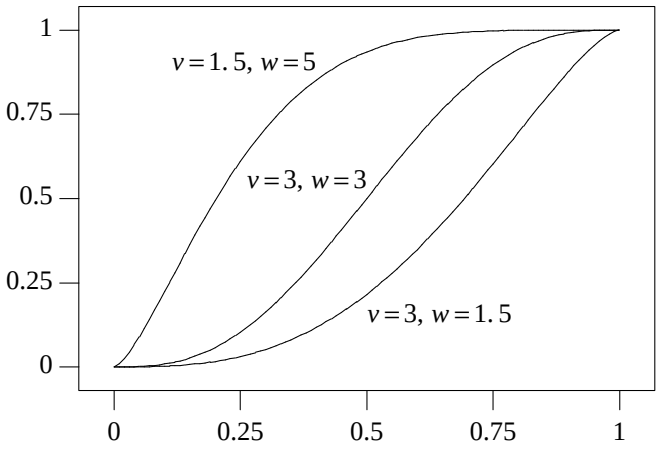


Figure 8. Beta Distribution Functions.

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:

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 9 and 10, respectively.

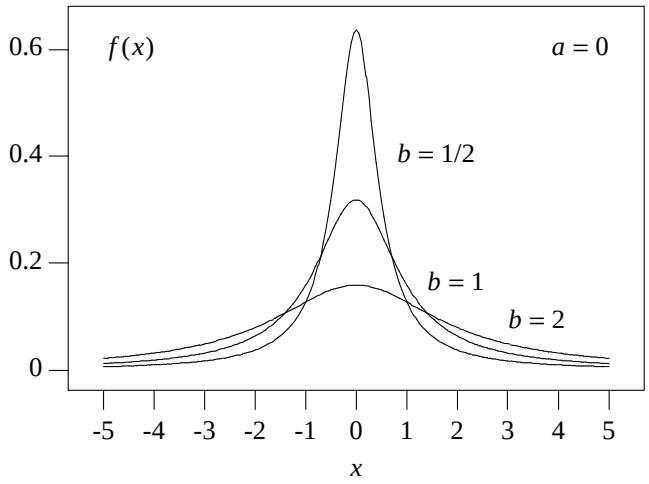


Figure 9. Cauchy Density Functions.

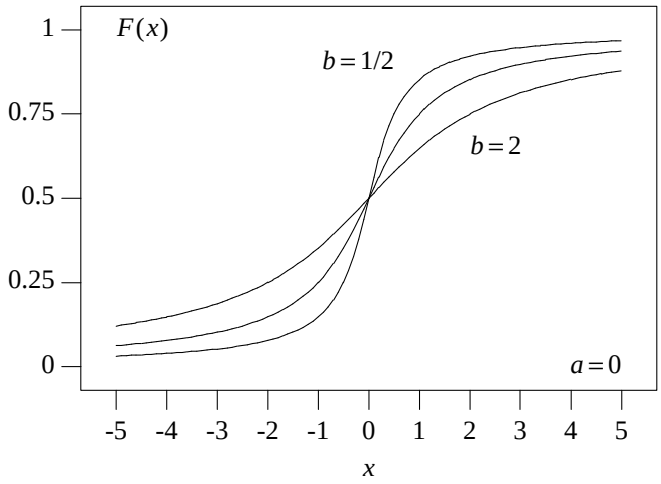


Figure 10. Cauchy Distribution Functions.

5.1.4 Chi-Square

Density Function:

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

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

Distribution Function:

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

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

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

Mode: $\nu - 2$ for $\nu \geq 2$

Mean: ν

Variance: 2ν

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

Source Code:

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

Notes:

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

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 11 and 12, respectively.

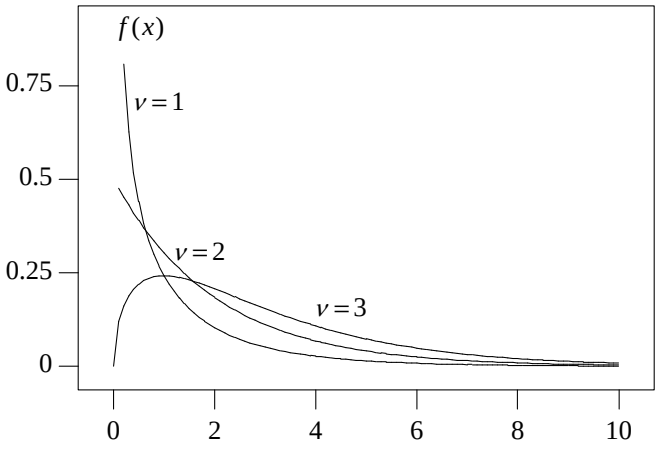


Figure 11. Chi-Square Density Functions.

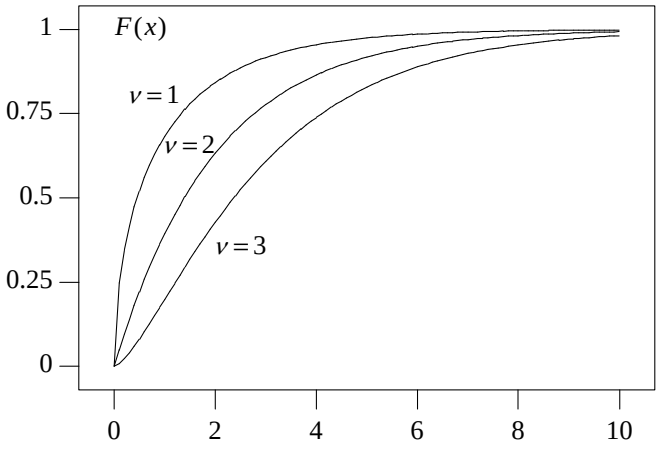


Figure 12. Chi-Square Distribution Functions.

5.1.5 Cosine

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

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

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

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

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

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

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

Variance: $(x_{\max} - x_{\min})^2(\pi^2 - 8)/4\pi^2$

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 $i = 1, 2, \cdots, N$

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

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

The probability density function and the cumulative distribution function are shown in Figures 13 and 14, respectively.

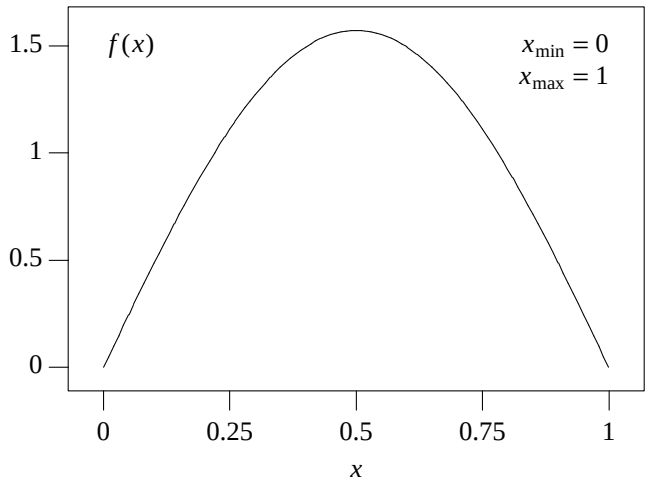


Figure 13. Cosine Density Function.

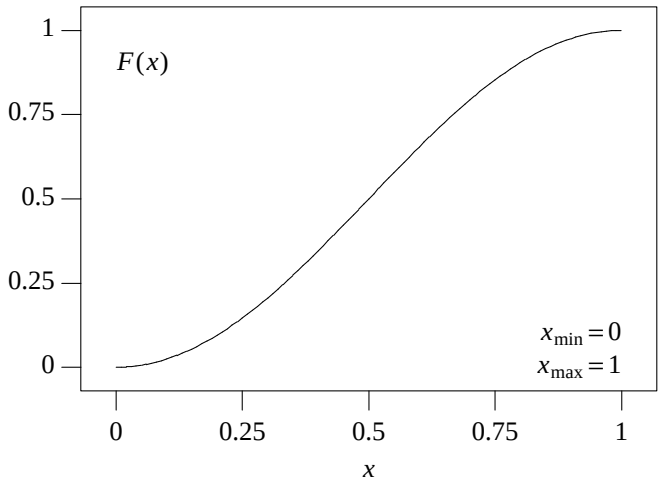


Figure 14. Cosine Distribution Function.

5.1.6 Double Log

Density Function:

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

Distribution Function:

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

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

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

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

Median: a

Mean: a

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

Algorithm: Based on composition and convolution formula 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, $U \sim$ Bernoulli (0.5)
(3) If $U = 1$, return $X = a + b\,U_1 U_2$; otherwise, return $X = a - b\,U_1 U_2$

Source Code:

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

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

The probability density function and the cumulative distribution function are shown in Figures 15 and 16, respectively.

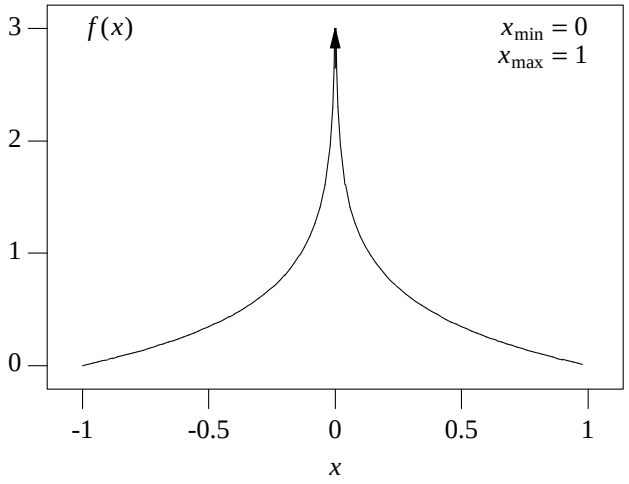


Figure 15. Double Log Density Function.

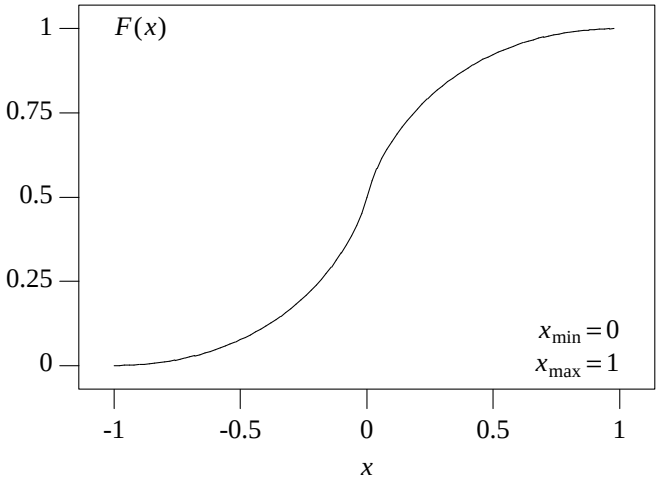


Figure 16. Double Log Distribution Function.

5.1.7 Erlang

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

Distribution Function:
$$F(x) = \begin{cases} 1 - e^{-x/b} \displaystyle\sum_{i=0}^{c-1} \frac{(x/b)^i}{i!} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

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

Source Code:
```
double erlang( double b, int c )
{
   assert( b > 0. && c >= 1 );

   double prod = 1.0;
   for ( int i = 0; i < c; i++ ) prod *= uniform( 0., 1. );
   return -b * log( prod );
}
```

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 17 and 18, respectively.

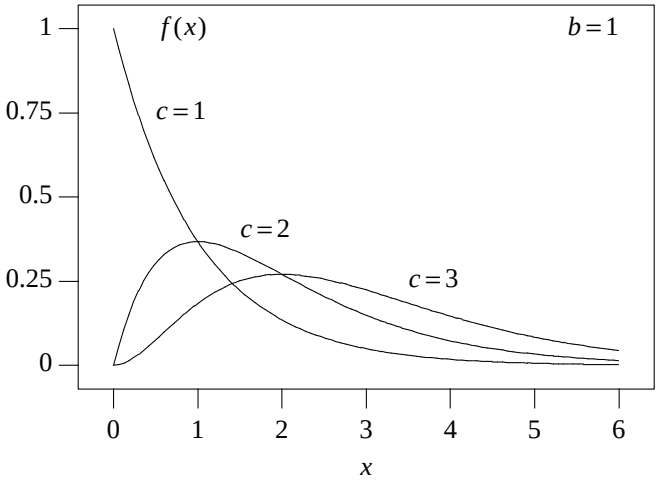


Figure 17. Erlang Density Functions.

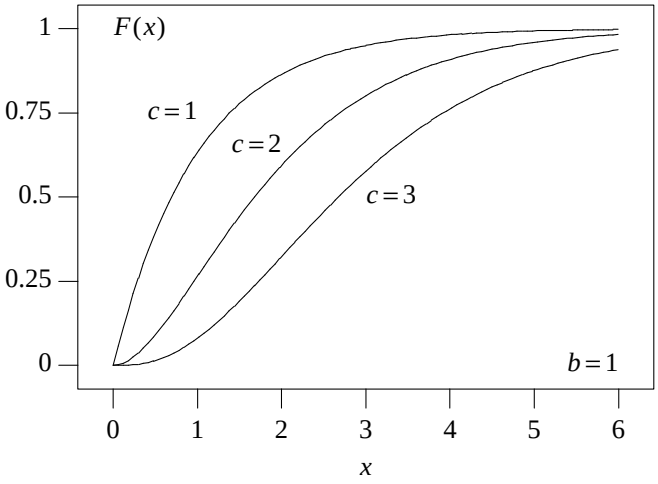


Figure 18. Erlang Distribution Functions.

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 $i = 1, 2, \cdots, N$

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:

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 19 and 20, respectively.

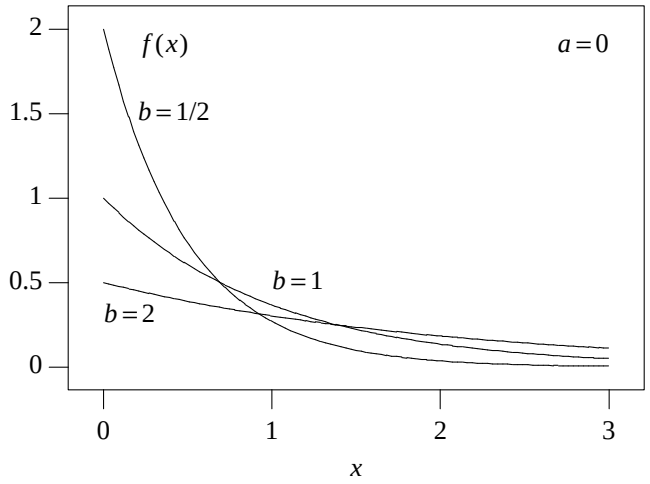


Figure 19. Exponential Density Functions.

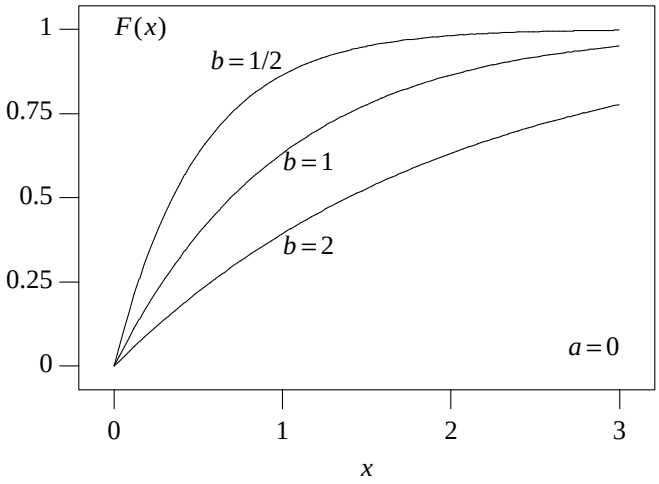


Figure 20. Exponential Distribution Functions.

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 $i = 1, 2, \cdots, N$

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

Source Code:

```
double extremeValue( double a, double b )
{
   assert( b > 0. );
   return a + b * log( -log( uniform( 0., 1. ) ) );
}
```

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 21 and 22, respectively.

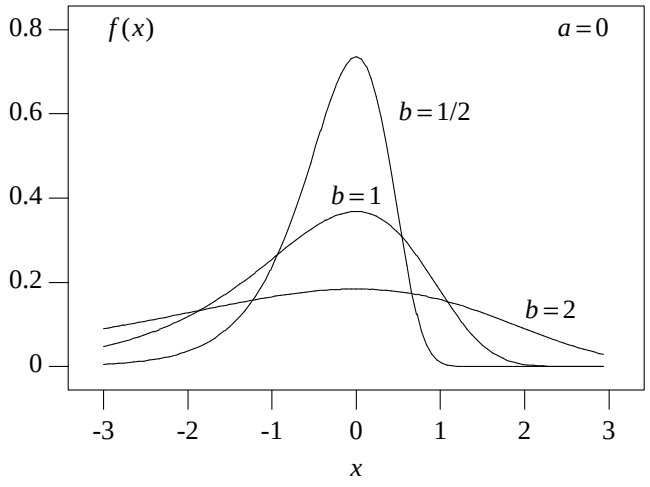


Figure 21. Extreme Value Density Functions.

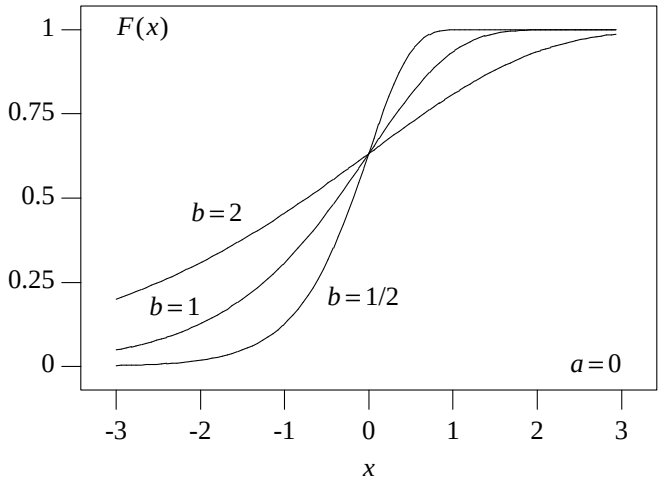


Figure 22. Extreme Value Distribution Functions.

5.1.10 F Ratio

Density Function:

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

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

Distribution Function: No closed form, in general.

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

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

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

Mean: $w/(w-2)$ for $w > 2$

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

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

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

Source Code:

```
double fRatio( int v, int w )
{
   assert( v >= 1 && w >= 1 );
   return ( chiSquare( v ) / v ) / ( chiSquare( w ) / w );
}
```

Examples of the probability density function and the cumulative distribution function are shown in Figures 23 and 24, respectively.

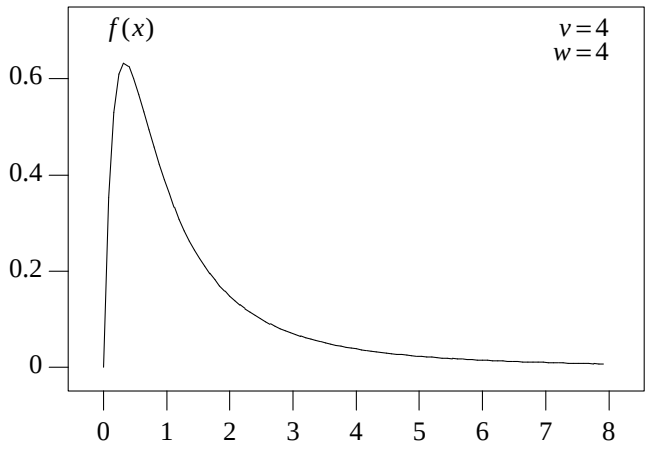


Figure 23. F-Ratio Density Function.

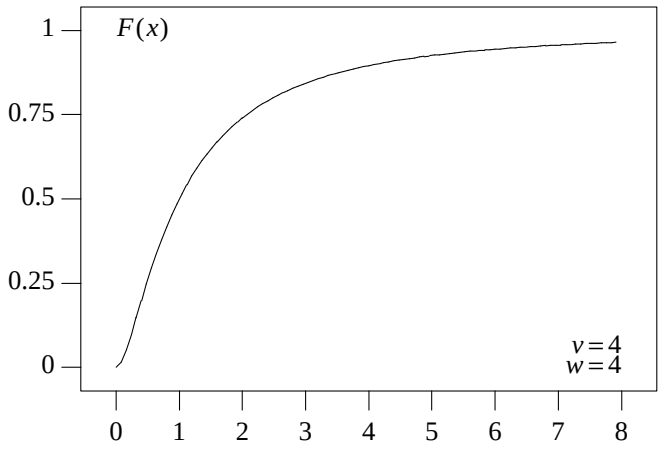


Figure 24. F-Ratio Distribution Function.

5.1.11 Gamma

Density Function:

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

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

If n is an integer, $\Gamma(n) = (n-1)!$

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

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

Input: Location parameter a; scale parameter $b > 0$; shape parameter $c > 0$

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

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

Mean: $a + bc$

Variance: b^2c

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

Case 1: 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.

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

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 $V = \alpha \ln[U_1/(1-U_1)]$, $Y = ce^V$, $Z = U_1^2U_2$, and $W = \beta + qV - Y$.
(3) If $W + d - \theta Z \geq 0$, return $X = Y$; otherwise, proceed to step 4.
(4) If $W \geq \ln Z$, return $X = Y$; otherwise, go back to step 1.

Source Code:

```
double gamma( double a, double b, double c )
{
   assert( b > 0. && c > 0. );

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

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

Notes:

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 25 and 26, respectively.

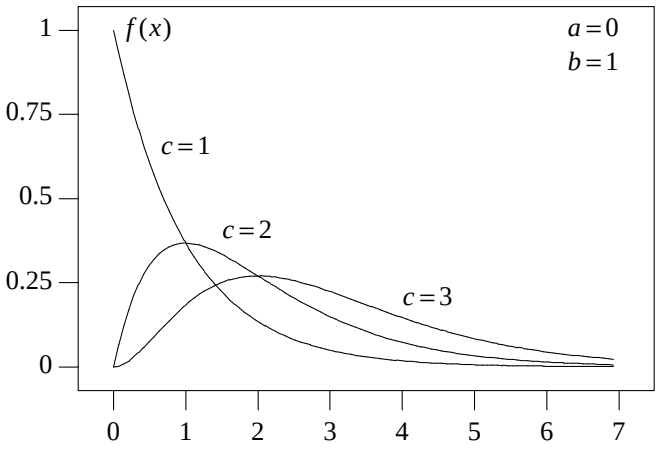


Figure 25. Gamma Density Functions.

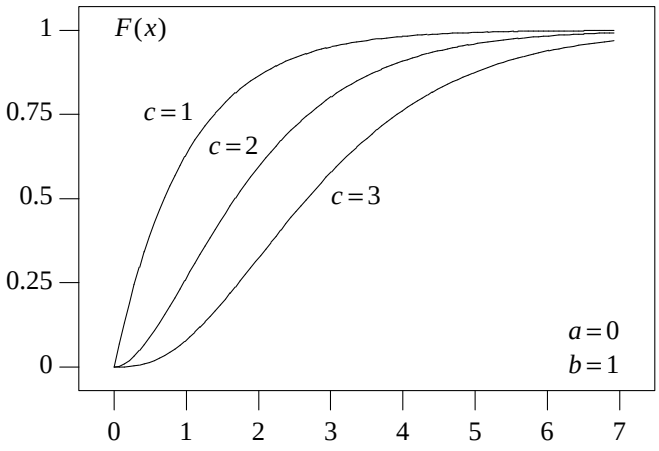


Figure 26. Gamma Distribution Functions.

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

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

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

Mode: a

Median: a

Mean: a

Variance: $2b^2$

Regression Equation: $\begin{cases} \ln(2F_i) = x_i/b - a/b & 0 \le F_i \le 1/2 \\ -\ln[2(1-F_i)] = x_i/b - a/b & 1/2 \le F_i \le 1 \end{cases}$
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \cdots, N$

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

Source Code:

```
double laplace( double a, double b )
{
   assert( b > 0. );

   // composition method

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 27 and 28, respectively.

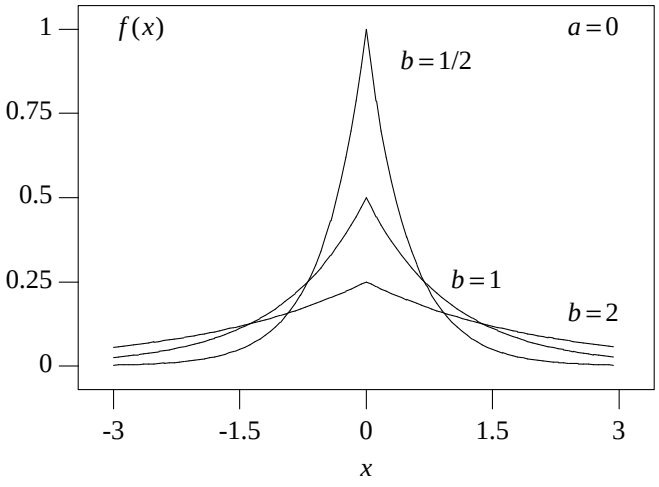


Figure 27. Laplace Density Functions.

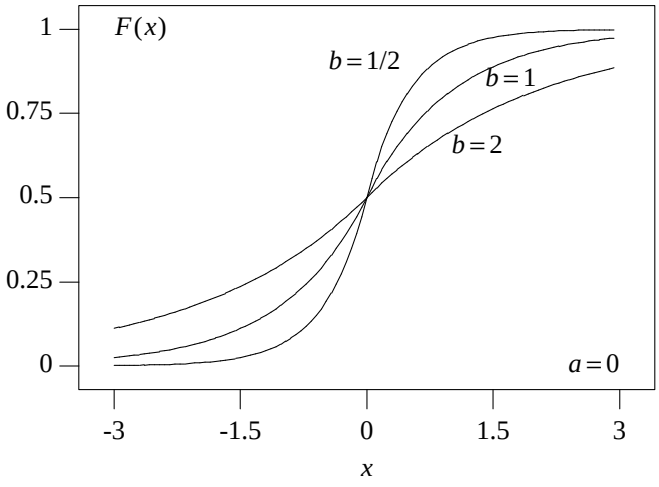


Figure 28. Laplace Distribution Functions.

5.1.13 Logarithmic

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

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

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

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

Mode: $x_{\min}$

Mean: $x_{\min} + \frac{1}{4}(x_{\max} - x_{\min})$

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 $X = a + b\,U_1 U_2$

Source Code:
```
double logarithmic( double xMin, double xMax )
{
   assert( xMin < xMax );

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

The probability density function and the cumulative distribution function are shown in Figures 29 and 30, respectively.

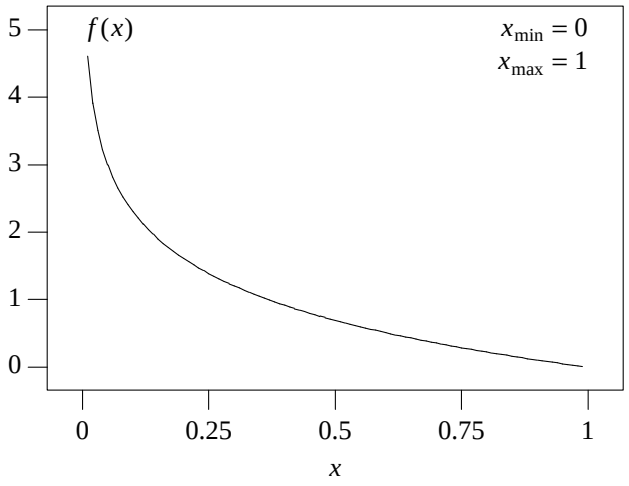


Figure 29. Logarithmic Density Function.

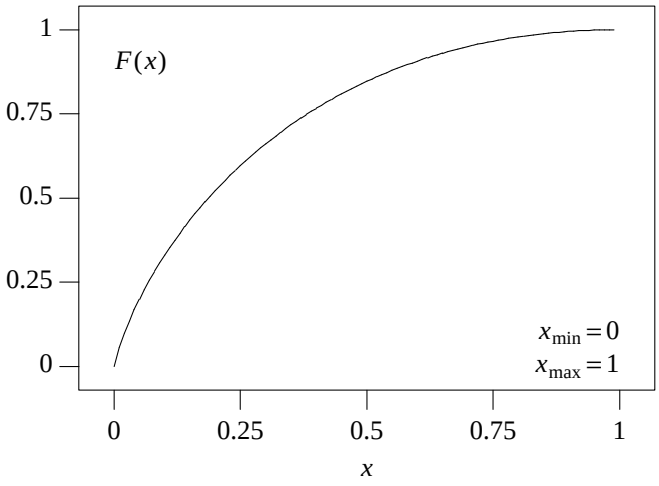


Figure 30. Logarithmic Distribution Function.

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: $\frac{\pi^2}{3} b^2$

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 $i = 1, 2, \cdots, N$

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

Source Code:

```
double logistic( double a, double b )
{
   assert( b > 0. );
   return a - b * log( 1. / uniform( 0., 1. ) - 1. );
}
```

Examples of probability density functions and cumulative distribution functions are shown in Figures 31 and 32, respectively.

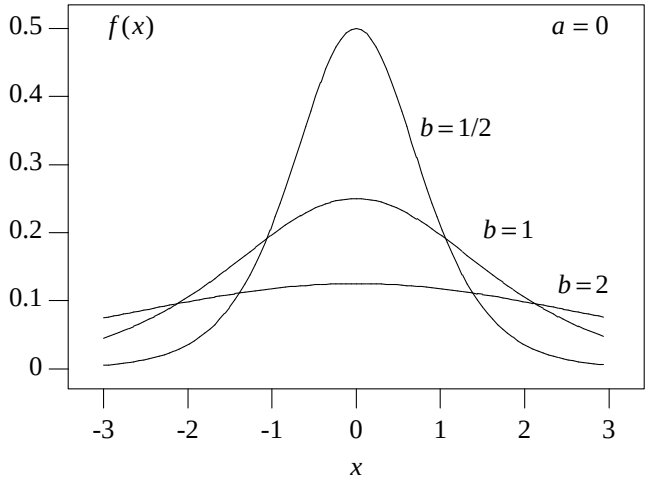


Figure 31. Logistic Density Functions.

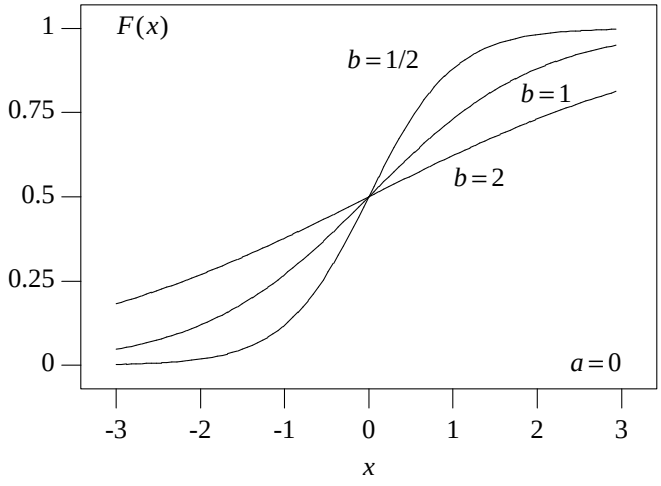


Figure 32. Logistic Distribution Functions.

5.1.15 Lognormal

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

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

Input: Location parameter a, any real number, merely shifts the origin; shape parameter $\sigma > 0$; scale parameter μ is 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: $\operatorname{erf}^{-1}(2F_i - 1) = \dfrac{1}{\sqrt{2}\sigma}\ln(x_i - a) - \dfrac{\mu}{\sqrt{2}\sigma}$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \cdots, N$

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

Algorithm:
(1) Generate $V \sim N(\mu, \sigma^2)$
(2) Return $X = a + e^V$

Source Code:
```
double lognormal( double a, double mu, double sigma )
{
   return a + exp( normal( mu, sigma ) );
}
```

Notes: $X \sim LN(\mu, \sigma^2)$ if and only if $\ln X \sim N(\mu, \sigma^2)$. Note that μ and σ^2 are not the mean and variance of the $LN(\mu, \sigma^2)$ distribution. To generate a lognormal random variate with *given* $\hat{\mu}$ and $\hat{\sigma}^2$, set $\mu = \ln(\hat{\mu}^2/\sqrt{\hat{\mu}^2 + \hat{\sigma}^2})$ and $\sigma^2 = \ln[(\hat{\mu}^2 + \hat{\sigma}^2)/\hat{\mu}^2]$.

Examples of probability density functions and cumulative distribution functions are shown in Figures 33 and 34, respectively.

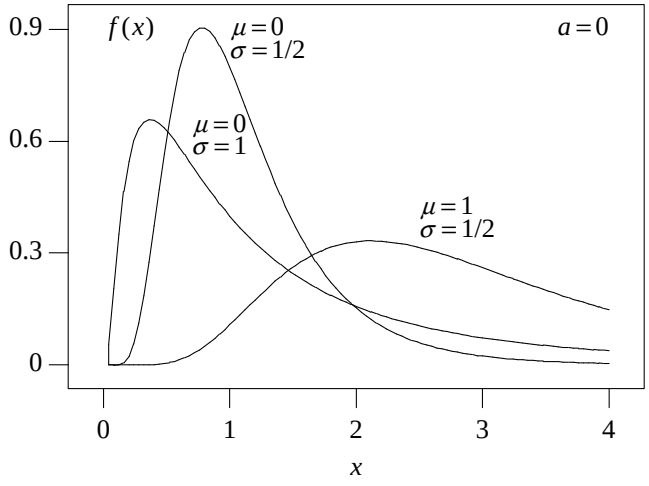


Figure 33. Lognormal Density Functions.

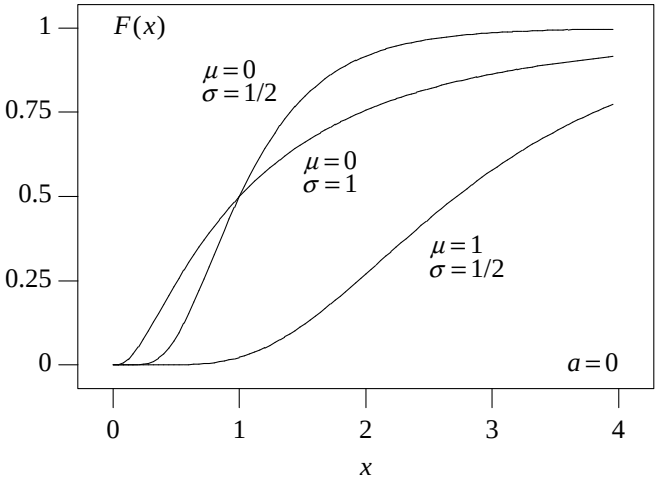


Figure 34. Lognormal Distribution Functions.

5.1.16 Normal (Gaussian)

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

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

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: $\text{erf}^{-1}(2F_i - 1) = x_i/\sqrt{2}\sigma - \mu/\sqrt{2}\sigma$,
where the x_i are arranged in ascending order, $F_i = i/N$, and $i = 1, 2, \cdots, N$

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

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

Source Code:
```
double normal( double mu, double sigma )
{
   assert( sigma > 0. );
   double p, p1, p2;
   do {
      p1 = uniform( -1., 1. );
      p2 = uniform( -1., 1. );
      p = p1 * p1 + p2 * p2;
   } while ( p >= 1. );
   return mu + sigma * p1 * sqrt( -2. * log( p ) / p );
}
```

Notes: If $X \sim N(\mu, \sigma^2)$, then $\exp(X) \sim LN(\mu, \sigma^2)$, the *lognormal distribution*.

The probability density function and cumulative distribution function for the standard normal are shown in Figures 35 and 36, respectively.

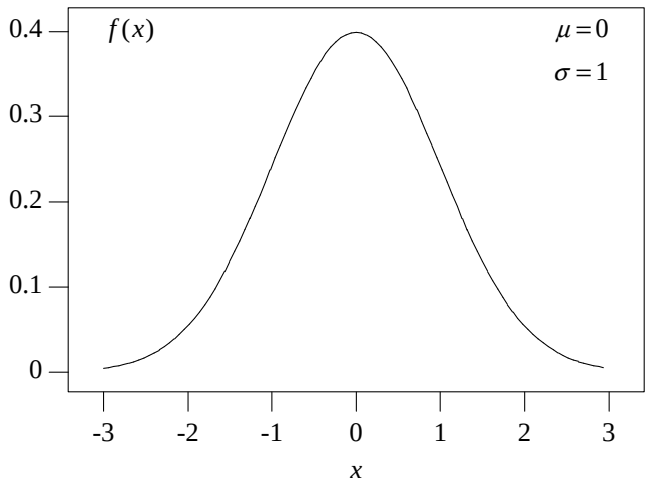


Figure 35. Normal Density Function.

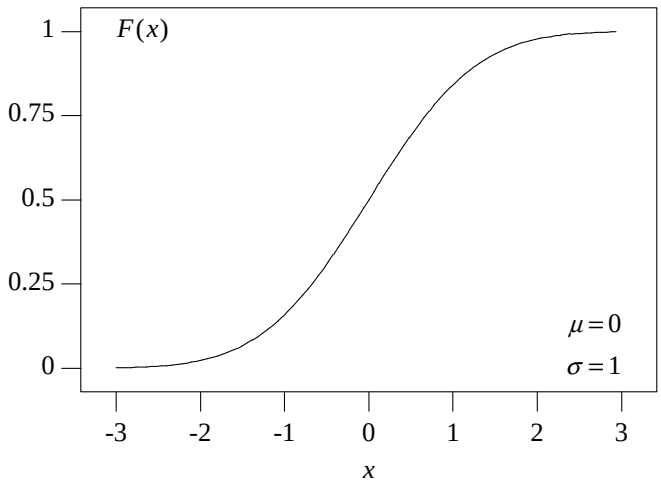


Figure 36. Normal Distribution Function.

5.1.17 Parabolic

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

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

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

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

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

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

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

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

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

Source Code:

```
// function to call for the Monte Carlo sampling
double parabola( double x, double xMin, double xMax )
{
   if ( x < xMin || x > xMax ) return 0.0;

   double a    = 0.5 * ( xMin + xMax );   // location parameter
   double b    = 0.5 * ( xMax - xMin );   // scale parameter
   double yMax = 3. / ( 4. * b );

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

// function which generates parabolic distribution
double parabolic( double xMin, double xMax )
{
   assert( xMin < xMax );

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

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

Notes: The parabolic distribution is a special case of the *beta* distribution (when $v = w = 2$).

The probability density function and the cumulative distribution function are shown in Figures 37 and 38, respectively.

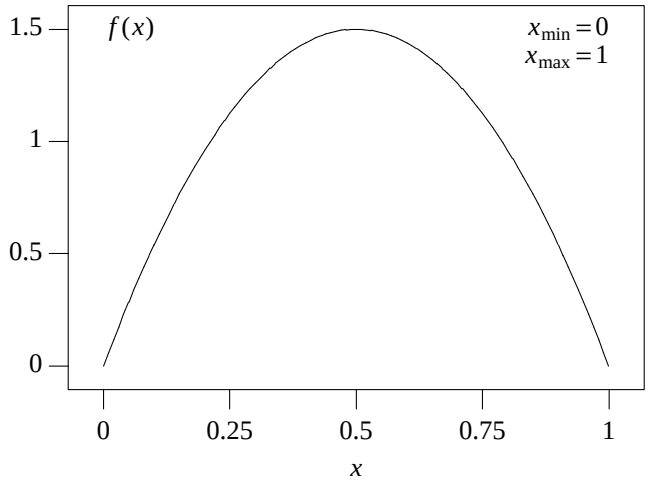


Figure 37. Parabolic Density Function.

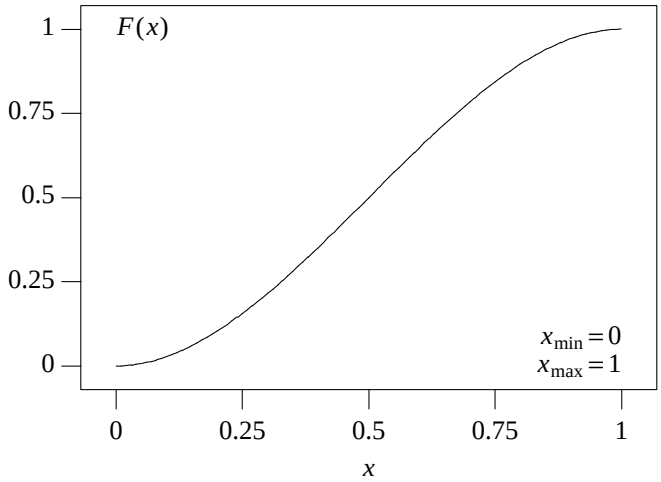


Figure 38. Parabolic Distribution Function.

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 $i = 1, 2, \cdots, N$

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

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

Source Code:

```
double pareto( double c )
{
   assert( c > 0. );
   return pow( uniform( 0., 1. ), -1. / c );
}
```

Examples of probability density functions and cumulative distribution functions are shown in Figures 39 and 40, respectively.

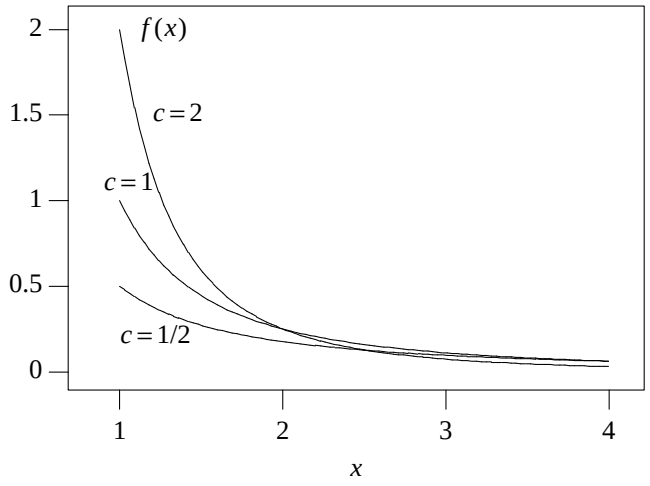


Figure 39. Pareto Density Functions.

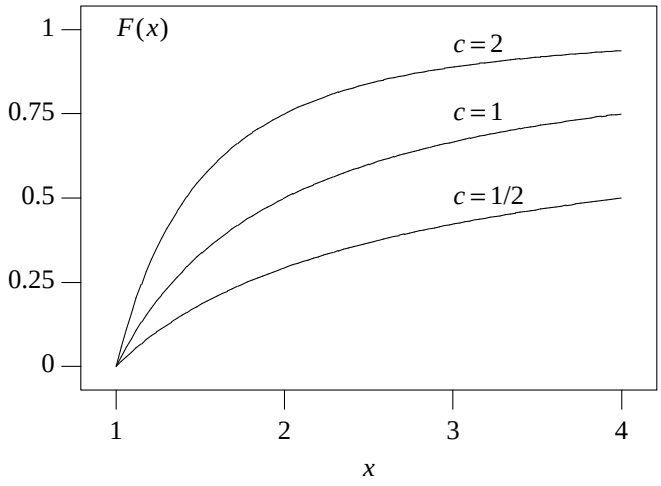


Figure 40. Pareto Distribution Functions.

5.1.19 Pearson's Type 5 (Inverted Gamma)

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

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

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

where $\Gamma(a, z)$ is the incomplete gamma function, 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:
```
double pearson5( double b, double c )
{
   assert( b > 0. && c > 0. );
   return 1. / gamma( 0., 1. / b , c);
}
```

Notes: $X \sim \text{PearsonType5}(b, c)$ 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*.

Examples of probability density functions and cumulative distribution functions are shown in Figures 41 and 42, respectively.

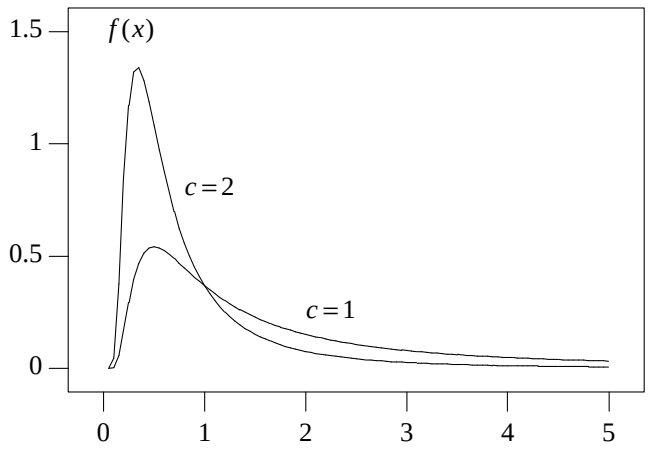


Figure 41. Pearson Type 5 Density Functions.

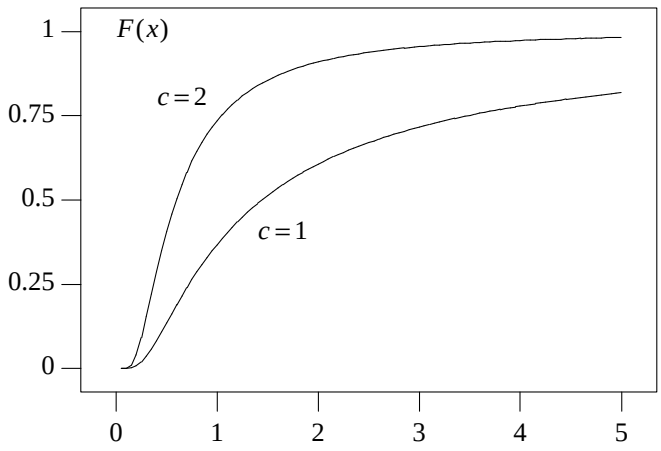


Figure 42. Pearson Type 5 Distribution Functions.

5.1.20 Pearson's Type 6

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

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

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

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

Input: Scale parameter $b > 0$; shape parameters $v > 0$ and $w > 0$

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

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

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

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

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

Source Code
```
double pearson6( double b, double v, double w )
{
   assert( b > 0. && v > 0. && w > 0. );
   return gamma( 0., b, v) / gamma( 0., b, w );
}
```

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 43 and 44, respectively.

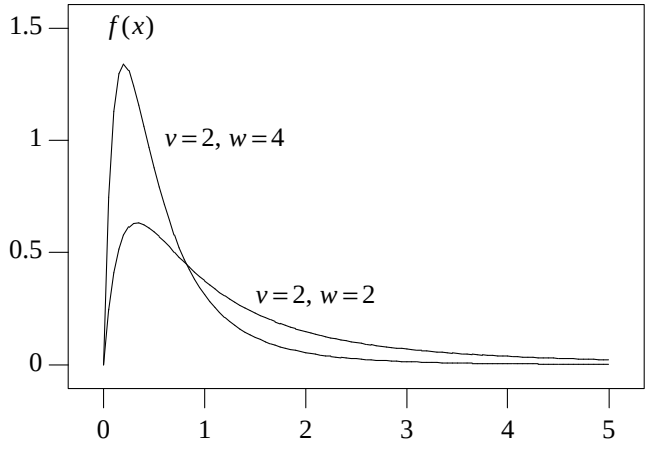


Figure 43. Pearson Type 6 Density Functions.

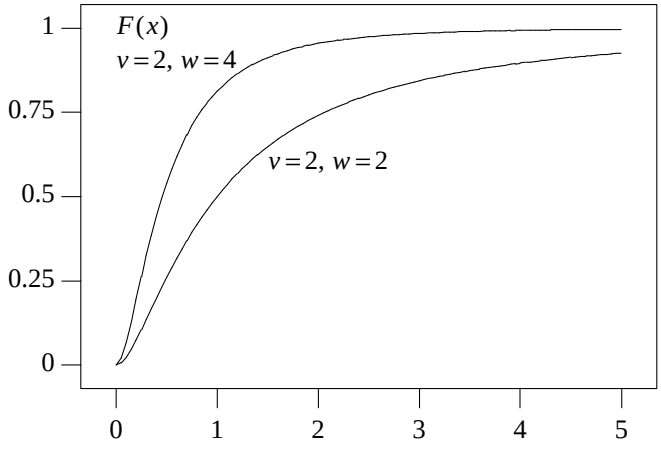


Figure 44. Pearson Type 6 Distribution Functions.

5.1.21 Power

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

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

Input: Shape parameter $c > 0$

Output: $x \in [0, 1)$

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

Median: $2^{-1/c}$

Mean: $\dfrac{c}{(c+1)}$

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

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

Maximum Likelihood: $c = -\left(\dfrac{1}{N} \sum_{i=1}^{N} \ln x_i \right)^{-1}$

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

Source Code:
```
double power( double c )
{
   assert( c > 0. );
   return pow( uniform( 0., 1. ), 1. / c );
}
```

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 45 and 46, respectively.

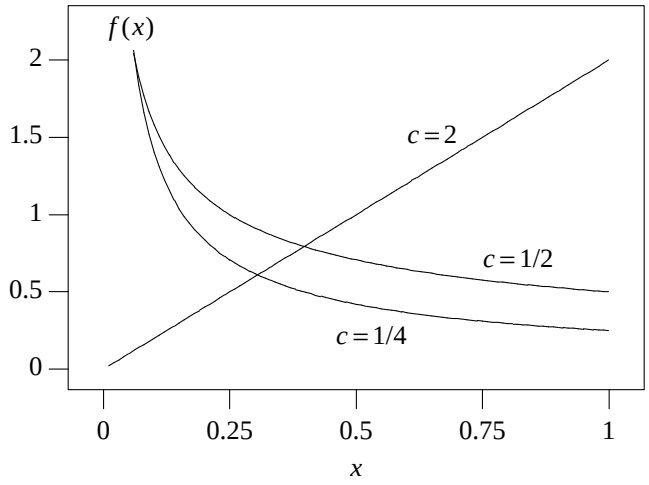


Figure 45. Power Density Functions.

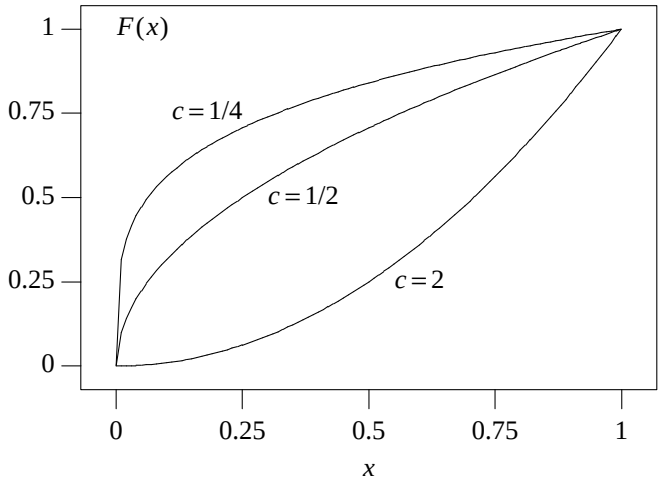


Figure 46. Power Distribution Functions.

5.1.22 Rayleigh

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

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

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 $i = 1, 2, \cdots, N$

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:
```
double rayleigh( double a, double b )
{
   assert( b > 0. );
   return a + b * sqrt( -log( uniform( 0., 1. ) ) );
}
```

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

Examples of the probability density function and the cumulative distribution function are shown in Figures 47 and 48, respectively.

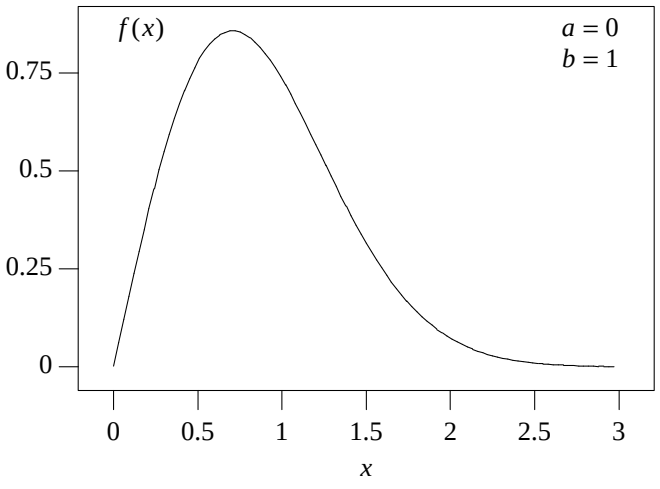


Figure 47. Rayleigh Density Function.

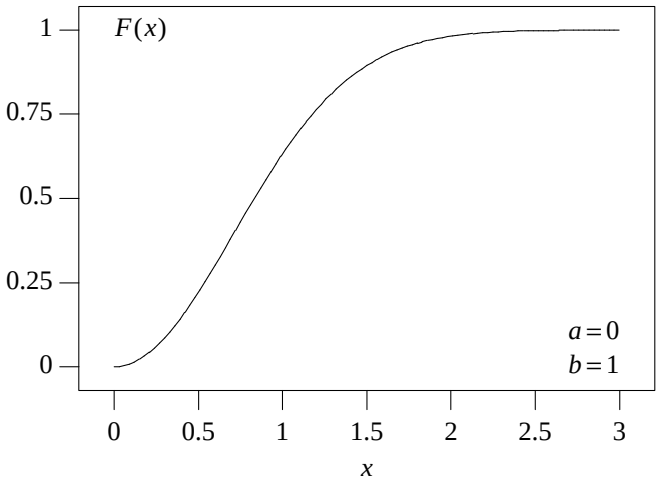


Figure 48. Rayleigh Distribution Function.

5.1.23 Student's t

Density Function: $f(x) = \dfrac{\Gamma[(\nu+1)/2]}{\sqrt{\pi\nu}\,\Gamma(\nu/2)}\left(1+\dfrac{x^2}{\nu}\right)^{-(\nu+1)/2} \qquad -\infty < x < \infty$

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

Distribution Function: No closed form, in general

Input: Shape parameter ν, a positive integer (number of degrees of freedom)

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

Mode: 0

Median: 0

Mean: 0

Variance: $\nu/(\nu-2)$ for $\nu > 2$

Algorithm:
(1) Generate $Y \sim N(0,1)$ and $Z \sim \chi^2(\nu)$
(2) Return $X = Y/\sqrt{Z/\nu}$

Source Code:

```
double studentT( int df )
{
   assert( df >= 1 );

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

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

Examples of the probability density function and the cumulative distribution function are shown in Figures 49 and 50, respectively.

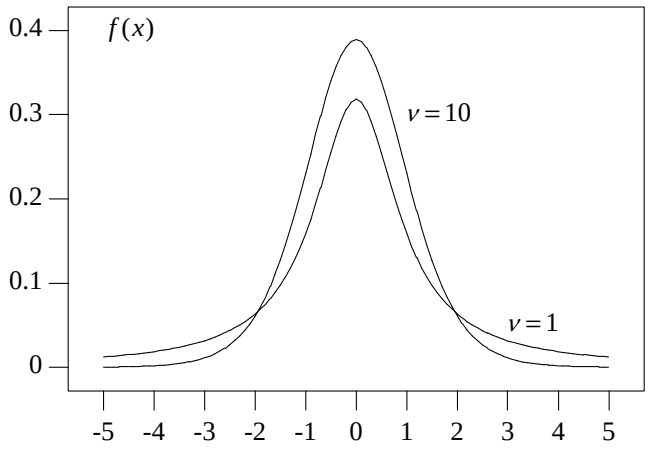


Figure 49. Student's t Density Functions.

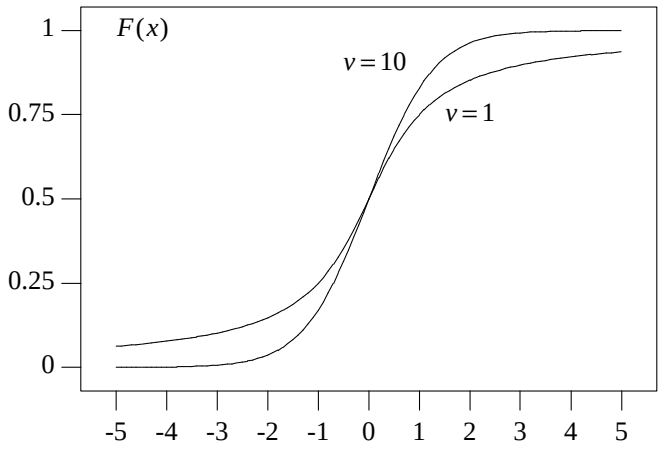


Figure 50. Student's t Distribution Functions.

5.1.24 Triangular

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

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

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

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

Source Code:

```
double triangular( double xMin, double xMax, double c )
{
   assert( xMin < xMax && xMin <= c && c <= xMax );

   double p = uniform( 0., 1. ), q = 1. - p;

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

Examples of probability density functions and cumulative distribution funtions are shown in Figures 51 and 52, respectively.

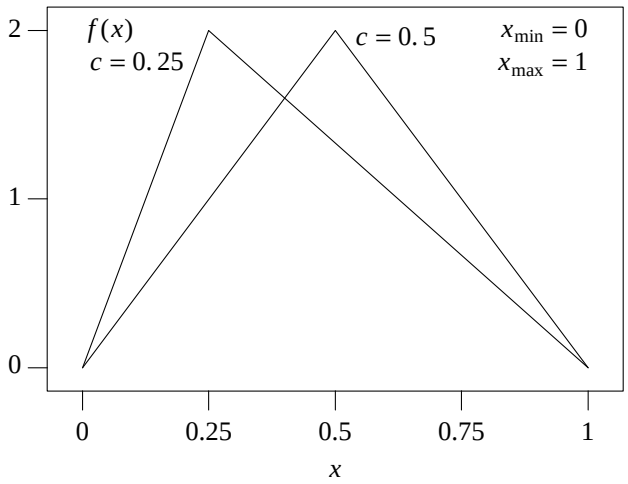


Figure 51. Triangular Density Functions.

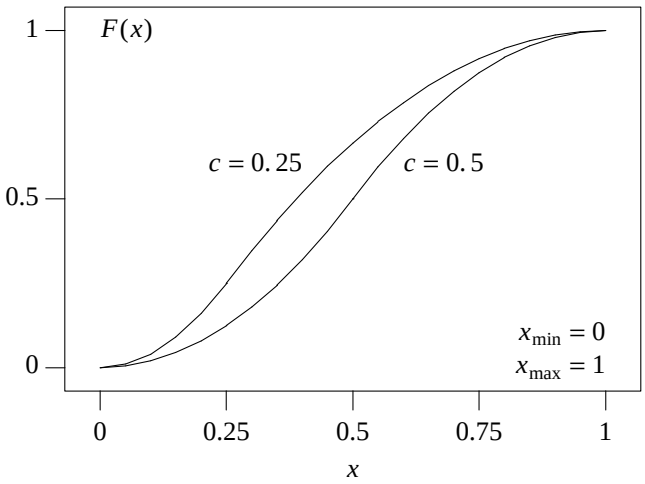


Figure 52. Triangular Distribution Functions.

5.1.25 Uniform

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

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

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

Source Code:
```
double uniform( double xMin, double xMax )
{
   assert( xMin < xMax );
   return xMin + ( xMax - xMin ) * _u();
}
```

Notes:
(1) The source code for **_u()** referenced above is given in section 6.
(2) The uniform distribution is the basis for most distributions in the Random class.
(3) The uniform distribution is a special case of the *beta* distribution (when $v = w = 1$).

The probability density function and the cumulative distribution function are shown in Figures 53 and 54, respectively.

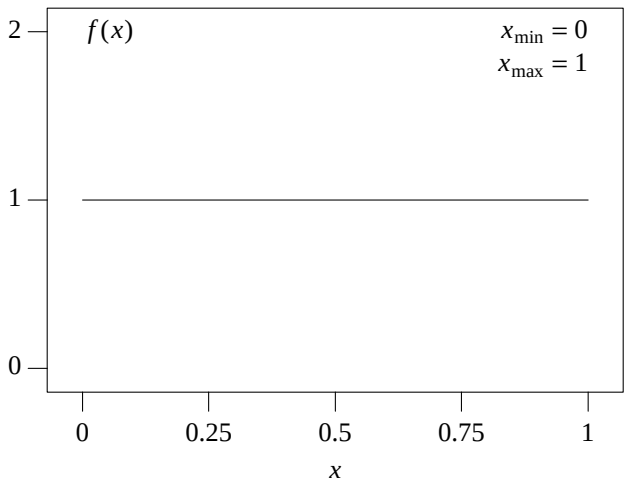


Figure 53. Uniform Density Function.

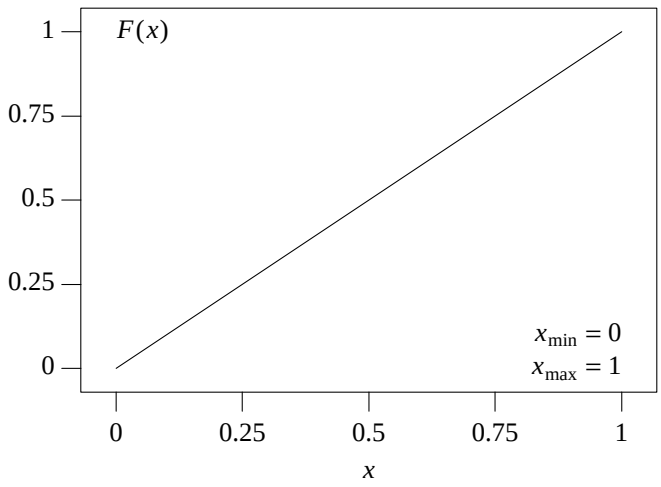


Figure 54. Uniform Distribution Function.

5.1.26 User-Specified

Density Function: User-specified, nonnegative function $f(x)$

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

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

Algorithm: (1) Generate $A \sim U(0, A_{\max})$ 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 doman and range
(2) Return $X = x_{\min} + A/(y_{\max} - y_{\min})$ if $f(X) \le Y$; otherwise, go back to step 1

Source Code:

```
double userSpecified( double( *usf )( double,     // function
                                      double,     // xMin
                                      double ),   // xMax
                      double xMin, double xMax,   // domain
                      double yMin, double yMax )  // range
{
   assert( xMin < xMax && yMin < yMax );
   double x, y, areaMax = ( xMax - xMin ) * ( yMax - yMin );

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

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

Notes: In order to qualify as a true probaility 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 nonnegative 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 arbirarily 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.

Examples of a user-specified bimodal probability density and the corresponding distribution are shown in Figures 55 and 56, respectively. Note that it is not necessary to have knowledge of $F(x)$, only $f(x)$ and that the function $f(x)$ can be arbitarily complex.

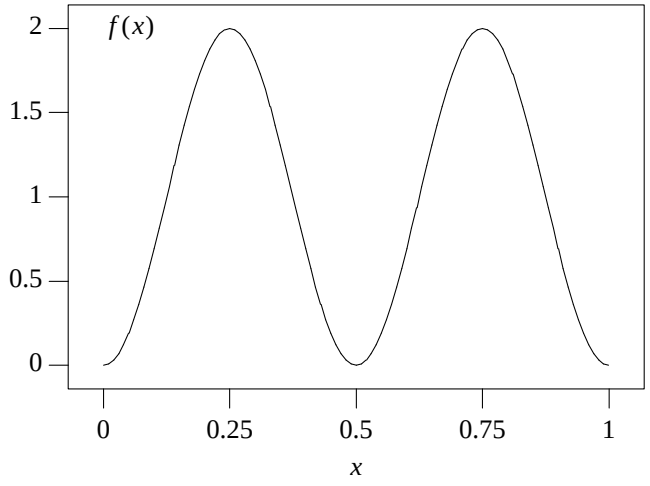


Figure 55. User-Specified Density Function.

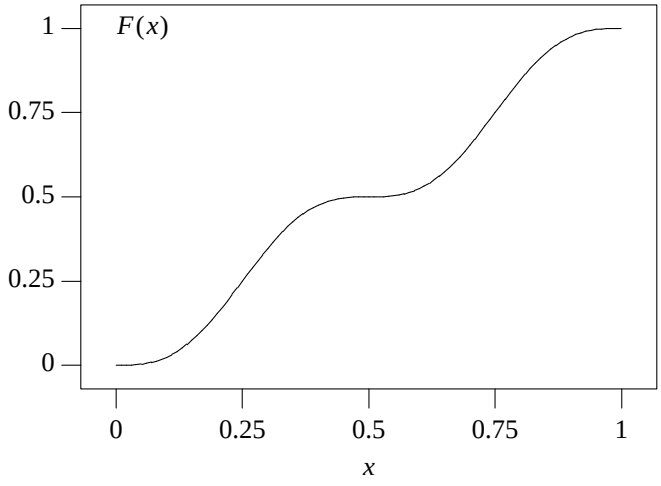


Figure 56. User-Specified Distribution Function.

5.1.27 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 $i = 1, 2, \cdots, N$

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

Source Code:
```
double weibull( double a, double b, double c )
{
   assert( b > 0. && c > 0. );
   return a + b * pow( -log( uniform( 0., 1. ) ), 1. / c );
}
```

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

Examples of probability density functions and cumulative distribution functions are shown in Figures 57 and 58, respectively.

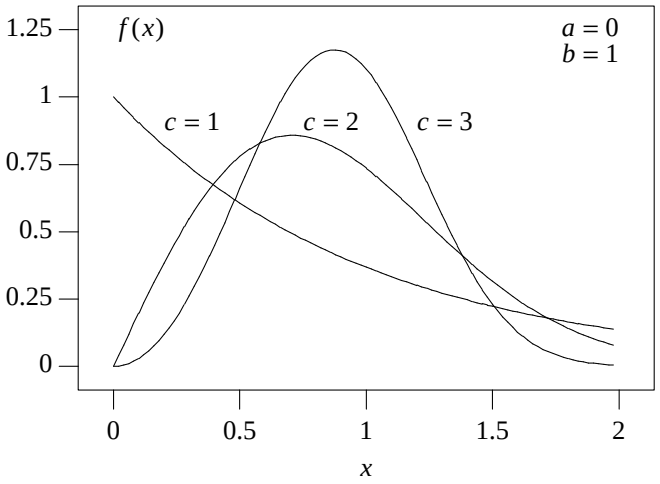


Figure 57. Weibull Density Functions.

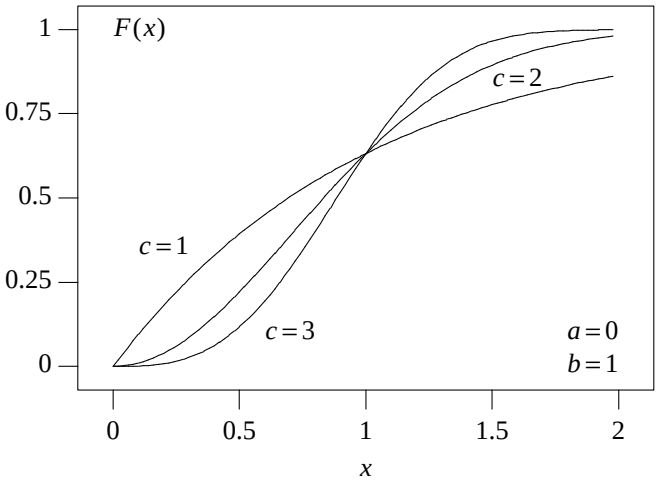


Figure 58. Weibull Distribution Functions.

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 ($0 \le k \le n$)
Geometric	p	number of trials before first success ($0 \le k < \infty$)
Hypergeometric	n, N, and K	number of successes ($0 \le k \le \min(n, K)$)
Multinomial	n, m, $p_1, \cdots, p_m$	number of successes of each event ($1 \le k_i \le m$)
Negative Binomial	p and K	number of failures before K accumulated successes ($0 \le k < \infty$)
Pascal	p and K	number of trials before K accumulated successes ($1 \le k < \infty$)
Poisson	μ	number of successes ($0 \le k < \infty$)
Uniform Discrete	i and j	integer selected ($i \le k \le j$)

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 on a single trial is p. It forms the basis for a number of other discrete distributions.

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

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

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

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

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

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

Source Code:

```
bool bernoulli( double p )
{
   assert( 0. <= p && p <= 1. );

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

Notes:
(1) Notice that if p is strictly zero, then 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

The binomial distribution represents the probability of k successes in n independent Bernoulli trials, where the probability of success in a single trial is p.

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

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

where the *binomial coefficient* $\binom{n}{i} = \frac{n!}{i!(n-i)!}$

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

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

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

Mean: np

Variance: $np(1-p)$

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

Algorithm:
(1) Generate n IID Bernoulli trials $X_i \sim$ Bernoulli(p), where $i = 1, \cdots, n$
(2) Return $X = X_1 + \cdots + X_n$

Source Code:
```
int binomial( int n, double p )
{
   assert( 0. <= p && p <= 1. && n >= 1 );
   int sum = 0;
   for ( int i = 0; i < n; i++ ) sum += bernoulli( p );
   return sum;
}
```

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

Examples of the probability density function and the cumulative distribution function are shown in Figures 59 and 60, respectively.

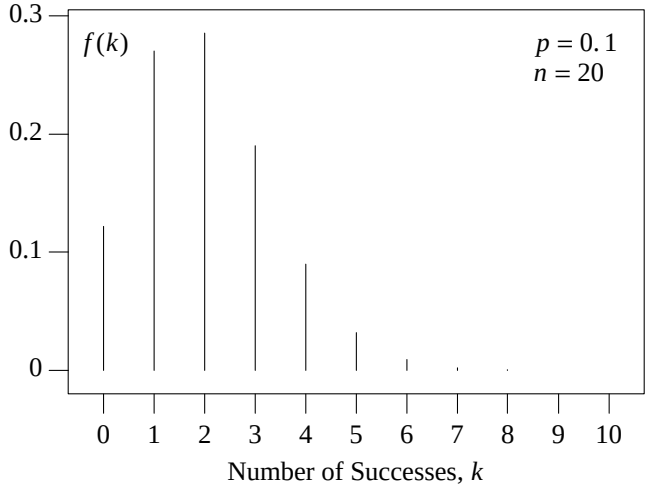


Figure 59. Binomial Density Function.

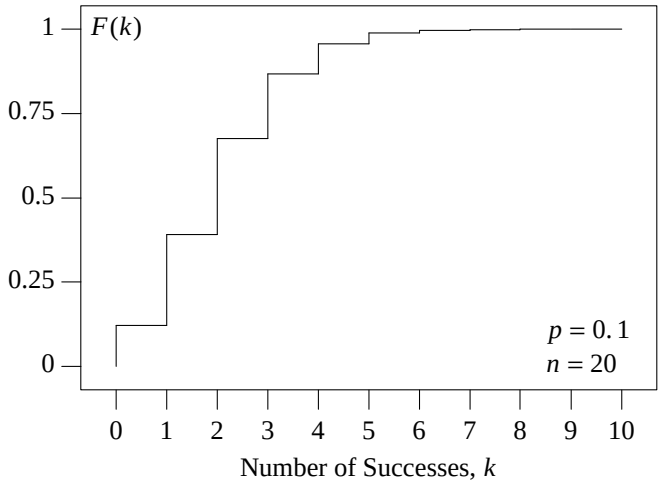


Figure 60. Binomial Distribution Function.

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 (i.e., the success itself is not counted).

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

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

Input: Probability of event, p, where $0 < p < 1$

Output: Number of trials before a success $k \in \{0, 1, \cdots\}$

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:

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

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

Examples of the probability density function and the cumulative distribution function are shown in Figures 61 and 62, respectively.

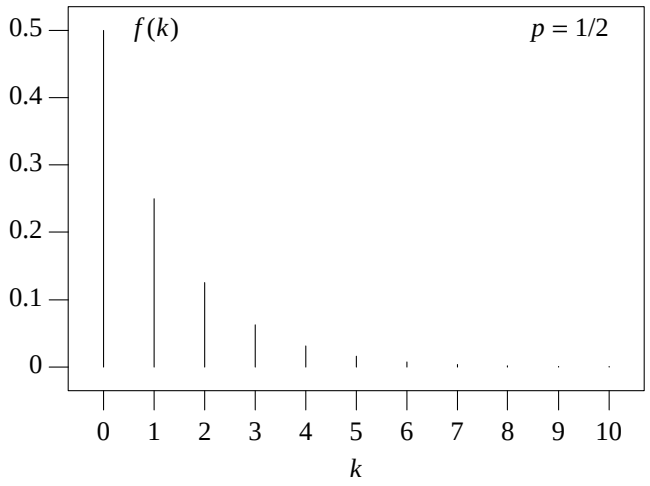


Figure 61. Geometric Density Function.

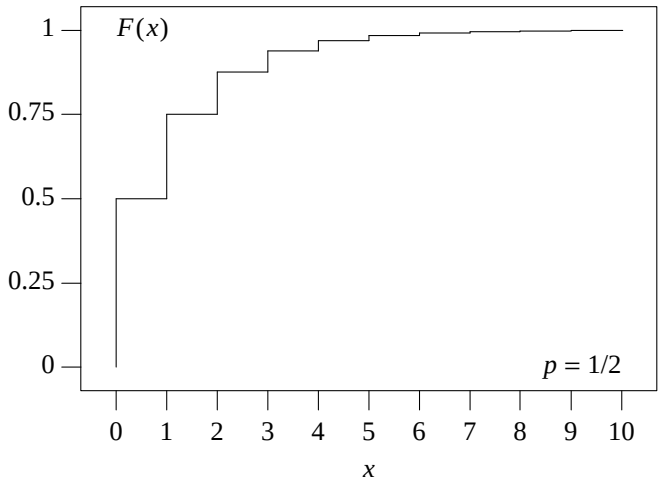


Figure 62. Geometric Distribution Function.

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 contains K successes.

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

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

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

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

Mean: np, where $p = K/N$

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

Source Code:

```
int hypergeometric( int n, int N, int K )
{
   assert( 0 <= n && n <= N );
   assert( N >= 1 && K >= 0 );

   int count = 0;
   for ( int i = 0; i < n; i++, N-- ) {

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

Notes: hypergeometric $(n, N, K) \approx$ binomial $(n, K/N)$, provided $n/N < 0.1$

Examples of the probability density function and the cumulative distribution function are shown in Figures 63 and 64, respectively.

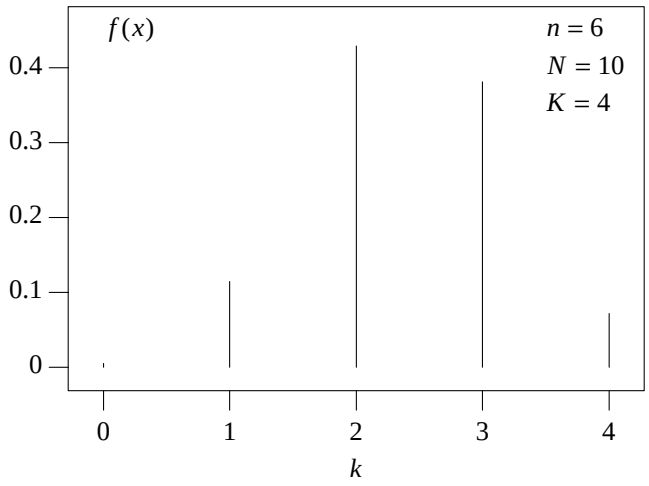


Figure 63. Hypergeometric Density Function.

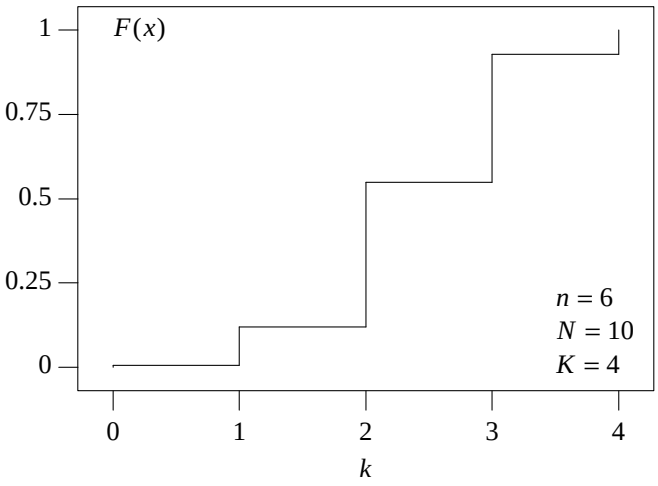


Figure 64. Hypergeometric Distribution Function.

5.2.5 Multinomial

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

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

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

Output: Number of times each of the m events occurs, $k_i \in \{0, \cdots, n\}$,
where $i = 1, \cdots, m$, and $k_1 + \cdots + k_m = n$

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

Source Code:

```
void multinomial( int    n,          // trials n
                  double p[],       // probability vector p,
                  int    count[],   // success vector count,
                  int    m )        // number of disjoint events m
{
   assert( m >= 2 );
   double sum = 0.;
   for ( int bin = 0; bin < m; bin++ ) sum += p[ bin ];
   assert( sum == 1. );

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

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

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

      double lower = 0., upper = 0., u = _u();

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

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

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

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

5.2.6 Negative Binomial

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

Density Function:
$$f(k) = \begin{cases} \dfrac{(s+k-1)!}{k!(s-1)!}\, p^s (1-p)^k & k \in \{0, 1, \cdots\} \\ 0 & \text{otherwise} \end{cases}$$

Distribution Function:
$$F(k) = \begin{cases} \displaystyle\sum_{i=0}^{k} \frac{(s+i-1)!}{i!(s-1)!}\, p^k (1-p)^i & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

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

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

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 $X = X_1 + \cdots + X_s$

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

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

Examples of the probability density function and the cumulative distribution function are shown in Figures 65 and 66, respectively.

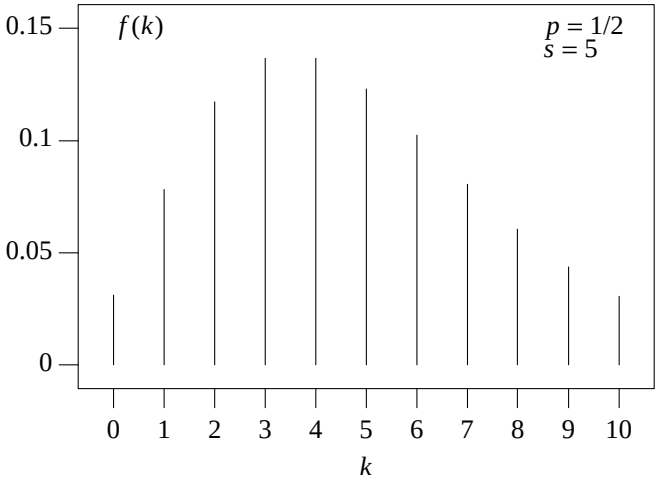


Figure 65. Negative Binomial Density Function.

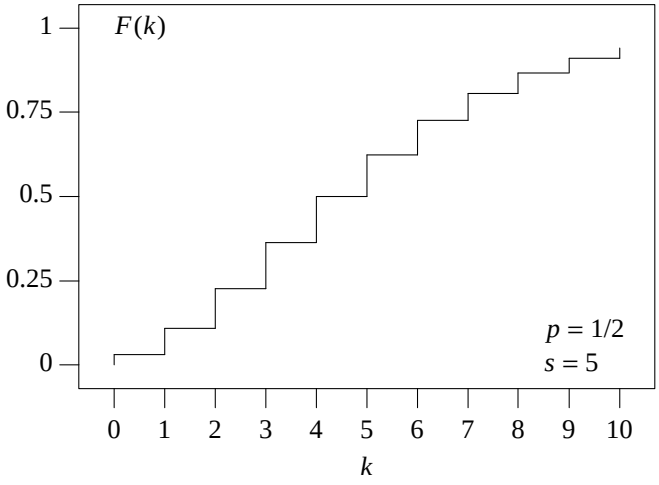


Figure 66. Negative Binomial Distribution Function.

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

Distribution Function:
$$F(k) = \begin{cases} \displaystyle\sum_{i=1}^{k} \frac{(k-1)!}{(i-s)!(s-1)!}\, p^s (1-p)^{i-s} & k \geq s \\ 0 & \text{otherwise} \end{cases}$$

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

Output: The number of trials $k \in \{s, s+1, \cdots\}$

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 [unbiassed estimate is $(s-1)/(n-1)$]

Algorithm: This algorithm takes advantage of the logical relationship to the negative binomial. Return $X = \text{negativeBinomial}(s, p) + s$

Source Code:

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

Notes:

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

(3) $\text{Pascal}(p, 1) = \text{geometric}(p) + 1$.

Examples of the probability density function and the cumulative distribution function are shown in Figures 67 and 68, respectively.

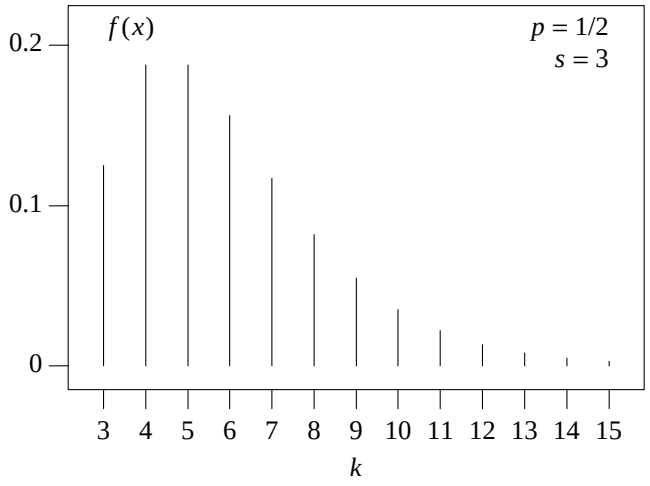


Figure 67. Pascal Density Function.

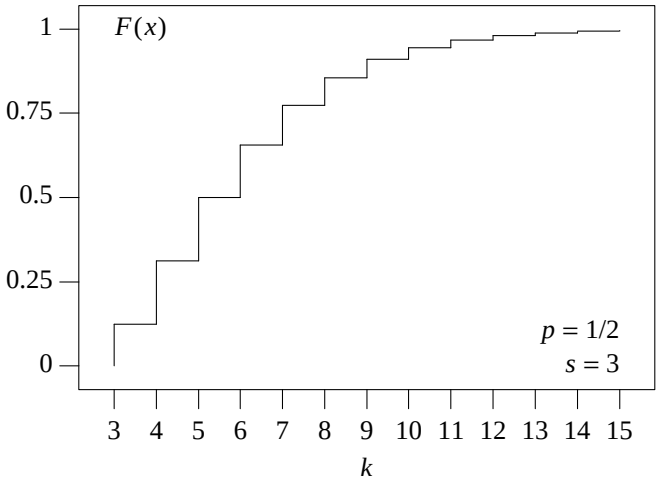


Figure 68. Pascal Distribution Function.

5.2.8 Poisson

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

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

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

Input: Rate of occurrence, $\mu > 0$

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

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

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

Examples of the probability density function and the cumulative distribution function are shown in Figures 69 and 70, respectively.

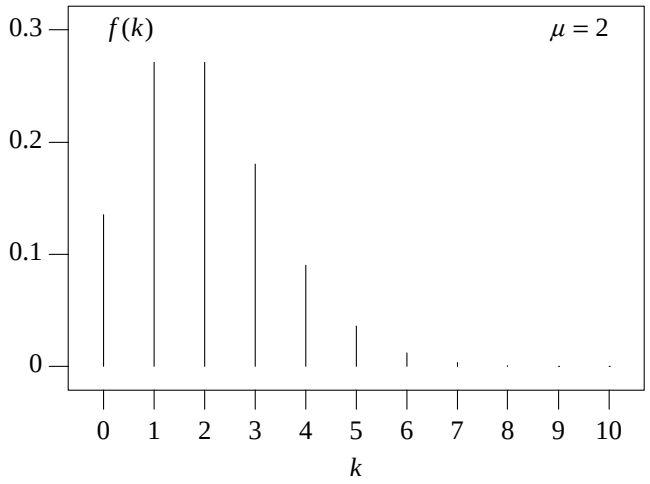


Figure 69. Poisson Density Function.

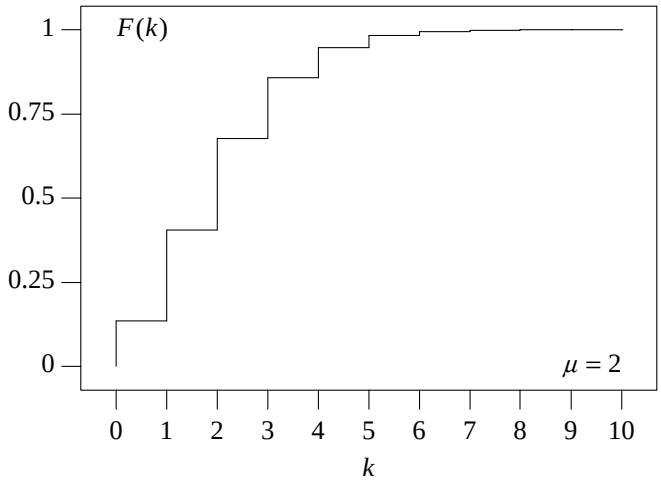


Figure 70. Poisson Distribution Function.

5.2.9 Uniform Discrete

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

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

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

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

Output: $k \in \{i_{\min}, \cdots, i_{\max}\}$

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

Mean: $\dfrac{1}{2}(i_{\min} + i_{\max})$

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

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

Source Code:
```
int uniformDiscrete( int i, int j )
{
   assert( i < j );

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

Notes:
(1) The distribution uniformDiscrete(0, 1) is the same as Bernoulli (1/2).
(2) Uniform Discrete distribution is the discrete analog of the uniform distribution.

Examples of the probability density function and the cumulative distribution function are shown in Figures 71 and 72, respectively.

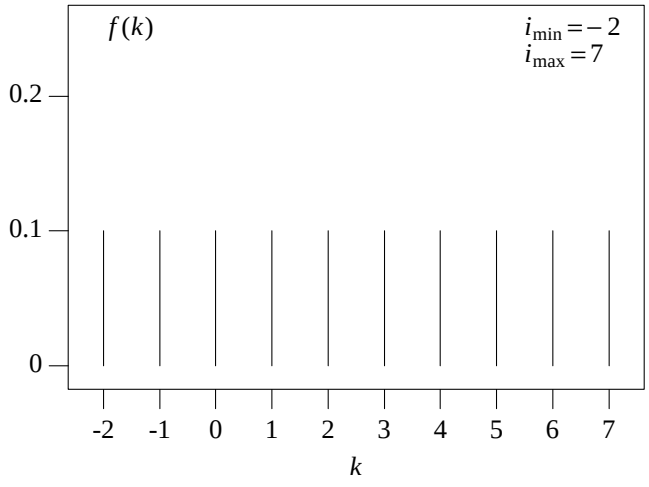


Figure 71. Uniform Discrete Density Function.

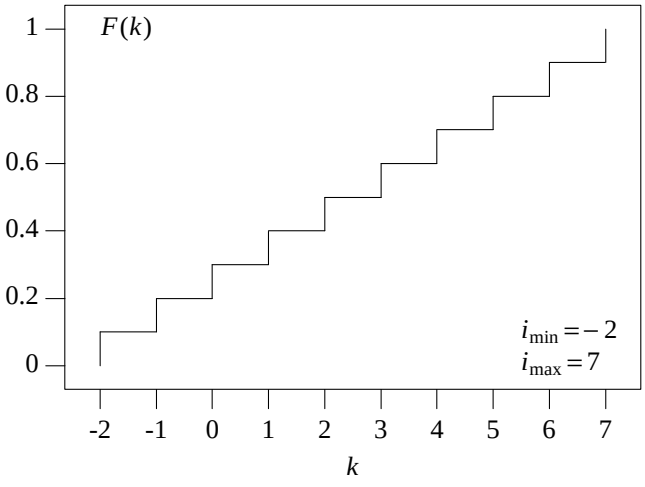


Figure 72. Uniform Discrete Distribution Function.

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) data pairs	interpolated data point x
Empirical Discrete	file of (k_i, p_i) data pairs	selection of a data point k
Sampling With and Without Replacement	file of k_i data	selection of a data point k
Stochastic Interpolation	file of 2-D data points (x_i, y_i)	new 2-D data point (x, y)

5.3.1 Empirical

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

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

where x_i, $i = 0, 1, \cdots n$ are the data points, and $F(x_i)$ is the cumulative probability at the point 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:

$$\begin{array}{ll} x_0 & F(x_0) \\ x_1 & F(x_1) \\ x_2 & F(x_2) \\ \cdots & \cdots \\ x_n & F(x_n)\,, \end{array}$$

where $F(x_0) = 0$, $F(x_n) = 1$, and $F(x_i) < F(x_{i+1})$. The data points are required be in *ascending order* but need not be equally spaced and the number of pairs is arbitrary.

Output: $x \in [x_0, x_n)$

Algorithm: This algorithm works by the inverse transform method.
(1) Generate $U \sim U(0, 1)$
(2) Locate index i such that $F(x_i) \le U < F(x_{i+1})$
(3) Return $X = x_i + \dfrac{U - F(x_i)}{F(x_{i+1}) - F(x_i)} (x_{i+1} - x_i)$

Source Code:

```
double empirical( void )
{
   static vector< double > x, cdf;
   static int              n;
   static bool             init = false;

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

      // check that this is indeed a cumulative distribution

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

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

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

5.3.2 Empirical Discrete

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

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

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

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

Output: $x \in \{k_1, k_2, \cdots, k_n\}$

Algorithm:

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

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

(3) Return $X = k_j$

Source Code:

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

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

      // form the cumulative distribution

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

      // check that the integer points are in ascending order

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

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

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

   double p = uniform( 0., max );

   // locate and return the corresponding index

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

Notes:

(1) The data must reside in a file named **empiricalDiscrete**.

(2) The number of data pairs in the file is arbitrary (and is not a required input as the code dynamically allocates the memory required).

As an example, consider the following hypothetical empirical data:

```
2    0.2
3    0.4
5    0.1
7    0.2
9    0.1
```

The probability density function and cumulative distribution function are shown in Figures 73 and 74, respectively.

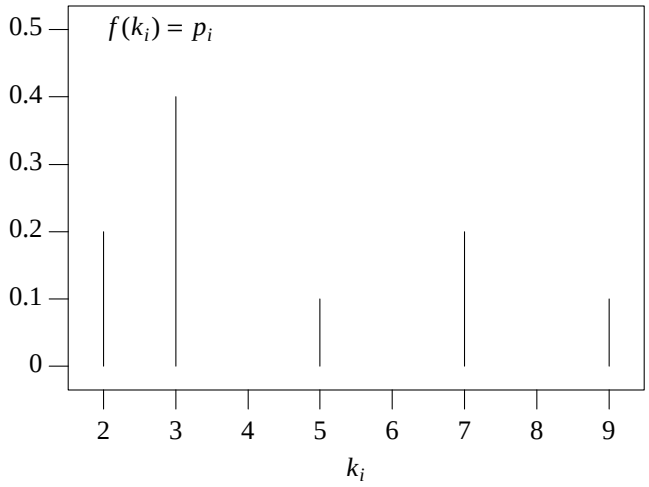


Figure 73. Discrete Empirical Density Function.

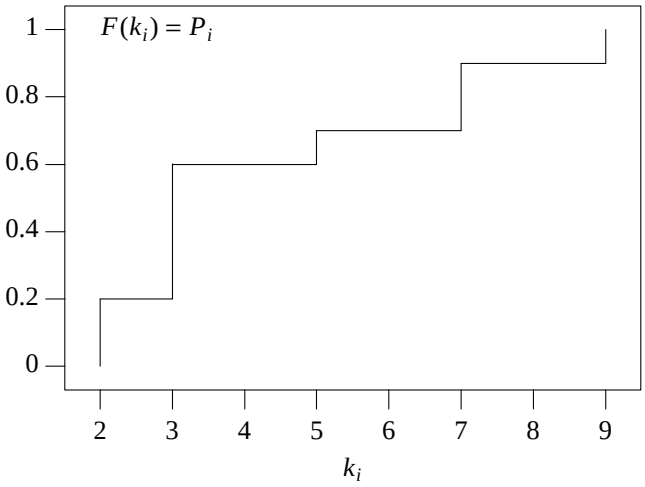


Figure 74. Discrete Empirical Distribution Function.

5.3.3 Sampling With and Without Replacement

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

$$P(k) = \binom{n}{k} \frac{K^k (N-K)^{n-k}}{N^n}, \quad \text{where} \quad \binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{48}$$

(See the binomial distribution in section 5.2.2.) Let the data be represented by $\{x_1, \cdots, x_N\}$. 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

$$P(k, n) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}, \quad \text{where} \quad \binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{49}$$

(See the hypergeometric distribution in section 5.2.4.) An algorithm for this case is as follows.

(1) Perform a random shuffle of the data points $\{x_1, \cdots, x_N\}$. (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 source code implements both methods—i.e., sampling with and without replacement.

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

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

         // shuffle contents of v once and for all
         // Ref: Knuth, D. E., The Art of Computer Programming, Vol. 2:
         //      Seminumerical Algorithms. London: Addison-Wesley, 1969.

         for ( int i = n - 1; i > 0; i-- ) {
            int j = int( ( i + 1 ) * _u() );
            swap( v[ i ], v[ j ] );
         }
      }
   }

   // return a random sample

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

5.3.4 Stochastic Interpolation

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

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

$$\mathbf{x} \Rightarrow (\mathbf{x} - \mathbf{x}_{\min})/|\mathbf{x}_{\max} - \mathbf{x}_{\min}|.$$

(2) Randomly select (with replacement) one of the n data points along with its nearest $m-1$ neighbors $\mathbf{x}_1, \cdots, \mathbf{x}_{m-1}$ 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 source code implements stochastic interpolation.

```
// comparison functor for use in determining the neighborhood of a data point

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

point stochasticInterpolation( void )

// Refs: Taylor, M. S. and J. R. Thompson, Computational Statistics & Data
//       Analysis, Vol. 4, pp. 93-101, 1986; Thompson, J. R., Empirical Model
//       Building, pp. 108-114, Wiley, 1989; Bodt, B. A. and M. S. Taylor,
//       A Data Based Random Number Generator for A Multivariate Distribution -
//       A User's Manual, ARBRL-TR-02439, BRL, APG, MD, Nov. 1982.
{
   static vector< point > data;
   static point           min, max;
   static int             m;
   static double          lower, upper;
   static bool            init = false;

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

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

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

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

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

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

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

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

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

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

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

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

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

   // make this point the origin of the coordinate system

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

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

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

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

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

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

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

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

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

   // restore the data to its original form

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

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

   p += mean;
   p += origin;

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

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

   return p;
}
```

Notes:
(1) Notice that with 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.
- **`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*
bivariateNormal	normal distribution in two dimensions	**`cartesianCoord`**
bivariateUniform	uniform distribution in two dimensions	**`cartesianCoord`**
corrNormal	normal distribution in two dimensions with correlation	**`cartesianCoord`**
corrUniform	uniform distributionin two dimensions with correlation	**`cartesianCoord`**
spherical	uniform distribution over the surface of the unit sphere	**`sphericalCoord`**
sphericalND	uniform distribution over the surface of the N-dimensional unit sphere	$(x_1, \cdots, x_N)$

5.4.1 Bivariate Normal (Bivariate Gaussian)

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:

```
cartesianCoord bivariateNormal( double muX, double sigmaX,
                                double muY, double sigmaY )
{
   assert( sigmaX > 0. && sigmaY > 0. );

   cartesianCoord p;
   p.x = normal( muX, sigmaX );
   p.y = normal( muY, sigmaY );
   return p;
}
```

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 Figures 75 and 76.

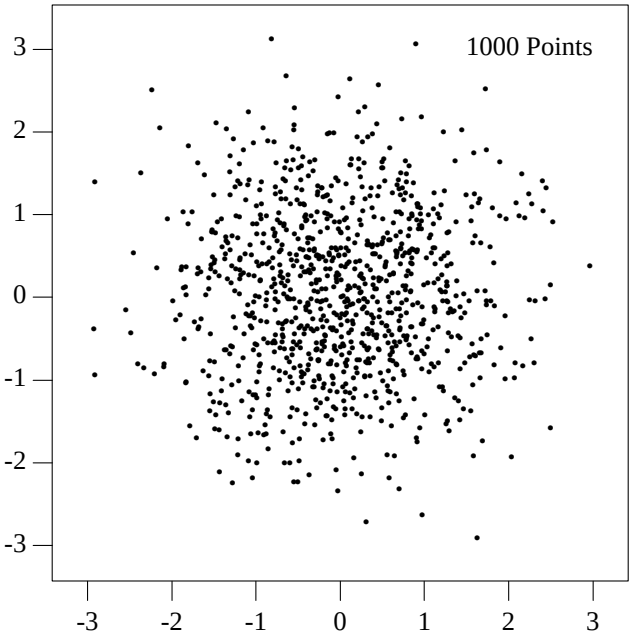


Figure 75. bivariateNormal(0., 1., 0., 1.).

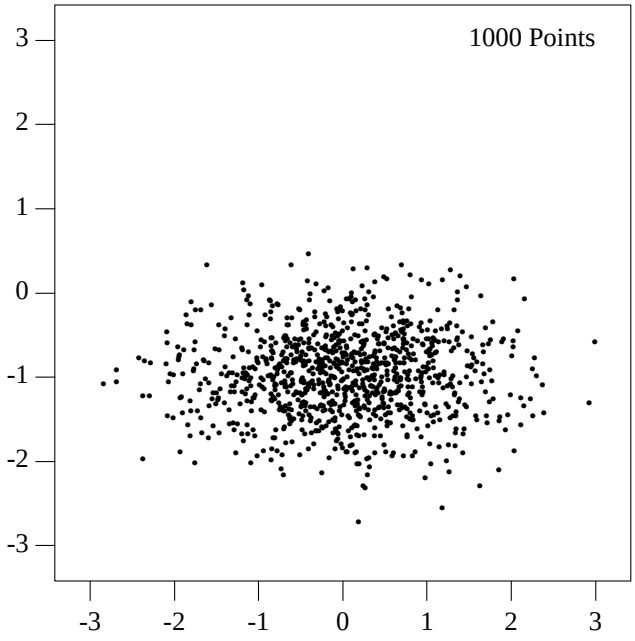


Figure 76. bivariateNormal(0., 1., -1, 0.5).

5.4.2 Bivariate Uniform

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

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

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

Algorithm:
(1) Independently generate $X \sim U(-1, 1)$ 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:
```
cartesianCoord bivariateUniform( double xMin, double xMax,
                                 double yMin, double yMax )
{
   assert( xMin < xMax && yMin < yMax );
   double x0 = 0.5 * ( xMin + xMax );
   double y0 = 0.5 * ( yMin + yMax );
   double a  = 0.5 * ( xMax - xMin );
   double b  = 0.5 * ( yMax - yMin );
   double x, y;

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

   cartesianCoord p;
   p.x = x0 + a * x;
   p.y = y0 + b * y;
   return p;
}
```

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 Figures 77 and 78.

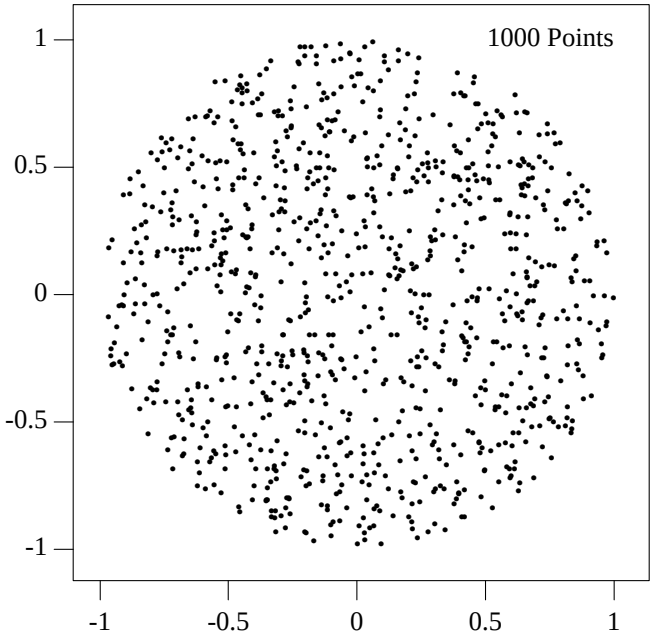


Figure 77. bivariateUniform(0., 1., 0., 1.).

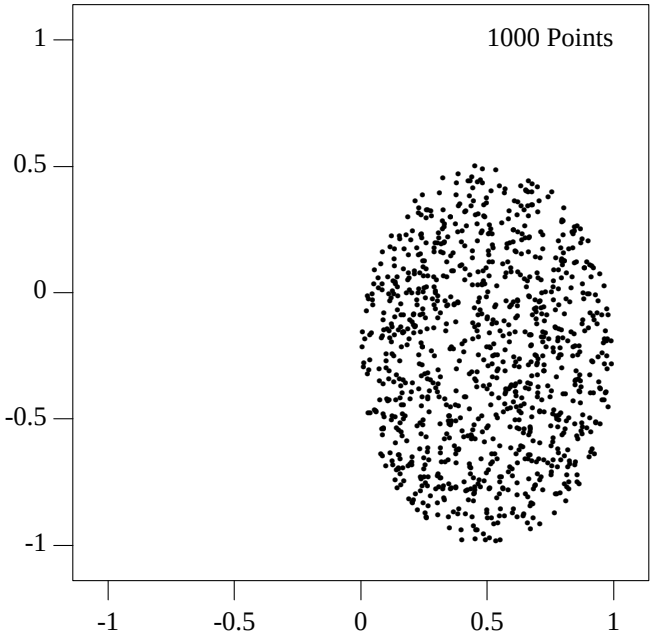


Figure 78. bivariateUniform(0., 1., -1., 0.5).

5.4.3 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; positive scale parameters (σ_x, σ_y); correlation coefficient, $-1 \le \rho \le 1$

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

Mode: (μ_x, μ_y)

Variance: (σ_x^2, σ_y^2)

Correlation Coefficient: ρ

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

Source Code:

```
cartesianCoord corrNormal( double r, double muX, double sigmaX,
                                     double muY, double sigmaY )
{
   assert( -1. <= r && r <= 1. );          // bounds on corr coeff
   assert( sigmaX > 0. && sigmaY > 0. );   // positive std dev

   double x = normal();
   double y = normal();

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

   cartesianCoord p;
   p.x = muX + sigmaX * x;                 // translate and scale
   p.y = muY + sigmaY * y;
   return p;
}
```

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

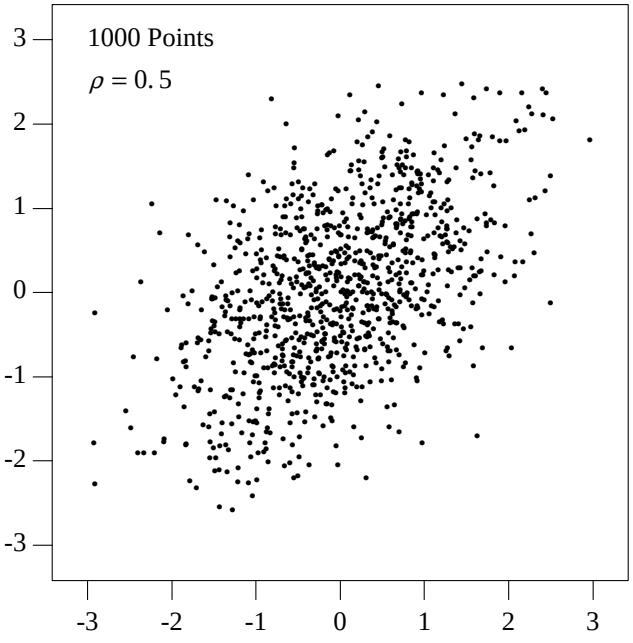


Figure 79. corrNormal(0.5, 0., 1., 0., 1.).

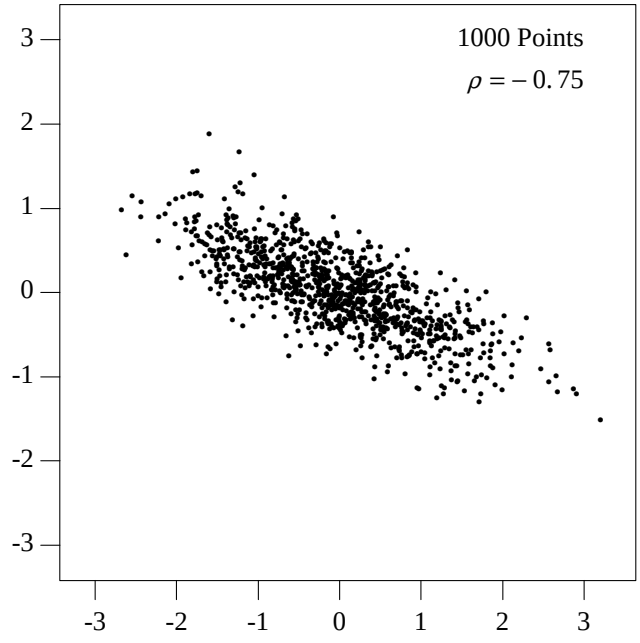


Figure 80. corrNormal(-0.75, 0., 1., 0., 0.5).

5.4.4 Correlated Uniform

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

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

Algorithm:
(1) Independently generate $X \sim U(-1, 1)$ 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 $Y = \rho X + \sqrt{1 - \rho^2}\, Z$
(4) Return $(x_0 + aX, y_0 + bY)$

Source Code:

```
cartesianCoord corrUniform( double r, double xMin, double xMax,
                                      double yMin, double yMax )
{
   assert( -1. <= r && r <= 1. );          // bounds on corr coeff
   assert( xMin < xMax && yMin < yMax );
   double x0 = 0.5 * ( xMin + xMax );
   double y0 = 0.5 * ( yMin + yMax );
   double a  = 0.5 * ( xMax - xMin );
   double b  = 0.5 * ( yMax - yMin );
   double x, y;

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

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

   cartesianCoord p;
   p.x = x0 + a * x;                     // translate & scale
   p.y = y0 + b * y;
   return p;
}
```

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

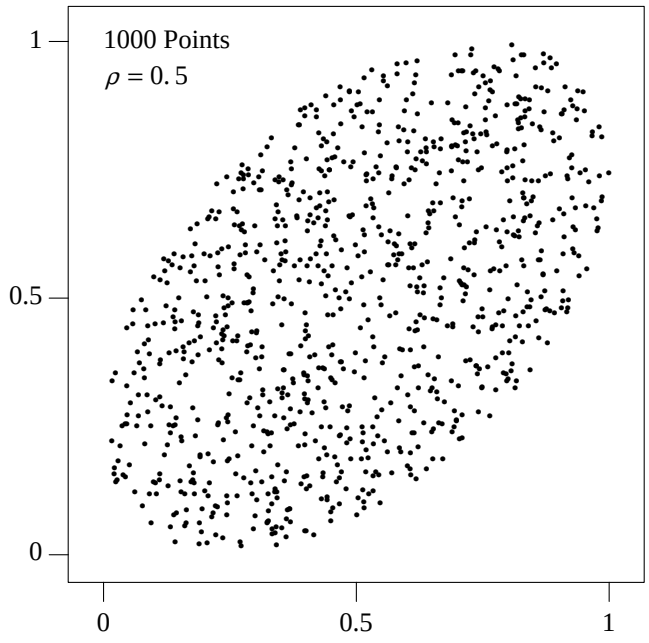


Figure 81. corrUniform(0.5, 0., 1., 0., 1.).

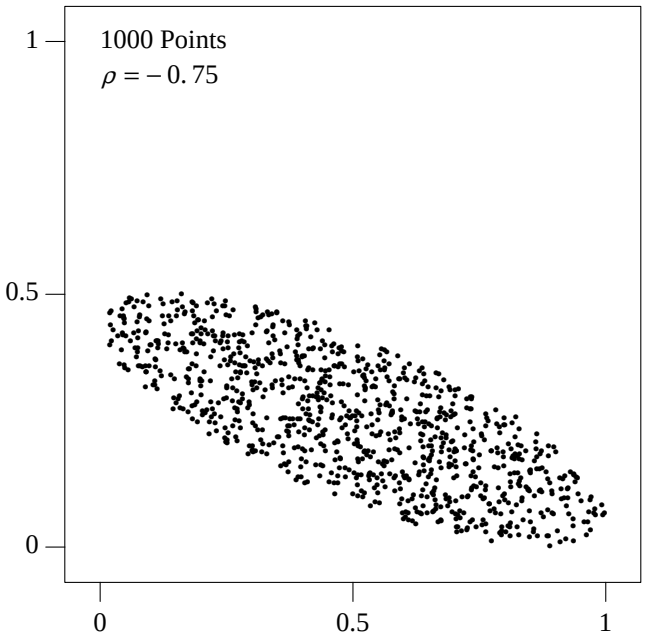


Figure 82. corrUniform(-0.75, 0., 1., 0., 0.5).

5.4.5 Spherical Uniform

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

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

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

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

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

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

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

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

Source Code:

```
sphericalCoord spherical( double thMin, double thMax,
                          double phMin, double phMax )
{
   assert( 0. <= thMin && thMin < thMax && thMax <= M_PI &&
           0. <= phMin && phMin < phMax && phMax <= 2. * M_PI );

   sphericalCoord p;
   p.polar   = acos( uniform( cos( thMax ), cos( thMin ) ) );
   p.azimuth = uniform( phMin, phMax );
   return p;
}
```

Figure 83 shows the uniform random distribution of 1,000 points on the surface of the unit sphere obtained via calls to this function.

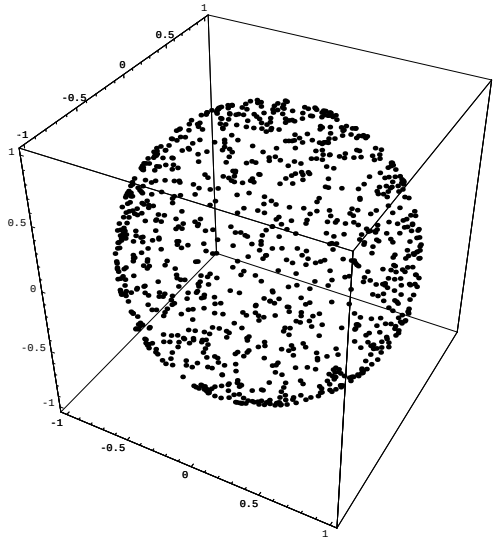


Figure 83. Uniform Spherical Distribution via spherical().

5.4.6 Spherical Uniform in N-Dimensions

This will generate uniformly distributed points on the surface of the unit sphere in n dimensions. Whereas the previous distribution (5.4.5) is designed to return the location 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 **X** to receive values; number of dimensions n

Output: Vector **X** of unit length (i.e., $X_1^2 + \cdots + X_n^2 = 1$)

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

Source Code:

```
void sphericalND( double x[],   // x array returns point
                  int    n )    // n is number of dimensions
{
// generate a point inside the unit n-sphere by normal polar method

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

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

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

Notes:
(1) When $n = 1$, this returns $\{-1, +1\}$.
(2) When $n = 2$, it generates coordinates of points on the unit circle.
(3) When $n = 3$, it generates coordinates of 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:*

(1) Tausworthe or Shift Register Generation of Random Bits, and

(2) Maximal Avoidance or Sub-Random Sequences.

5.5.1 Tausworthe Random Bit Generator

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

$$P_n(x) = (x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + 1) \bmod 2, \tag{50}$$

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

$$a_n = (x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + 1) \bmod 2 \tag{51}$$

will generate a string of 1's and 0's that has a maximal cycle length of $2^n - 1$ (i.e., all possible values excluding the case of all zeroes). Primitive polynomials of order n from 1 to 100 have been tabluated (Watson 1962).

Since the truth table of integer addition modulo 2 is the same as ''exclusive or,'' 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 4,294,967,295.

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

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

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

Source Code:

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

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

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

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

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

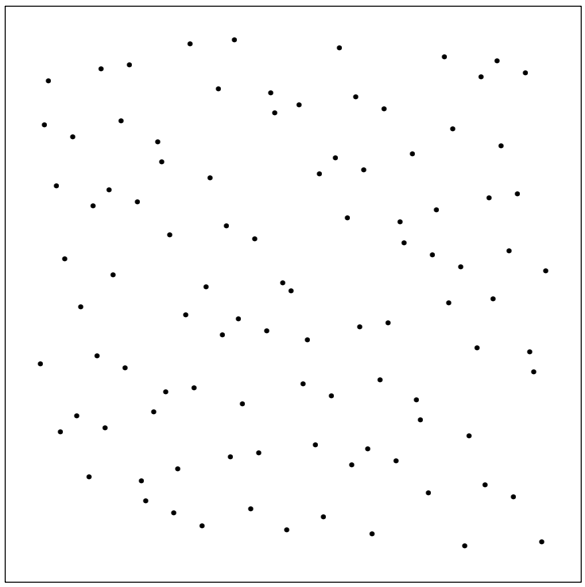
Notes: (1) The constants used in the above source code are defined in **`Random.h`**.
(2) This generator is 3.6 times faster than **`bernoulli( 0.5 )`**.

---

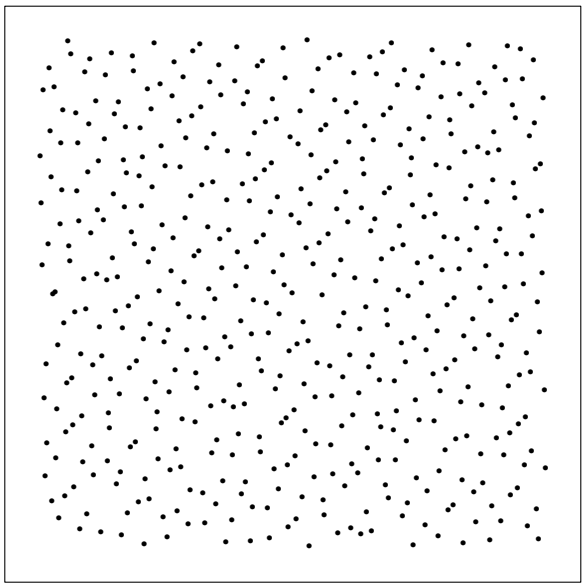
* The theory underlying these techniques is quite involved, but Press et al. (1992) and sources cited therein provide a starting point.

5.5.2 Maximal Avoidance (Quasi-Random)

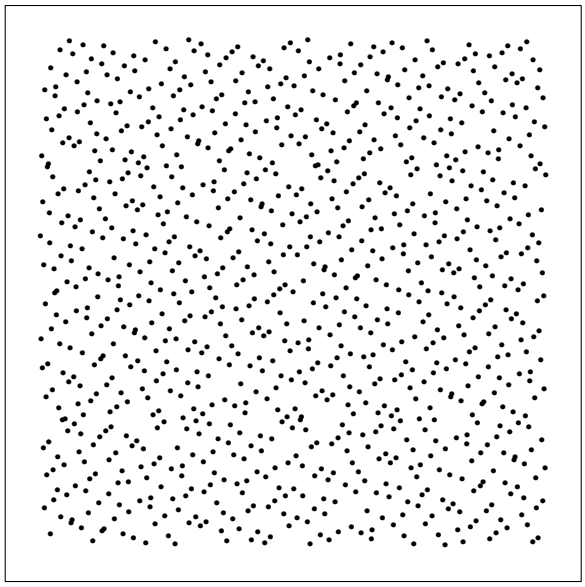
Maximal avoidance is a technique for generating points in a multidimensional space that are simultaneously self-avoiding, while appearing to be random. For example, the first three plots in Figure 84 show points generated with this technique to demonstrate how they tend to avoid one another. The last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



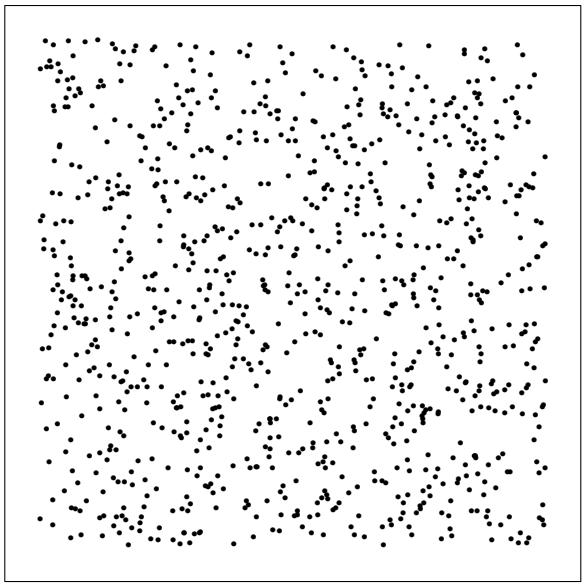
100 Maximal Avoidance Data Points



500 Maximal Avoidance Data Points



1000 Maximal Avoidance Data Points



1000 Uniformly Distributed Data Points

Figure 84. Maximal Avoidance Compared to Uniformly Distributed.

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

```
double avoidance( void )   // 1-dimension (overloaded for convenience)
{
   double x[ 1 ];
   avoidance( x, 1 );
   return x[ 0 ];
}
void avoidance( double x[], int ndim )   // multi-dimensional
{
   static const int MAXBIT = 30;
   static const int MAXDIM = 6;

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

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

6. DISCUSSION AND EXAMPLES

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

6.1 Making Sense of the Discrete Distributions

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

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

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

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

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

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

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

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

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

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

6.2 Adding New Distributions

We show here how it is possible to extend the list of distributions. Suppose that we want to generate random numbers according to the probability density function shown in Figure 85.

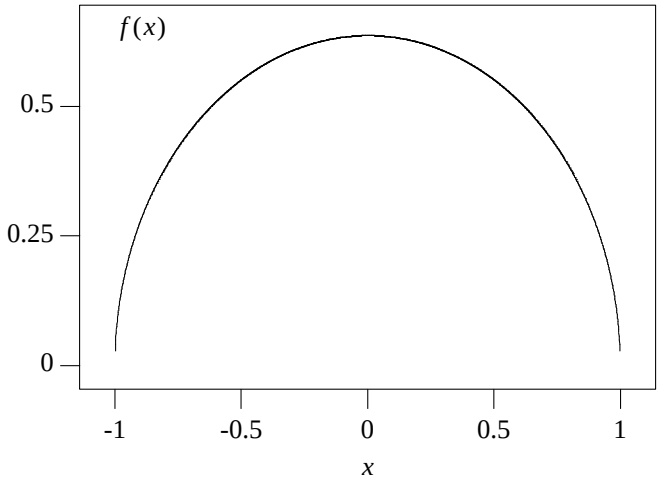


Figure 85. Semi-Elliptical Density Function.

The figure is that of a semi-ellipse, and its equation is

$$f(x) = \frac{2}{\pi}\sqrt{1-x^2}, \quad \text{where} \quad -1 \le x \le 1. \tag{52}$$

Integrating, we find that the cumulative distribution function is

$$F(x) = \frac{1}{2} + \frac{x\sqrt{1-x^2} + \sin^{-1}(x)}{\pi}, \quad \text{where} \quad -1 \le x \le 1. \tag{53}$$

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

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

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

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

```
double ellipse( double x, double, double )   // Ellipse Function
{
   return sqrt( 1. - x * x ) / M_PI_2;
}

double Random::elliptical( void )   // Elliptical Distribution
{
   const double X_MIN = -1.;
   const double X_MAX = 1.;
   const double Y_MIN = 0.;
   const double Y_MAX = 1. / M_PI_2;

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

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

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

void main( void )
{
   Random rv;
   for ( int i = 0; i < 1000; i++ ) cout << rv.elliptical() << endl;
}
```

6.3 Bootstrap Method as an Application of Sampling

If we are only interested in the mean value, $\bar{x}$, of a set of data and wish to know the accuracy of the sample mean, then there is a handy formula available:

$$\text{Standard Error of the Mean} = \left[\frac{1}{N(N-1)} \sum_{i=1}^{N} (x_i - \bar{x})^2 \right]^{1/2} . \tag{54}$$

On the other hand, if we are interested in some other metric, such as the correlation coefficient, then there is no simple analytical formula that allows us to estimate the error. The bootstrap method was designed to address this situation. The basic idea is that we sample the original data with replacement to obtain a synthetic data set of another N data points, and, from this, we compute the statistic we are interested in. We then repeat this process over and over until we have built up a set M of computed values of the relevant statistic. We then compute the standard deviation of these M values; it will provide the standard error of the statistic. Given the high cost and consequent scarcity of data in many applications, combined with the reduced cost and abundance of computing power, the bootstrap method becomes a very attractive technique for extracting information from empirical data (Diaconis and Efron 1983; Efron and Tibshirani 1991). The *New York Times* had this to say:

> A new technique that involves powerful computer calculations is greatly enhancing the statistical analysis of problems in virtually all fields of science. The method, which is now surging into practical use after a decade of refinement, allows statisticians to determine more accurately the reliability of data analysis in subjects ranging from politics to medicine to particle physics.... (Nov. 8, 1988, C1, C6).

Here, we give an example of how sampling may be applied to empirical data in order to compute a bootstrap error estimate. The data consist of 150 spall fragments that were collected when a penetrator perforated a plate of armor. Each spall fragment was weighed and the dimensionless shape factor (see section 3.7 for the definition) was measured from 16 different directions in order to compute an average shape factor. Thus, the experimental data consist of 150 mass, average shape factor pairs. The question arises as to whether there is any correlation between mass and average shape factor. For example, one might expect small fragments to be more compact and large fragments to be more irregular in shape. This would be reflected in a positive correlation coefficient. The correlation coefficient computed from the original experimental data is -0.132874. Since the absolute value is considerably smaller than 1, there appears to be no correlation between mass of the fragment and its average shape factor.* Now, we would like to know how much variation to expect in the correlation coefficient. The 150 data pairs were put into a file called "sampleData." The following source code then implements the bootstrap method.

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

void main( void )
{
   const int N_DATA = 150;   // number of data points
   const int N_DIMS = 2;     // number of dimensions
   Random rv;

   double data[ N_DIMS ];
   for ( int i = 0; i < N_DATA; i++ ) {
      rv.sample( data, N_DIMS );
      cout << data[ 0 ] << " " << data[ 1 ] << endl;
   }
}
```

This will generate a synthetic set of 150 data pairs, from which we compute the corresponding correlation coefficient. We then replicate the process 128, 256, 512, and 1,024 times. After 128 replications, we find the following statistics on the *state of correlation* among the two variables, shape factor and mass.

---

* More formally, the value of the t statistic is -1.63094, and since $|-1.63094| < 1.96$, the critical value for a two-sided test at a 0.05 significance level, it fails the Student t test. Consequently, we cannot reject the null hypothesis that the data pairs are uncorrelated.

```
n       =       128
min     =       -0.251699
max     =       -0.0775172
sum     =       -17.8033
ss      =       2.60043
mean    =       -0.139088
var     =       0.000978001
sd      =       0.031273
se      =       0.00276417
skew    =       -0.83868
kurt    =       4.44186
```

The statistics after 256 replications are as follows.

```
n       =       256
min     =       -0.247179
max     =       0.00560571
sum     =       -36.5577
ss      =       5.51328
mean    =       -0.142803
var     =       0.00114789
sd      =       0.0338805
se      =       0.00211753
skew    =       0.254268
kurt    =       4.59266
```

The statistics after 512 replications are as follows.

```
n       =       512
min     =       -0.247179
max     =       0.00560571
sum     =       -72.0359
ss      =       10.7341
mean    =       -0.140695
var     =       0.00117227
sd      =       0.0342384
se      =       0.00151314
skew    =       0.161558
kurt    =       3.91064
```

The statistics after 1,024 replications are as follows.

```
n       =       1024
min     =       -0.280715
max     =       0.00560571
sum     =       -142.313
ss      =       21.0328
mean    =       -0.138978
var     =       0.00122616
sd      =       0.0350165
se      =       0.00109427
skew    =       0.118935
kurt    =       4.05959
```

Thus, we may say (with 1–σ confidence) that the correlation coeffient is -0.139 ± 0.035 and conclude that the data are uncorrelated.

Performing a bootstrap on the t statistic gives the following results.

N	value of t statistic
128	-1.72921 ± 0.401098
256	-1.70181 ± 0.407198
512	-1.70739 ± 0.445316
1024	-1.68787 ± 0.428666

6.4 Monte Carlo Sampling to Evaluate an Integral

A simple application of random number distributions is in the evaluation of integrals. Integration of a function of a single variable, $f(x)$, is equivalent to evaluating the area that lies below the function. Thus, a simple way to estimate the integral

$$\int_a^b f(x)\,dx \tag{55}$$

is to first find the bounding rectangle $[a,b]\times[0,y_{\max}]$, as shown in Figure 86; select uniform random points (X,Y) within the rectangle; and, for every point that satisfies the condition $Y < f(X)$, increment the area estimate by the amount $(b-a)y_{\max}/N$, where N is the total number of sample points in the bounding rectangle.

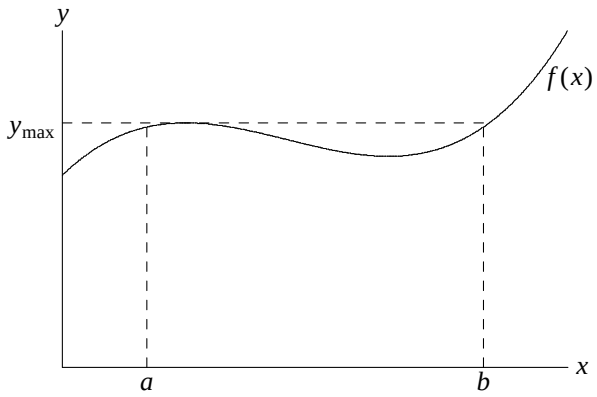


Figure 86. Integration as an Area Evaluation via Acceptance-Rejection Algorithm

This can be accomplished with the following source code.

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

double f( double x ) { return ··· }

void main( void )
{
   Random rv;

   const double A     = ···
   const double B     = ···
   const double Y_MAX = ···
   const int    N     = ···

   double area = 0.;
   for ( int i = 0; i < N; i++ ) {
      double x = rv.uniform( A, B );
      double y = rv.uniform( 0., Y_MAX );
      if ( y < f( x ) ) area += ( B - A ) * Y_MAX / N;
   }
   cout << "area estimate = " << area << endl;
}
```

This is essentially a binomial process with a probability of success p equal to the ratio of the area under the curve to the area of the bounding rectangle. The standard deviation of the area estimate, therefore (see section 5.2.2), is

$$\sigma = \frac{\sqrt{Np(1-p)}}{N} \times \text{area of bounding rectangle.} \tag{56}$$

Note that the factor $\sqrt{p(1-p)}$ is close to 0.5 unless we happen to have a very close-fitting bounding rectangle. This so-called ‘‘hit-or-miss’’ method is also inefficient in that two calls are made to the random number generator for each sample point. A more efficient method is obtained by first approximating the integral in eq. (55) by its Riemann sum:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} f(x_i)\,\Delta x_i = (b-a)\,\frac{1}{N}\sum_{i=1}^{N} f(x_i). \tag{57}$$

This dispenses with the bounding rectangle and only requires uniform random points along the x-axis. Consequently, the source code can be simplified to the following.

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

double f( double x ) { return ··· }

void main( void )
{
   Random rv;

   const double A = ···
   const double B = ···
   const int    N = ···

   double sum = 0.;
   for ( int i = 0; i < N; i++ ) sum += f( rv.uniform( A, B ) );
   cout << "area estimate = " << ( B - A ) * sum / N << endl;
}
```

Notice that eq. (57) expresses the integral as $(b-a)\times$mean value of f. Thus, we increase accuracy to the extent that the points are spread uniformly over the x-axis. Maximal avoidance is perfectly suited to do this and avoid the clustering of points that we get with the uniform random generator. This can be accomplished by simply replacing

```
rv.uniform( A, B )   ⇒   rv.avoidance() * ( B - A )
```

in the above source code.

To illustrate the advantage of maximal avoidance over uniform random in a simple case, let us consider the cosine density:

$$f(x) = \frac{1}{2b}\cos\left(\frac{x-a}{b}\right), \quad 0 \le x \le 1, \tag{58}$$

where $a = 1/2$ and $b = 1/\pi$. The integral can be performed analytically:

$$F(x) = \int_0^x f(\xi)\,d\xi = \frac{1}{2}\left[1+\sin\left(\frac{x-a}{b}\right)\right], \quad 0 \le x \le 1, \tag{59}$$

and $F(1) = 1$. Table 5 shows the errors with the two methods.

Table 5. Comparison of Uniform Random and Maximal Avoidance in Monte Carlo Sampling

Number of	*Uniform Random*		*Maximal Avoidance*	
Sample Points	*Value*	*% Error*	*Value*	*% Error*
100	1.00928	+0.93	1.01231	+1.23
1,000	0.993817	–0. 6183	1.0005	+0.05
10,000	1.00057	+0.057	1.00015	+0.015
100,000	0.999771	–0. 0229	1.00001	+0.001
1,000,000	1.00026	+0.026	1	$< 10^{-5}$

It can be shown (e.g., Press et al. [1992] pp. 309–314) that the fractional error term in maximal avoidance decreases as $\ln N/N$, which is almost as fast as $1/N$. In contrast, the fractional error term in uniform random sampling decreases as $N^{-1/2}$, the same as in the hit-or-miss Monte Carlo sampling (cf. eq. [56]).

6.5 Application of Stochastic Interpolation

The technique of stochastic interpolation is very useful in those cases where the data does not seem to fit any known distribution. It allows us to simulate the essential characteristics of the data without returning the same data points over and over again. For example, Figure 87 shows bifurcated data in the x–y plane. Without an understanding of the underlying mechanism, it would be very difficult to fit a distribution to this data. However, it is easy to create synthetic realizations using stochastic interpolation. We must first place the data in a file named ‘‘stochasticData.’’ Then the following code will produce another realization such as that shown in Figure 88.

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

void main( int argc, char* argv[] )
{
   long seed = long( getpid() );
   if ( argc == 2 ) seed = atoi( argv[ 1 ] );

   Random rv( seed );

   for ( int i = 0; i < N_DATA; i++ ) {
      point p = rv.stochasticInterpolation();
      cout << p.x << " " << p.y << endl;
   }
}
```

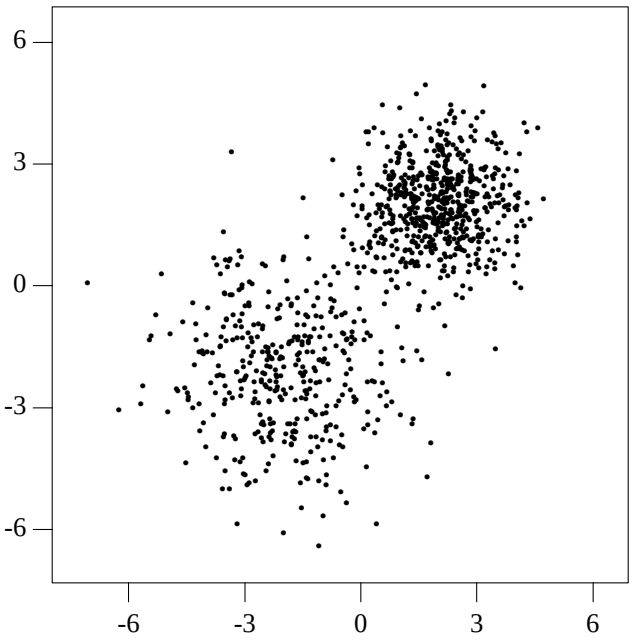


Figure 87. Stochastic Data for Stochastic Interpolation.

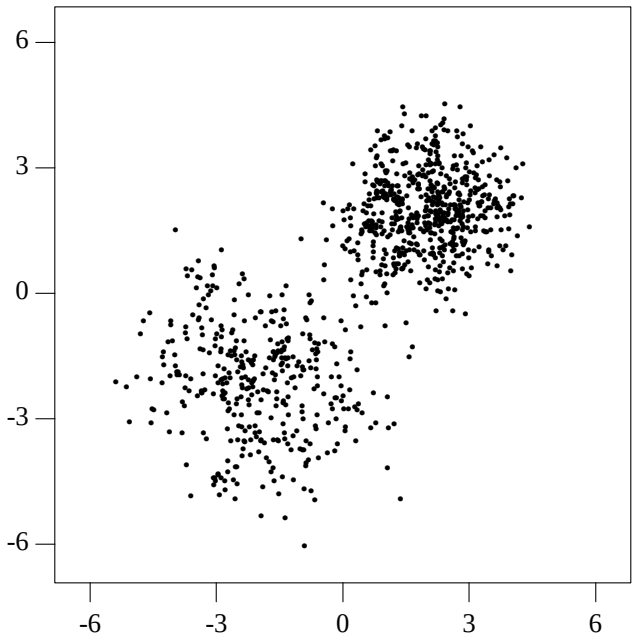


Figure 88. Synthetic Data via Stochastic Interpolation.

6.6 Combining Maximal Avoidance With Distributions

It is also possible to combine techniques. For example, we could use maximal avoidance to generate points in space and then perform a transformation to get the density of points appropriate for a desired distribution. This amounts to using maximal avoidance rather than uniform random to generate the points for transformation. However, since maximal avoidance is deterministic rather than pseudo-random, the price we pay is that the pattern generated will always be the same. Figure 89 is an example to generate bivariate normal.

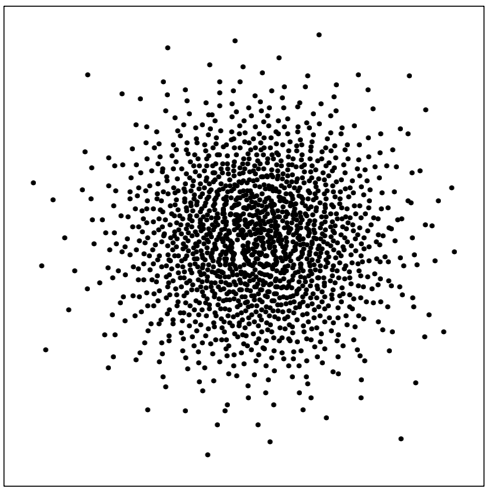


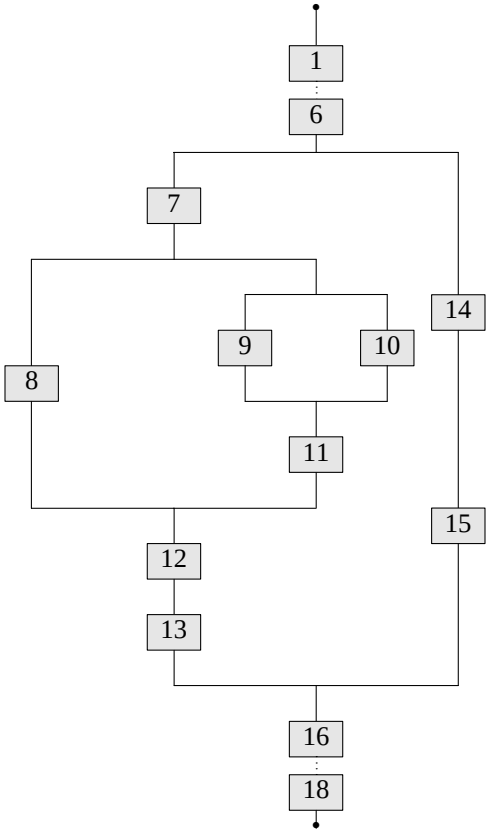
Figure 89. Combining Maximal Avoidance With Bivariate Normal.

The source code is as follows.

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

void main( void )
{
   Random rv;
   const int N = 2000;   // number of points

   const double MU    = 0.;
   const double SIGMA = 1.;
   const double X_MIN = -1.;
   const double X_MAX = 1.;
   const double Y_MIN = -1.;
   const double Y_MAX = 1.;

   double data[ 2 ];

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

      rv.avoidance( data, 2 );

      double x = X_MIN + ( X_MAX - X_MIN ) * data[ 0 ];
      double y = Y_MIN + ( Y_MAX - Y_MIN ) * data[ 1 ];
      double p = x * x + y * y;
      if ( p < 1. ) {
         cout << MU + SIGMA * x * sqrt( -2. * log( p ) / p ) << "      "
              << MU + SIGMA * y * sqrt( -2. * log( p ) / p ) << endl;
      }
   }
}
```

6.7 Application of Tausworthe Random Bit Vector

Various systems in a combat vehicle are composed of critical components that are functionally related through the use of one or more *fault trees*. For instance, Figure 90 shows the fault tree for the main gun of the M1A1 tank (Ploskonka et al. 1988).



ID	*Description*	*ID*	*Description*
1	Main Gun Tube	10	Gunner's Control Handle
2	Main Gun Breech	11	Cable 1W200-9
3	Recoil Mechanism	12	Cable 1W104
4	Recoil Replenisher and Hose	13	Gunner's Primary Sight - Lower Panel
5	Main Gun Trunnions	14	Blasting Machine
6	Turret Networks Box	15	Cable 1W105-9
7	Electric Power - Turret	16	Cable 1W107-9
8	Manual Elevation Pump Handle	17	Cable 1W108-9
9	Commander's Control Handle	18	Main Gun Safety Switch

Figure 90. Fault Tree for Main Armament of M1A1 Tank.

Each of the 18 components comprising this diagram is considered critical because its dysfunction may have an adverse affect upon the gun functioning. However, as long as there is at least one continuous path of functioning components from the top node to the bottom node, the main gun will still function. It is clear, for example, that the fault tree as a whole is more sensitive to the loss of component 1 than it is to the loss of component 8. There are other cases where it is not so clear. Here, we show how the random bit vector can be used to rank the sensitivity of the components based upon the functioning of this fault tree. We need a bit vector of length $n = 18$ and, in order to generate all the possible combinations of states, we need to take $2^{18} - 1 = 262,143$ samples, the cycle length. We are guaranteed that each combination will occur once and only once in the cycle (although in random order). The following code will print out the state of each of the 18 components along with the state of the fault tree.

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

void main( void )
{
   const unsigned LEN  = 18;                          // number of components
   const int      N    = int( pow( 2, LEN ) - 1 );    // number of combinations
   unsigned       seed = 123456789;                   // seed for tausworthe generator
   bool           c[ LEN ];                           // boolean component array
   Random         rv;

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

      rv.tausworthe( seed, LEN, c );   // assign a state to each component

      for ( int i = 0; i < LEN; i++ ) cout << c[ i ] << " ";
      c[ 0 ]  |= c[ 1 ] | c[ 2 ] | c[ 3 ] | c[ 4 ] | c[ 5 ];
      c[ 8 ]  &= c[ 9 ];
      c[ 8 ]  |= c[ 10 ];
      c[ 7 ]  &= c[ 8 ];
      c[ 6 ]  |= c[ 7 ] | c[ 11 ] + c[ 12 ];
      c[ 13 ] |= c[ 14 ];
      c[ 6 ]  &= c[ 13 ];
      c[ 0 ]  |= c[ 6 ] | c[ 15 ] | c[ 16 ] | c[ 17 ];
      cout << c[ 0 ] << endl;
   }
}
```

Next, we determine the correlation coefficient between each of the components and the overall state of the fault tree. The results are shown in Table 6.

Table 6. Correlation Coefficient Between Component and Fault Tree Deactivation

Component	*Correlation Coefficient*
1	0.024713
2	0.024713
3	0.024713
4	0.024713
5	0.024713
6	0.024713
7	0.00494267
8	0.00216247
9	0.000309005
10	0.000309005
11	0.00123574
12	0.00494267
13	0.00494267
14	0.0179169
15	0.0179169
16	0.024713
17	0.024713
18	0.024713

It is apparent that the components fall into groups based upon their significance to the overall fault tree. Sorting the components from most significant to least significant gives the results shown in Table 7.

Table 7. Ranking of Component Significance to Fault Tree Deactivation

Group	*Components*
1	1, 2, 3, 4, 5, 6, 16, 17, and 18
2	14 and 15
3	7, 12, and 13
4	8
5	11
6	9 and 10

This shows that components fall into six distinct classes based upon their significance to the overall fault tree. It also gives a certain degree of verification that the coding is correct since it tell us that all the components in a given group occur in the fault tree with the same footing. It is clear by examing Figure 90 that components 7, 12, and 13, for instance, all have the same significance to the vulnerability of the main gun.

Now, for the case shown here, where the number of components is 18, we could easily write a program consisting of a series of nested loops to enumerate all $2^{18} - 1$ nonzero states. However, we do not have to add very many more components before this direct enumeation approach will not be feasible. For instance, 30 components results in more than one billion possible states. The Tausworthe random bit generator provides an alternative approach. We could replicate, say 100,000 times, knowing that the bit vectors will be unique as well as ‘‘random.’’ As such, it provides a useful tool for the examination and verification of fault trees.

7. REFERENCES

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.

Diaconis, P., and B. Efron. ‘‘Computer-Intensive Methods in Statistics.’’ *Scientific American*, pp. 116–130, May, 1983.

Efron, B., and R. Tibshirani. ‘‘Statistical Data Analysis in the Computer Age.’’ *Science*, pp. 390–395, 26 July 1991.

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

Hastings, N. A. J., and J. B. Peacock. *Statistical Distributions: A Handbook for Students and Practitioners*. New York: John Wiley & Sons, 1975.

Helicon Publishing. *The Hutchinson Encyclopedia*. http://www.helicon.co.uk, 1999.

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

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

New York Times. New York, C1, C6, 8 November 1988.

Pitman, J. *Probability*. New York: Springer-Verlag, 1993.

Ploskonka, J. J., T. M. Muehl, and C. J. Dively. ‘‘Criticality Analysis of the M1A1 Tank.’’ BRL-MR-3671, U.S. Army Ballistic Research Laboratory, Aberdeen Proving Ground, MD, June 1988.

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.

Ripley, B. D. *Stochastic Simulation*. New York: John Wiley & Sons, 1987.

Sachs, L. *Applied Statistics: A Handbook of Techniques*. New York: Springer-Verlag, Second Edition, 1984.

Spiegel, M. R. *Probability and Statistics*. New York: McGraw-Hill, 1975.

Tausworthe, R. C. ‘‘Random Numbers Generated by Linear Recurrence Modulo Two.’’ *Mathematics of Computation*. Vol. 19, pp. 201–209, 1965.

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.

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

Watson, E. J. ‘‘Primitive Polynomials (Mod 2).’’ *Mathematics of Computation*. Vol. 16, pp. 368–369, 1962.

INTENTIONALLY LEFT BLANK.

APPENDIX A:
UNIFORM RANDOM NUMBER GENERATOR

The underlying random number generator used is one that will generate uniformly distributed random numbers on the half-open interval $[0, 1)$. Any other distribution of random numbers, with few exceptions, results from mathematical transformation. Thus, it is clear that we need a good generator of random numbers $U(0, 1)$. Here we discuss some criteria of what constitutes a good generator. After this, we discuss some tests that can be applied and show the test results when applied to several candidate generators.

A.1 Selection Criteria

We will consider four attributes of what constitutes a good random number generator.

- *Range*
  A good generator should have the capability of producing a large range of random numbers.
- *Speed*
  A good generator should be fast.
- *Portability*
  The generator should give the same sequence of random numbers, regardless of what computer it is run on. As a minimum, we need the explicit source code.
- *Validation*
  A good generator should exhibit apparent randomness by passing certain well-accepted validation tests.

Six candidate generators were examined and tested for consideration as the underlying generator for $U(0, 1)$. They are all *linear conguential generators* of the form

$$X_{i+1} = (aX_i + c) (\mathrm{mod}\, m). \tag{A-1}$$

This is a recurrence relation for generating the next number in the sequence, given the multiplier a, the increment c, and the modulus m. The linear conguential method is a well-accepted technique for generating a sequence of numbers that take on the appearance of randomness. They possess the following properties.[1,2]

- They are relatively fast, requiring few arithmetic operations.
- They produce a *range* of values no greater than the modulus m.
- They have a *period* or *cycle length* no greater than m.
- However, they are not free of sequential correlations.

We consider two generators available in standard C libraries, **`rand`** and **`drand48`**—and thus, commonly used—and four versions of a generator proposed by Park and Miller.[2,3] First, we give a short description of each one and list the source code.

`rand`

This is the ANSI C version of a random number generator and has been implemented in C as follows:

```
unsigned long next = 1;

void srand( unsigned int seed )   /* set the seed */
{
   next = seed;
}

int rand( void )   /* return a pseudo-random number */
{
   next = next * 1103515245 + 12345;
   return ( unsigned int )( next / 65536 ) % 32768;
}
```

---

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

2 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.

3 Park, S. K., and K. W. Miller. *Communications of the ACM*. Vol. 31, pp. 1192–1201, 1988.

Now, it may appear that the modulus here is 32,768, but, in fact, it is 2^{32}, due to the fact that ‘‘`unsigned long`’’ is 32 bits in length and so modulus 2^{32} is implicit. However, due to the fact that the returned number is ‘‘`% 32768`’’ or ‘‘modulo 32768’’ means that it is only capable of producing, at most, 32,768 distinct numbers. Thus, it has a period of 2^{32} and a range no greater than 32,768.

`drand48`

This also is in the standard C library. This generator uses three 16-bit integer words in order to perform 48-bit arithmetic. The constants are $a = 25,214,903,917$, $c = 11$, and $m = 2^{48}$, so that it thas a very long period.

```
// drand48.C: A portable implementation of drand48, this routine will
//            generate precisely the same stream of pseudo-random numbers
//            in the interval [0,1) as drand48, for the same seed value.
//            The drand48 algorithm is based on the linear congruential
//               x[ n + 1 ] = ( a * x[ n ] + c ) ( mod m ),
//            where
//               a = 25214903917 (0x5DEECE66D),
//               c = 11 (0xB), and
//               m = 2^48 = 281474976710656 (0x1000000000000),
//            while using 48-bit integer arithmetic, but ignoring
//            multiplication and addition overflows of two 16-bit integers.

static const unsigned int N_BITS = 16;
static const double       TWO_16 = 1. / ( 1L << N_BITS );

static const unsigned int MASK = unsigned( 1 << ( N_BITS - 1 ) ) +
                                 unsigned( 1 << ( N_BITS - 1 ) ) - 1;   // 65535

static const unsigned int X0   = 0x330E;   // 13070
static const unsigned int X1   = 0xABCD;   // 43981
static const unsigned int X2   = 0x1234;   // 4660
static const unsigned int A0   = 0xE66D;   // 58989
static const unsigned int A1   = 0xDEEC;   // 57068
static const unsigned int A2   = 0x5;      // 5
static const unsigned int C    = 0xB;      // 11

static unsigned int x[ 3 ] = { X0, X1, X2 };
static unsigned int a[ 3 ] = { A0, A1, A2 };
static unsigned int c = C;

static void next( void );

void my_srand48( long seed )
{
   x[ 0 ] = X0;
   x[ 1 ] = unsigned( seed ) & MASK;                 // store low-order bits
   x[ 2 ] = ( unsigned( seed ) >> N_BITS ) & MASK;   // store high-order bits
   a[ 0 ] = A0;
   a[ 1 ] = A1;
   a[ 2 ] = A2;
   c = C;
}

double drand48( void )
{
   next();
   return TWO_16 * ( TWO_16 * ( TWO_16 * x[ 0 ] + x[ 1 ] ) + x[ 2 ] );
}

static void next( void )
{
   unsigned p[ 2 ], q[ 2 ], r[ 2 ];
   bool     carry0, carry1, carry2;
   long     prod;

   prod   = long( a[ 0 ] ) * long( x[ 0 ] );
   p[ 0 ] = unsigned( prod ) & MASK;
   p[ 1 ] = unsigned( prod >> N_BITS ) & MASK;

   carry0 = long( p[ 0 ] ) + long( c ) > MASK;
   carry1 = long( p[ 1 ] ) + long( carry0 ) > MASK;
   p[ 0 ] = unsigned( p[ 0 ] + c ) & MASK;
   p[ 1 ] = unsigned( p[ 1 ] + carry0 ) & MASK;

   prod   = long( a[ 0 ] ) * long( x[ 1 ] );
   q[ 0 ] = unsigned( prod ) & MASK;
   q[ 1 ] = unsigned( prod >> N_BITS ) & MASK;
```

```
   carry0 = long( p[ 1 ] ) + long( q[ 0 ] ) > MASK;
   p[ 1 ] = unsigned( p[ 1 ] +  q[ 0 ] ) & MASK;

   prod   = long( a[ 1 ] ) * long( x[ 0 ] );
   r[ 0 ] = unsigned( prod ) & MASK;
   r[ 1 ] = unsigned( prod >> N_BITS ) & MASK;

   carry2 = long( p[ 1 ] ) + long( r[ 0 ] ) > MASK;
   x[ 2 ] = unsigned( carry0 + carry1 + carry2 + q[ 1 ] + r[ 1 ] +
                      a[ 0 ] * x[ 2 ] + a[ 1 ] * x[ 1 ] +
                      a[ 2 ] * x[ 0 ] ) & MASK;
   x[ 1 ] = unsigned( p [ 1 ] + r[ 0 ] ) & MASK;
   x[ 0 ] = unsigned( p [ 0 ] ) & MASK;
}
```

ran0

This is the ''minimal'' random number generator of Park and Miller.[3] The constants are $a = 16,807$, $c = 0$, and $m = 2^{31} - 1$ (a Mersenne prime). It uses Schrage's method[4] to implement the recurrence formula without overflow of a 32-bit word. It has a period of $2^{31} - 2 = 2,147,483,646$.

```
// ran0.C: Minimal random number generator of Park and Miller.
//         Returns a uniform random deviate in [0,1) with a period of 2^31-2.
//         Set or reset seed to any integer value (except the value 0) to
//         initialize the sequence; seed must not be altered between calls for
//         successive deviates in the sequence.
// Ref: Press, W. H., et al., "Numerical Recipes in C", Cambridge, 1992, p. 278

#include <assert.h>

double ran0( long& seed )
{
   static const long   M = 2147483647;   // Mersenne prime 2^31-1
   static const long   A = 16807;        // 7^5 is a primitive root of M
   static const long   Q = 127773;
   static const long   R = 2836;
   static const double F = 1. / M;

   assert( seed != 0 );                   // since it won't work if seed = 0

   long k = seed / Q;                     // compute seed = ( A * seed ) % M
   seed = ( seed - k * Q ) * A - k * R;   // without overflow
   if ( seed < 0 ) seed += M;             // by Schrage's method

   return seed * F;                       // convert to a floating point
}
```

ran1

This is the same as `ran0`, with the same constants, but also makes use of Bays-Durham shuffle[5] to break up sequential correlations inherent in the linear congruential method.

```
// ran1.C: Random number generator of Park and Miller with Bays-Durham shuffle.
//         Returns a uniform random deviate in [0,1) with a period of 2^31-2.
//         Set the seed to any integer value (except zero) to initialize the
//         sequence; seed must not be altered between calls for successive
//         deviates in the sequence.
// Ref: Press, W. H., et al., "Numerical Recipes in C", Cambridge, 1992, p. 278

#include <assert.h>

double ran1( long& seed )
{
   static const long   M    = 2147483647;   // Mersenne prime 2^31-1
   static const long   A    = 16807;        // 7^5 is a primitive root of M
   static const long   Q    = 127773;
   static const long   R    = 2836;
   static const double F    = 1. / M;
   static const short  NTAB = 32;
```

---

3 Park, S. K., and K. W. Miller. *Communications of the ACM*. Vol. 31, pp. 1192–1201, 1988.

4 Schrage, L. *ACM Transactions on Mathematical Software*. Vol. 5, pp. 132–138, 1979.

5 Bays, C., and S. D. Durham. ''Improving a Poor Random Number Generator.'' *ACM Transactions on Mathematical Software*, Vol. 2, pp. 59–64, 1976.

```
   static const long   DIV  = 1 + ( M - 1 ) / NTAB;

   assert( seed != 0 );   // since it won't work if seed = 0

   static long value = 0;
   static long table[ NTAB ];

   if ( value == 0 ) {   // load the shuffle table the first time through

      for ( int i = NTAB + 7; i >= 0; i-- ) {   // first perform 8 warm-ups

         long k = seed / Q;
         seed = A * ( seed - k * Q ) - k * R;
         if ( seed < 0 ) seed += M;

         if ( i < NTAB ) table[ i ] = seed;
      }
      value = table[ 0 ];
   }

   long k = seed / Q;                     // compute seed = ( A * seed ) % M
   seed = A * ( seed - k * Q ) - k * R;   // without overflow
   if ( seed < 0 ) seed += M;             // by Schrage's method

   int i = value / DIV;                   // Bays-Durham shuffle algorithm
   value = table[ i ];
   table[ i ] = seed;

   return value * F;                      // return a floating point
}
```

ran1v2

This is version 2 of `ran1`. It uses the multiplier, $a = 48,271$.

```
// ran1v2.C: Random number generator of Park and Miller (version 2) with
//           Bays-Durham shuffle algorithm.
//           Returns a uniform random deviate in [0,1) with a period of 2^31-2.
//           Set the seed to any integer value (except zero) to initialize the
//           sequence; seed must not be altered between calls for successive
//           deviates in the sequence.
// Ref: Press, W. H., et al., "Numerical Recipes in C", Cambridge, 1992, p. 278

#include <assert.h>

double ran1v2( long& seed )
{
   static const long   M    = 2147483647;   // Mersenne prime 2^31-1
   static const long   A    = 48271;        // this is a prime number
   static const long   Q    = 44488;
   static const long   R    = 3399;
   static const double F    = 1. / M;
   static const short  NTAB = 32;
   static const long   DIV  = 1 + ( M - 1 ) / NTAB;

   assert( seed != 0 );   // since it won't work if seed = 0

   static long value = 0;
   static long table[ NTAB ];

   if ( value == 0 ) {   // load the shuffle table the first time through

      for ( int i = NTAB + 7; i >= 0; i-- ) {   // first perform 8 warm-ups

         long k = seed / Q;
         seed = A * ( seed - k * Q ) - k * R;
         if ( seed < 0 ) seed += M;

         if ( i < NTAB ) table[ i ] = seed;
      }
      value = table[ 0 ];
   }

   long k = seed / Q;                     // compute seed = ( A * seed ) % M
   seed = A * ( seed - k * Q ) - k * R;   // without overflow
   if ( seed < 0 ) seed += M;             // by Schrage's method

   int i = value / DIV;                   // Bays-Durham shuffle algorithm
   value = table[ i ];
```

```
   table[ i ] = seed;

   return value * F;                        // return a floating point
}
```

ran1v3

This is version 3 of `ran1`. It uses the multiplier $a = 69,621$.

```
// ran1v3.C: Minimal random number generator of Park and Miller (version 3 ).
//         Returns a uniform random deviate in [0,1) with a period of 2^31-2.
//         Set the seed to any integer value (except zero) to initialize the
//         sequence; seed must not be altered between calls for successive
//         deviates in the sequence.
// Ref: Press, W. H., et al., "Numerical Recipes in C", Cambridge, 1992, p. 278

#include <assert.h>

double ran1v3( long& seed )
{
   static const long   M    = 2147483647;   // Mersenne prime 2^31-1
   static const long   A    = 69621;
   static const long   Q    = 30845;
   static const long   R    = 23902;
   static const double F    = 1. / M;
   static const short  NTAB = 32;
   static const long   DIV  = 1 + ( M - 1 ) / NTAB;

   assert( seed != 0 );   // since it won't work if seed = 0

   static long value = 0;
   static long table[ NTAB ];

   if ( value == 0 ) {   // load the shuffle table the first time through

      for ( int i = NTAB + 7; i >= 0; i-- ) {   // first perform 8 warm-ups

         long k = seed / Q;
         seed = A * ( seed - k * Q ) - k * R;
         if ( seed < 0 ) seed += M;

         if ( i < NTAB ) table[ i ] = seed;
      }
      value = table[ 0 ];
   }

   long k = seed / Q;                       // compute seed = ( A * seed ) % M
   seed = A * ( seed - k * Q ) - k * R;     // without overflow
   if ( seed < 0 ) seed += M;               // by Schrage's method

   int i = value / DIV;                     // Bays-Durham shuffle algorithm
   value = table[ i ];
   table[ i ] = seed;

   return F * value;   // return a floating point
}
```

One of the most important considerations when choosing a random number generator is how many distinct random numbers it generates. Let N be the number of random deviates requested, and let n be the actual number of distinct random deviates generated. Table A-1 shows the results.

Table A-1. Capability to Generate Distinct Random Numbers

Number Requested, N	*Actual Number Generated (% requested)*					
	`rand`	`drand48`	`ran0`	`ran1`	`ran1v2`	`ran1v3`
10^2	100	100	100	100	100	100
10^3	98.20	100	100	100	100	100
10^4	86.54	100	99.97	100	100	100
10^5	31.15	100	99.80	100	100	100
10^6	3.28	100	98.06	100	100	100

It is not unreasonable to require 100,000 or more random numbers for a simulation, so these results alone should disqualify `rand` as a serious random number generator.

Next, let us compare the speed of the generators. Table A-2 shows the time to generate one million random deviates on a Silicon Graphics workstation with a 200–MHz processor.

Table A-2. Uniform Random Number Generator Timings

Number Requested, N	*Computer Time (s)*					
	`rand`	`drand48`	`ran0`	`ran1`	`ran1v2`	`ran1v3`
10^6	0.37	1.17	0.88	1.21	1.21	1.20

The relative timings are as follows.

`rand`: 1
`drand48`: 3.2
`ran0`: 2.4
`ran1`: 3.3
`ran1v2`: 3.3
`ran1v3`: 3.3

Although `drand48` is over three times slower than `rand`, it is a much better generator. Also, as is apparent from Table A-2, the computer time involved in generating uniform deviates is not likely to be a significant burden in a simulation.

Since we have the source code for all six of these generators, they all satisfy the portability requirement.

A.2 Validation Tests

Next, we subject the six candidate random number generators to four tests of randomness.

A.2.1 *Chi-Square Test*

The chi-square test is a check that the generated random numbers are distributed uniformly over the unit interval. In order to compute a value of χ^2 from our random number generators, we first subdivide the interval [0,1] into k subintervals of equal length and then count the number n_i that fall into the ith bin when a total of n random numbers have been generated. The computed value of χ^2 is obtained from the formula

$$\chi^2 = \frac{k}{n}\sum_{i=1}^{k}(n_i - n/k)^2, \tag{A-2}$$

where

k is the number of bins,
n is the total number of random deviates, and
n_i is the number of random deviates in the ith bin.

As a general rule when carrying out this test, k should be at least 100 and n/k should be at least 5. As the number of random samples, n, was increased, we increased the number of bins, k, such that the ratio n/k remained constant at 8. The critical value of χ^2 can be calculated with the aid of the formula, valid for large k,

$$\chi^2_{k-1,1-\alpha} = (k-1)\left(1 - \frac{2}{9(k-1)} + z_{1-\alpha}\sqrt{2/[9(k-1)]}\right)^3, \tag{A-3}$$

where

k is the numbers of bins,
α is the significance level, and
$z_{1-\alpha}$ is the $1-\alpha$ critical point of the standard normal distribution, and has the value 1.645 when $\alpha = 0.05$.

The results are displayed in Table A-3. (The seed used for all of these tests was 123456789.) All of the generators do about the same until the number of bins exceeds 32,768. Beyond this point, `rand` completely fails the chi-square

test. This is a manifestation of the fact that `rand` is only capable of generating, at most, 32,768 different random numbers (i.e., its range). Now, a good generator may still fail the test for a particular seed. Indeed, by definition, a perfectly good generator should fail the test 5% of the time. So, the fact that the other generators occasionally go slightly above the critical value is probably not significant. The fact that both `ran1v2` and `ran2v3` never fail the test may be significant but would have to be tested further.

Table A-3. Chi-Square Test Results (at a 0.05 Level of Significance)

Number of Samples, n	*Number of Bins, k*	*Critical Value*	*Computed Value of* χ^2					
			`rand`	`drand48`	`ran0`	`ran1`	`ran1v2`	`ran1v3`
1024	128	154	137	125	149	151	108	140
2048	256	293	264	264	268	268	259	271
4096	512	565	506	529	538	534	502	480
8192	1024	1099	912	1031	972	972	1026	984
16384	2048	2153	2064	2077	2021	2009	2065	1932
32768	4096	4245	4199	4248	4141	4135	4153	3945
65536	8192	8403	8246	8235	8416	8415	8170	8043
131072	16384	16682	16310	16634	16712	16718	16381	16196
262144	32768	33189	32737	32960	33122	33128	32703	32363
524288	65536	66132	588888	65577	66154	66167	65439	64893
1048576	131072	131914	3275060	130942	131715	131733	131104	130817

Notice that `ran1v3` does very well on this test. We also ran 20 independent tests with different seeds for the case when $n = 131,072$ and $k = 16,384$. We found that `ran1` failed the test three times (or 15% of the time), `ran1v2` failed the test two times (or 10% of the time), and `ran1v3` never failed.

A.2.2 *Sequential Correlation Tests*

Let $U_i \sim U(0,1)$ be the ith random number from a random number generator. Now, if the U_i's are really independent and identically distributed (IID) $U(0,1)$ random variates, then the nonoverlapping d-tuples

$$\mathbf{U_1} = (U_1, U_2, \cdots, U_d), \quad \mathbf{U_2} = (U_{d+1}, U_{d+2}, \cdots, U_{2d}), \quad \cdots \tag{A-4}$$

should be IID random *vectors* distributed uniformly on the d-dimensional unit hypercube, $[0,1)^d$. Let $n_{i_1 i_2 \cdots i_d}$ be the number of $\mathbf{r_i}$'s having first component in subinterval i_1, second component in subinterval i_2, $\cdots$, and dth component in subinterval i_d. Then the computed chi-square is given by the formula

$$\chi^2(d) = \frac{k^d}{n} \sum_{i_1=1}^{k} \sum_{i_2=1}^{k} \cdots \sum_{i_d=1}^{k} \left(n_{i_1 i_2 \cdots i_d} - \frac{n}{k^d} \right)^2. \tag{A-5}$$

That is, this quantity should have an approximate χ^2 distribution with $k^d - 1$ degrees of freedom. Table A-4 shows the test results for sequential pairs of random numbers.

Table A-4. Two-Dimensional Chi-Square Test Results (at a 0.05 Level of Significance)

Number of Samples, n	*Number of Bins, k*	*Critical Value*	*Computed Value of* χ^2					
			`rand`	`drand48`	`ran0`	`ran1`	`ran1v2`	`ran1v3`
2048	16^2	293	263	273	256	235	276	267
8192	32^2	1099	1031	1065	1066	1012	981	1053
32768	64^2	4245	4053	4164	4096	3956	4015	4163
131072	128^2	16682	16138	16412	16690	16380	16303	16283
524288	256^2	66132	65442	66009	65526	65788	65507	65168
2097152	512^2	263335	260149	263072	261968	262948	262913	261518

With one exception, `ran0` for 131072 samples, they all pass this test.

Table A-5 shows the test results for sequential triplets of random numbers.

Table A-5. Three-Dimensional Chi-Square Test Results (at a 0.05 Level of Significance)

Number of Samples, n	*Number of Bins, k*	*Critical Value*	*Computed Value of* χ^2					
			`rand`	`drand48`	`ran0`	`ran1`	`ran1v2`	`ran1v3`
512	4^3	83	54	65	52	69	63	63
4096	8^3	565	479	522	495	519	491	536
32768	16^3	4245	4210	4096	4224	4131	4071	4222
262144	32^3	33189	32872	32486	32558	32626	32654	32675
2097152	64^3	263335	262365	261818	261986	261716	262854	262002

All six generators pass this test.

A.2.3 *Runs-Up Test*

This is a test for independence. Each sequence of random numbers is examined for unbroken subsequences of maximal length within which the numbers increase monotonically; such a subsequence is called a *run-up*. Define

$$n_i = \begin{cases} \text{number of runs of length } i & \text{for } i = 1, 2, \cdots, 5 \\ \text{number of runs of length} \geq 6 & \text{for } i = 6 \end{cases} \tag{A-6}$$

and let n be the total length of the sequence. Then the test statistic is given by the formula

$$R = \frac{1}{n} \sum_{i=1}^{6} \sum_{j=1}^{6} A_{ij} \, (n_i - nb_i) \, (n_j - nb_j), \tag{A-7}$$

where A_{ij} is the ijth element of the matrix (Knuth 1969)

$$A = \begin{bmatrix} 4529.4 & 9044.9 & 13568. & 18091. & 22615. & 27892. \\ 9044.9 & 18097. & 27139. & 36187. & 45234. & 55789. \\ 13568. & 27139. & 40721. & 54281. & 67852. & 83685. \\ 18091. & 36187. & 54281. & 72414. & 90470. & 111580. \\ 22615. & 45234. & 67852. & 90470. & 113262. & 139476. \\ 27892. & 55789. & 83685. & 111580. & 139476. & 172860. \end{bmatrix} \tag{A-8}$$

and the b_i's are given by

$$(b_1, b_2, \cdots, b_n) = \left(\frac{1}{6}, \frac{5}{24}, \frac{11}{120}, \frac{19}{720}, \frac{29}{5040}, \frac{1}{840} \right). \tag{A-9}$$

For large n ($n \geq 4000$), R will have an approximate chi-square distribution with 6 degrees of freedom, so that $\chi^2_{6,0.95} = 12.6$ for an $\alpha = 0.05$ significance level. Table A-6 shows the results from this test.

Table A-6. Runs-Up Test Results (at a 0.05 Level of Significance)

Number of Samples, n	χ^2 *Critical Value*	*Computed Value of R*					
		`rand`	`drand48`	`ran0`	`ran1`	`ran1v2`	`ran1v3`
10^4	12.6	1.69	5.69	4.58	2.93	2.97	7.36
10^5	12.6	4.37	3.78	0.92	6.27	5.54	3.31
10^6	12.6	9.07	5.67	6.59	7.27	8.93	11.8

All the generators pass this test.

A.2.4 *Kolmogorov-Smirnov (K-S) Test*

This, like the chi-square test, is a test for uniformity. But, unlike the chi-square test, the K-S test does not require us to bin the data. Instead, it is a direct comparison between the empirical distribution and $F(x)$, the cumulative distribution—which in this case, is simply $F(x) = x$. The K-S statistic is the largest vertical distance between the theoretical distribution and the empirical distribution. In practice, we compute

$$D_n^+ = \max_{1 \le i \le n} \left[\frac{i}{n} - x_i \right], \qquad D_n^- = \max_{1 \le i \le n} \left[x_i - \frac{i-1}{n} \right] \quad \text{and} \quad D_n = \max \{ D_n^+, D_n^- \}. \tag{A-10}$$

We reject the null hypothesis that the generated numbers are uniformly distributed over the interval $[0, 1)$ if

$$\left(\sqrt{n} + 0.12 + \frac{0.11}{\sqrt{n}} \right) D_n > c_{1-\alpha}, \tag{A-11}$$

where the critical value $c_{1-\alpha}$ for $\alpha = 0.05$, is 1.358. Table A-7 shows the test results.

Table A-7. K-S Test Results (at a 0.05 Level of Significance)

Number of Samples, n	*K-S Critical Value*	*Computed Value of* $(\sqrt{n} + 0.12 + 0.11/\sqrt{n})D_n$					
		`rand`	`drand48`	`ran0`	`ran1`	`ran1v2`	`ran1v3`
10^3	1.358	0.860	0.821	0.794	0.700	0.651	0.543
10^4	1.358	0.780	0.693	0.948	0.928	0.394	0.860
10^5	1.358	0.870	0.830	0.956	0.950	1.035	0.943
10^6	1.358	0.613	0.697	1.021	1.026	1.085	0.650

We see that all six generators pass this test.

From these four validation test results, we see that, with the exception of `rand`, all the generators are about the same. Overall, `ran1v3` seems to perform the best, and so it was chosen as the uniform random number generator in the Random class. Incidently, it should be mentioned that the Random class permits more than one independent random number stream. For example, if we set

```
Random rv1( seed1 ), rv2( seed2 );
```

then `rv1` and `rv2` are distinct objects so that the stream generated by `rv1` alone is not altered by the presence of `rv2`. This can be useful if we want to vary a selected stochastic process while retaining all other source of stochasticism.

INTENTIONALLY LEFT BLANK.

APPENDIX B:
RANDOM CLASS SOURCE CODE

The definition and implementation of the Random class is consolidated into a single header file, `Random.h`. As a consequence, it is only necessary to include this header file in any program that makes use of random number distributions. For example, here is a simple program that makes use of the Random class.

```
// Sample program for using the Random class

#include <iostream.h>
#include "Random.h"                    ⇐ include the definition of the Random class

void main( void )
{
   Random rv;                          ⇐ declare a random variate

   for ( int i = 0; i < 1000; i++ )
      cout << rv.normal()              ⇐ reference the normal distribution (with default parameters)
           << endl;
}
```

If this code is contained in the file `main.C`, then it can be compiled and linked into an executable program, `main`, with the aid of the GNU C++ compiler by invoking the following UNIX command.

```
c++ -o main main.C -lm
```

This program will set the random number seed from the UNIX process ID. If we want to set the seed explicitly to 123456789, simply make the following replacement.

```
Random rv;   ⇒   Random rv( 123456789 );
```

If at any time later in the program we want to reset the seed, to say 773, we can do that with the following statement.

```
Random rv.reset( 773 );
```

The remaining pages of this appendix contain the complete source code of the `Random` class.

```
// Random.h: Definition and Implementation of Random Number Distribution Class

#ifndef RANDOM_H
#define RANDOM_H

#include <fstream.h>
#include <math.h>
#include <limits.h>
#include <unistd.h>
#include <assert.h>
#include <stl.h>

// Constants for Tausworthe random bit generator
// Ref: Tausworthe, Robert C., "Random Numbers Generated by Linear Recurrence
//      Modulo Two," Mathematics of Computation, vol. 19, pp. 201-209, 1965.

static const unsigned DEGREE_MAX = 32;   // max degree (bits per word)

static const unsigned BIT[ 1 + DEGREE_MAX ] = {

// Hexidecimal       Value      Degree
// -----------       -----      ------
   0x00000000,    // 0          0
   0x00000001,    // 2^0        1
   0x00000002,    // 2^1        2
   0x00000004,    // 2^2        3
   0x00000008,    // 2^3        4
   0x00000010,    // 2^4        5
   0x00000020,    // 2^5        6
   0x00000040,    // 2^6        7
   0x00000080,    // 2^7        8
   0x00000100,    // 2^8        9
   0x00000200,    // 2^9        10
   0x00000400,    // 2^10       11
   0x00000800,    // 2^11       12
   0x00001000,    // 2^12       13
   0x00002000,    // 2^13       14
   0x00004000,    // 2^14       15
   0x00008000,    // 2^15       16
   0x00010000,    // 2^16       17
   0x00020000,    // 2^17       18
   0x00040000,    // 2^18       19
   0x00080000,    // 2^19       20
   0x00100000,    // 2^20       21
   0x00200000,    // 2^21       22
   0x00400000,    // 2^22       23
   0x00800000,    // 2^23       24
   0x01000000,    // 2^24       25
   0x02000000,    // 2^25       26
   0x04000000,    // 2^26       27
   0x08000000,    // 2^27       28
   0x10000000,    // 2^28       29
   0x20000000,    // 2^29       30
   0x40000000,    // 2^30       31
   0x80000000     // 2^31       32
};

// Coefficients that define a primitive polynomial (mod 2)
// Ref: Watson, E. J., "Primitive Polynomials (Mod 2),"
//      Mathematics of Computation, vol. 16, pp. 368-369, 1962.

static const unsigned MASK[ 1 + DEGREE_MAX ] = {

   BIT[0],                                    // 0
   BIT[0],                                    // 1
   BIT[1],                                    // 2
   BIT[1],                                    // 3
   BIT[1],                                    // 4
   BIT[2],                                    // 5
   BIT[1],                                    // 6
   BIT[1],                                    // 7
   BIT[4] + BIT[3] + BIT[2],                  // 8
   BIT[4],                                    // 9
   BIT[3],                                    // 10
   BIT[2],                                    // 11
   BIT[6] + BIT[4] + BIT[1],                  // 12
   BIT[4] + BIT[3] + BIT[1],                  // 13
   BIT[5] + BIT[3] + BIT[1],                  // 14
   BIT[1],                                    // 15
   BIT[5] + BIT[3] + BIT[2],                  // 16
   BIT[3],                                    // 17
   BIT[5] + BIT[2] + BIT[1],                  // 18
   BIT[5] + BIT[2] + BIT[1],                  // 19
   BIT[3],                                    // 20
   BIT[2],                                    // 21
   BIT[1],                                    // 22
   BIT[5],                                    // 23
   BIT[4] + BIT[3] + BIT[1],                  // 24
   BIT[3],                                    // 25
   BIT[6] + BIT[2] + BIT[1],                  // 26
   BIT[5] + BIT[2] + BIT[1],                  // 27
   BIT[3],                                    // 28
   BIT[2],                                    // 29
```

```
   BIT[6] + BIT[4] + BIT[1],                    // 30
   BIT[3],                                      // 31
   BIT[7] + BIT[5] + BIT[3] + BIT[2] + BIT[1]   // 32
};

// for convenience, define some data structures for bivariate distributions

struct cartesianCoord {   // cartesian coordinates in 2-D

   double x, y;
   cartesianCoord& operator+=( const cartesianCoord& p ) {
      x += p.x;
      y += p.y;
      return *this;
   }
   cartesianCoord& operator-=( const cartesianCoord& p ) {
      x -= p.x;
      y -= p.y;
      return *this;
   }
   cartesianCoord& operator*=( double scalar ) {
      x *= scalar;
      y *= scalar;
      return *this;
   }
   cartesianCoord& operator/=( double scalar ) {
      x /= scalar;
      y /= scalar;
      return *this;
   }
};

struct sphericalCoord {  // spherical coordinates on unit sphere

   double theta, phi;
   double x( void ) { return sin( theta ) * cos( phi ); }   // x-coordinate
   double y( void ) { return sin( theta ) * sin( phi ); }   // y-coordinate
   double z( void ) { return cos( theta ); }                // z-coordinate
};

class Random {

// friends list
// overloaded relational operators

   friend bool operator==( const Random& p, const Random& q )
   {
      bool equal = ( p._seed == q._seed ) && ( p._next == q._next );
      for ( int i = 0; i < p._NTAB; i++ )
         equal = equal && ( p._table[ i ] == q._table[ i ] );
      return equal;
   }

   friend bool operator!=( const Random& p, const Random& q )
   {
      return !( p == q );
   }

// overloaded stream operator

   friend istream& operator>>( istream& is, Random& rv )
   {
      cout << "Enter a random number seed "
           << "(between 1 and " << LONG_MAX - 1 << ", inclusive): " << endl;

      is >> rv._seed;

      assert( rv._seed != 0 && rv._seed != LONG_MAX );

      rv._seedTable();

      return is;
   }

public:

   Random( long seed )   // constructor to set the seed
   {
      assert( seed != 0 && seed != LONG_MAX );
      _seed = seed;
      _seedTable();

      _seed2 = _seed | 1;  // for tausworthe random bit generation
   }

   Random( void )   // default constructor uses process id to set the seed
   {
      _seed = long( getpid() );
      _seedTable();
      _seed2 = _seed | 1;  // for tausworthe random bit generation
   }

  ~Random( void )   // default destructor
```

```
   {
   }

   Random( const Random& r )   // copy constructor (copies current state)
   {
      _seed  = r._seed;
      _seed2 = r._seed2;

      // copy the current state of the shuffle table

      _next = r._next;
      for ( int i = 0; i < _NTAB; i++ ) _table[ i ] = r._table[ i ];
   }

   Random& operator=( const Random& r )   // overloaded assignment
   {
      if ( *this == r ) return *this;

      _seed  = r._seed;
      _seed2 = r._seed2;

      // copy the current state of the shuffle table

      _next = r._next;
      for ( int i = 0; i < _NTAB; i++ ) _table[ i ] = r._table[ i ];

      return *this;
   }

// utility functions

   void reset( long seed )   // reset the seed explicitly
   {
      assert( seed != 0 && seed != LONG_MAX );
      _seed = seed;
      _seedTable();
      _seed2 = _seed | 1;   // so that all bits cannot be zero
   }

   void reset( void )   // reset seed from current process id
   {
      _seed = long( getpid() );
      _seedTable();
      _seed2 = _seed | 1;   // so that all bits cannot be zero
   }

// Continuous Distributions

   double arcsine( double xMin = 0., double xMax = 1. )   // Arc Sine
   {
      double q = sin( M_PI_2 * _u() );
      return xMin + ( xMax - xMin ) * q * q;
   }

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

   double cauchy( double a = 0., double b = 1. )   // Cauchy (or Lorentz)
   {
      // a is the location parameter and b is the scale parameter
      // b is the half width at half maximum (HWHM) and variance doesn't exist

      assert( b > 0. );

      return a + b * tan( M_PI * uniform( -0.5, 0.5 ) );
   }

   double chiSquare( int df )   // Chi-Square
   {
      assert( df >= 1 );

      return gamma( 0., 2., 0.5 * double( df ) );
   }

   double cosine( double xMin = 0., double xMax = 1. )   // Cosine
   {
      assert( xMin < xMax );

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

      return a + b * asin( uniform( -1., 1. ) );
   }

   double doubleLog( double xMin = -1., double xMax = 1. )   // Double Log
   {
      assert( xMin < xMax );
```

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

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

double erlang( double b, int c )   // Erlang (b > 0. and c >= 1)
{
   assert( b > 0. && c >= 1 );

   double prod = 1.;
   for ( int i = 0; i < c; i++ ) prod *= _u();

   return -b * log( prod );
}

double exponential( double a = 0., double c = 1. )   // Exponential
{                                                     // location a, shape c
   assert( c > 0.0 );

   return a - c * log( _u() );
}

double extremeValue( double a = 0., double c = 1. )   // Extreme Value
{                                                      // location a, shape c
   assert( c > 0. );

   return a + c * log( -log( _u() ) );
}

double fRatio( int v, int w )   // F Ratio (v and w >= 1)
{
   assert( v >= 1 && w >= 1 );

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

double gamma( double a, double b, double c )  // Gamma
{                                             // location a, scale b, shape c
   assert( b > 0. && c > 0. );

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

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

double laplace( double a = 0., double b = 1. )   // Laplace
{                                                // (or double exponential)
   assert( b > 0. );

   // composition method

   if ( bernoulli( 0.5 ) ) return a + b * log( _u() );
   else                    return a - b * log( _u() );
}

double logarithmic( double xMin = 0., double xMax = 1. )   // Logarithmic
{
   assert( xMin < xMax );

   double a = xMin;          // location parameter
   double b = xMax - xMin;   // scale parameter
```

```
   // use convolution formula for product of two IID uniform variates

   return a + b * _u() * _u();
}

double logistic( double a = 0., double c = 1. )   // Logistic
{
   assert( c > 0. );

   return a - c * log( 1. / _u() - 1. );
}

double lognormal( double a, double mu, double sigma )   // Lognormal
{
   return a + exp( normal( mu, sigma ) );
}

double normal( double mu = 0., double sigma = 1. )   // Normal
{
   assert( sigma > 0. );

   static bool f = true;
   static double p, p1, p2;
   double q;

   if ( f ) {
      do {
         p1 = uniform( -1., 1. );
         p2 = uniform( -1., 1. );
         p = p1 * p1 + p2 * p2;
      } while ( p >= 1. );
      q = p1;
   }
   else
      q = p2;
   f = !f;

   return mu + sigma * q * sqrt( -2. * log( p ) / p );
}

double parabolic( double xMin = 0., double xMax = 1. )   // Parabolic
{
   assert( xMin < xMax );

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

   return userSpecified( _parabola, xMin, xMax, 0., yMax );
}

double pareto( double c )   // Pareto
{                           // shape c
   assert( c > 0. );

   return pow( _u(), -1. / c );
}

double pearson5( double b, double c )   // Pearson Type 5
{                                       // scale b, shape c
   assert( b > 0. && c > 0. );

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

double pearson6( double b, double v, double w )   // Pearson Type 6
{                                                 // scale b, shape v & w
   assert( b > 0. && v > 0. && w > 0. );

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

double power( double c )   // Power
{                          // shape c
   assert( c > 0. );

   return pow( _u(), 1. / c );
}

double rayleigh( double a, double b )   // Rayleigh
{                                       // location a, scale b
   assert( b > 0. );

   return a + b * sqrt( -log( _u() ) );
}

double studentT( int df )   // Student's T
{                           // degres of freedom df
   assert( df >= 1 );

   return normal() / sqrt( chiSquare( df ) / df );
}

double triangular( double xMin = 0.,     // Triangular
```

```
                    double xMax = 1.,     // with default interval [0,1)
                    double c    = 0.5 )   // and default mode 0.5
   {
      assert( xMin < xMax && xMin <= c && c <= xMax );

      double p = _u(), q = 1. - p;

      if ( p <= ( c - xMin ) / ( xMax - xMin ) )
         return xMin + sqrt( ( xMax - xMin ) * ( c - xMin ) * p );
      else
         return xMax - sqrt( ( xMax - xMin ) * ( xMax - c ) * q );
   }

   double uniform( double xMin = 0., double xMax = 1. )   // Uniform
   {                                                      // on [xMin,xMax)
      assert( xMin < xMax );

      return xMin + ( xMax - xMin ) * _u();
   }

   double userSpecified(                // User-Specified Distribution
        double( *usf )(                 // pointer to user-specified function
             double,                    // x
             double,                    // xMin
             double ),                  // xMax
        double xMin, double xMax,       // function domain
        double yMin, double yMax )      // function range
   {
      assert( xMin < xMax && yMin < yMax );

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

      // acceptance-rejection method

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

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

      return x;
   }

   double weibull( double a, double b, double c )   // Weibull
   {                                                // location a, scale b,
      assert( b > 0. && c > 0. );                   // shape c

      return a + b * pow( -log( _u() ), 1. / c );
   }

// Discrete Distributions

   bool bernoulli( double p = 0.5 )   // Bernoulli Trial
   {
      assert( 0. <= p && p <= 1. );

      return _u() < p;
   }

   int binomial( int n, double p )   // Binomial
   {
      assert( n >= 1 && 0. <= p && p <= 1. );

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

   int geometric( double p )   // Geometric
   {
      assert( 0. < p && p < 1. );

      return int( log( _u() ) / log( 1. - p ) );
   }

   int hypergeometric( int n, int N, int K )            // Hypergeometric
   {                                                    // trials n, size N,
      assert( 0 <= n && n <= N && N >= 1 && K >= 0 );   // successes K

      int count = 0;
      for ( int i = 0; i < n; i++, N-- ) {

         double p = double( K ) / double( N );
         if ( bernoulli( p ) ) { count++; K--; }
      }
      return count;
   }

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

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

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

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

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

         double lower = 0., upper = 0., u = _u();

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

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

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

   int negativeBinomial( int s, double p )   // Negative Binomial
   {                                         // successes s, probability p
      assert( s >= 1 && 0. < p && p < 1. );

      int sum = 0;
      for ( int i = 0; i < s; i++ ) sum += geometric( p );
      return sum;
   }

   int pascal( int s, double p )              // Pascal
   {                                          // successes s, probability p
      return negativeBinomial( s, p ) + s;
   }

   int poisson( double mu )   // Poisson
   {
      assert ( mu > 0. );

      double a = exp( -mu );
      double b = 1.;

      int i;
      for ( i = 0; b >= a; i++ ) b *= _u();
      return i - 1;
   }

   int uniformDiscrete( int i, int j )   // Uniform Discrete
   {                                     // inclusive i to j
      assert( i < j );

      return i + int( ( j - i + 1 ) * _u() );
   }

// Empirical and Data-Driven Distributions

   double empirical( void )   // Empirical Continuous
   {
      static vector< double > x, cdf;
      static int              n;
      static bool             init = false;

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

         // check that this is indeed a cumulative distribution

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

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

```
int empiricalDiscrete( void )   // Empirical Discrete
{
   static vector< int >    k;
   static vector< double > f[ 2 ];   // f[ 0 ] is pdf and f[ 1 ] is cdf
   static double           max;
   static int              n;
   static bool             init = false;

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

      // form the cumulative distribution

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

      // check that the integer points are in ascending order

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

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

   // select a uniform variate between 0 and the max value of the cdf

   double p = uniform( 0., max );

   // locate and return the corresponding index

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

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

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

         // shuffle contents of v once and for all
         // Ref: Knuth, D. E., The Art of Computer Programming, Vol. 2:
         //      Seminumerical Algorithms. London: Addison-Wesley, 1969.

         for ( int i = n - 1; i > 0; i-- ) {
            int j = int( ( i + 1 ) * _u() );
            swap( v[ i ], v[ j ] );
         }
      }
   }

   // return a random sample

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

void sample( double x[], int ndim )   // Sample from a given distribution
{                                     // of multi-dimensional data
   static const int N_DIM = 6;
   assert( ndim <= N_DIM );

   static vector< double > v[ N_DIM ];
```

```
   static bool init = false;
   static int n;

   if ( !init )  {
      ifstream in( "sampleData" );
      if ( !in ) {
         cerr << "Cannot open
         exit( 1 );
      }
      double d;
      while ( !in.eof() ) {
         for ( int i = 0; i < ndim; i++ ) {
            in >> d;
            v[ i ].push_back( d );
         }
      }
      in.close();
      n = v[ 0 ].size();
      init = true;
   }
   int index = uniformDiscrete( 0, n - 1 );
   for ( int i = 0; i < ndim; i++ ) x[ i ] = v[ i ][ index ];
}

// comparison functor for determining the neighborhood of a data point

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

cartesianCoord stochasticInterpolation( void )   // Stochastic Interpolation

// Refs: Taylor, M. S. and J. R. Thompson, Computational Statistics & Data
//       Analysis, Vol. 4, pp. 93-101, 1986; Thompson, J. R., Empirical Model
//       Building, pp. 108-114, Wiley, 1989; Bodt, B. A. and M. S. Taylor,
//       A Data Based Random Number Generator for A Multivariate Distribution
//       - A User's Manual, ARBRL-TR-02439, BRL, APG, MD, Nov. 1982.
{
   static vector< cartesianCoord > data;
   static cartesianCoord           min, max;
   static int                      m;
   static double                   lower, upper;
   static bool                     init = false;

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

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

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

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

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

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

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

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

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

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

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

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

   cartesianCoord origin = data[ uniformDiscrete( 0, data.size() - 1 ) ];
```

```
      // make this point the origin of the coordinate system

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

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

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

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

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

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

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

         double rn;
         if ( m == 1 ) rn = 1.;
         else          rn = uniform( lower, upper );
         p.x += rn * ( data[ n ].x - mean.x );
         p.y += rn * ( data[ n ].y - mean.y );
      }

      // restore the data to its original form

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

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

      p += mean;
      p += origin;

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

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

      return p;
   }

// Multivariate Distributions

   cartesianCoord bivariateNormal( double muX    = 0.,   // Bivariate Gaussian
                                   double sigmaX = 1.,
                                   double muY    = 0.,
                                   double sigmaY = 1. )
   {
      assert( sigmaX > 0. && sigmaY > 0. );

      cartesianCoord p;
      p.x = normal( muX, sigmaX );
      p.y = normal( muY, sigmaY );
      return p;
   }

   cartesianCoord bivariateUniform( double xMin = -1.,    // Bivariate Uniform
                                    double xMax =  1.,
                                    double yMin = -1.,
                                    double yMax =  1. )
   {
      assert( xMin < xMax && yMin < yMax );

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

      do {
         x = uniform( -1., 1. );
         y = uniform( -1., 1. );

      } while( x * x + y * y > 1. );

      cartesianCoord p;
      p.x = x0 + a * x;
      p.y = y0 + b * y;
      return p;
   }

   cartesianCoord corrNormal( double r,              // Correlated Normal
                              double muX    = 0.,
                              double sigmaX = 1.,
                              double muY    = 0.,
                              double sigmaY = 1. )
   {
      assert( -1. <= r && r <= 1. &&          // bounds on correlation coeff
              sigmaX > 0. && sigmaY > 0. );   // positive std dev
```

```
      double x = normal();
      double y = normal();

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

      cartesianCoord p;
      p.x = muX + sigmaX * x;                 // translate and scale
      p.y = muY + sigmaY * y;
      return p;
   }

   cartesianCoord corrUniform( double r,        // Correlated Uniform
                               double xMin = 0.,
                               double xMax = 1.,
                               double yMin = 0.,
                               double yMax = 1. )
   {
      assert( -1. <= r && r <= 1. &&          // bounds on correlation coeff
              xMin < xMax && yMin < yMax );   // bounds on domain

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

      do {
         x = uniform( -1., 1. );
         y = uniform( -1., 1. );

      } while ( x * x + y * y > 1. );

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

      cartesianCoord p;
      p.x = x0 + a * x;                     // translate and scale
      p.y = y0 + b * y;
      return p;
   }

   sphericalCoord spherical( double thMin = 0.,     // Uniform Spherical
                             double thMax = M_PI,
                             double phMin = 0.,
                             double phMax = 2. * M_PI )
   {
      assert( 0. <= thMin && thMin < thMax && thMax <= M_PI &&
              0. <= phMin && phMin < phMax && phMax <= 2. * M_PI );

      sphericalCoord p;                         // azimuth
      p.theta = acos( uniform( cos( thMax ), cos( thMin ) ) );   // polar angle
      p.phi   = uniform( phMin, phMax );                         // azimuth
      return p;
   }

   void sphericalND( double x[], int n )   // Uniform over the surface of
                                           // an n-dimensional unit sphere
   {
      // generate a point inside the unit n-sphere by normal polar method

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

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

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

// Number Theoretic Distributions

   double avoidance( void )   // Maximal Avoidance (1-D)
   {                          // overloaded for convenience
      double x[ 1 ];
      avoidance( x, 1 );
      return x[ 0 ];
   }

   void avoidance( double x[], int ndim )   // Maximal Avoidance (N-D)
   {
      static const int MAXBIT = 30;
      static const int MAXDIM = 6;

      assert( ndim <= MAXDIM );

      static unsigned long ix[ MAXDIM + 1 ] = { 0 };
      static unsigned long *u[ MAXBIT + 1 ];
      static unsigned long mdeg[ MAXDIM + 1 ] = { // degree of
         0,                                       // primitive polynomial
         1, 2, 3, 3, 4, 4
      };
```

```
      static unsigned long p[ MAXDIM + 1 ] = {   // decimal encoded
         0,                                      // interior bits
         0, 1, 1, 2, 1, 4
      };
      static unsigned long v[ MAXDIM * MAXBIT + 1 ] = {
          0,
          1,  1, 1,  1,  1,  1,
          3,  1, 3,  3,  1,  1,
          5,  7, 7,  3,  3,  5,
         15, 11, 5, 15, 13,  9
      };

      static double fac;
      static int in = -1;
      int j, k;
      unsigned long i, m, pp;

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

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

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

   void tausworthe( bool*    bitvec,   // Tausworthe random bit generator
                    unsigned n )       // returns a bit vector of length n

   // It is guaranteed to cycle through all possible combinations of n bits
   // (except all zeros) before repeating, i.e., cycle has maximal length 2^n-1.
   // Ref: Press, W. H., B. P. Flannery, S. A. Teukolsky and W. T. Vetterling,
   //      Numerical Recipes in C, Cambridge Univ. Press, Cambridge, 1988.
   {
      assert( 1 <= n && n <= 32 );   // length of bit vector

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

private:

   static const long   _M    = 0x7fffffff; // 2147483647 (Mersenne prime 2^31-1)
   static const long   _A    = 0x10ff5;    // 69621
   static const long   _Q    = 0x787d;     // 30845
   static const long   _R    = 0x5d5e;     // 23902
   static const double _F    = 1. / _M;
   static const short  _NTAB = 32;         // arbitrary length of shuffle table
   static const long   _DIV  = 1+(_M-1)/_NTAB;
   long         _table[ _NTAB ];          // shuffle table of seeds
   long         _next;                    // seed to be used as index into table
   long         _seed;                    // current random number seed
   unsigned     _seed2;                   // seed for tausworthe random bit

   void _seedTable( void )                          // seeds the shuffle table
   {
      for ( int i = _NTAB + 7; i >= 0; i-- ) {      // first perform 8 warm-ups
```

```
         long k = _seed / _Q;                        // seed = ( A * seed ) % M
         _seed = _A * ( _seed - k * _Q ) - k * _R;   // without overflow by
         if ( _seed < 0 ) _seed += _M;               // Schrage's method

         if ( i < _NTAB ) _table[ i ] = _seed;       // store seeds into table
      }
      _next = _table[ 0 ];                           // used as index next time
   }

   double _u( void )                                 // uniform rng
   {
      long k = _seed / _Q;                           // seed = ( A*seed ) % M
      _seed = _A * ( _seed - k * _Q ) - k * _R;      // without overflow by
      if ( _seed < 0 ) _seed += _M;                  // Schrage's method

      int index = _next / _DIV;                      // Bays-Durham shuffle
      _next = _table[ index ];                       // seed used for next time
      _table[ index ] = _seed;                       // replace with new seed

      return _next * _F;                             // scale value within [0,1)
   }

   static double _parabola( double x, double xMin, double xMax )  // parabola
   {
      if ( x < xMin || x > xMax ) return 0.0;

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

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

#endif
```

GLOSSARY

Variate

A random variable from a probability distribution. Typically, capital letters $X, Y, \cdots$ are used to denote variates.

U ~ U (0,1)

Signifies a variate U is drawn or sampled from the uniform distribution.

IID

Independent and identically distributed.

Probability Density Function (PDF)

$f(k)$ for a discrete distribution.
$f(x)$ for a continuous distribution.

Cumulative Distribution Function (CDF)

$F(k)$ for a discrete distribution.
$F(x)$ for a continuous distribution.

Mode

The value of k where $f(k)$ is a global maximum for a discrete distribution.
The value of x where $f(x)$ is a global maximum for a continuous distribution.

Median

The point such that half the values are greater for a discrete distribution. If the points are ordered from smallest to largest, then

$$k_{\text{median}} = \begin{cases} k_{(n+1)/2} & \text{if } n \text{ is odd} \\ (k_{n/2} + k_{n/2+1})/2 & \text{if } n \text{ is even} \end{cases}$$

$F(x_{\text{median}}) = 1/2$ for a continuous distribution.

Mean

$\bar{k} = \sum_{\text{all values of } k} k\ f(k)$ for a discrete distribution.

$\bar{x} = \int_{-\infty}^{\infty} x f(x) dx$ for a continuous distribution.

Variance

$\sigma^2 = \sum_{\text{all values of } k} (k - \bar{k})^2 f(k)$ for a discrete distribution.

$\sigma^2 = \int_{-\infty}^{\infty} (x - \bar{x})^2 f(x) dx$ for a continuous distribution.

Standard Deviation

σ, the square root of the variance.

Skewness (as defined by Pearson)

S_k = (mean – mode) / standard deviation.

Chebyshev's Inequality

For *any* distribution, $Prob\{ |x - \bar{x}| \geq k\sigma_x \} \leq \frac{1}{k^2}$, where k is any positive real number.

Leibniz's Rule

$$\frac{d}{dx} \int_{a(x)}^{b(x)} f(u, x)\, du = \int_{a(x)}^{b(x)} \frac{\partial f}{\partial x}\, du + f(b(x), x) \frac{db(x)}{dx} - f(a(x), x) \frac{da(x)}{dx}\ .$$

INTENTIONALLY LEFT BLANK.