Wolfram *Mathematica*® Tutorial Collection

RANDOM NUMBER GENERATION

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Content authored by:
Darren Glosemeyer and Rob Knapp

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Contents

Random Number Generation

Introduction

The ability to generate pseudorandom numbers is important for simulating events, estimating probabilities and other quantities, making randomized assignments or selections, and numerically testing symbolic results. Such applications may require uniformly distributed numbers, nonuniformly distributed numbers, elements sampled with replacement, or elements sampled without replacement.

The functions `RandomReal`, `RandomInteger`, and `RandomComplex` generate uniformly distributed random numbers. `RandomReal` and `RandomInteger` also generate numbers for built-in distributions. `RandomPrime` generates primes within a range. The functions `RandomChoice` and `RandomSample` sample from a list of values with or without replacement. The elements may have equal or unequal weights. A framework is also included for defining additional methods and distributions for random number generation.

A sequence of nonrecurring events can be simulated via `RandomSample`. For instance, the probability of randomly sampling the integers 1 through n in order might be simulated.

This estimates the probability of getting n elements in order for n from 2 to 8.

```
In[1]:= Block[{trials = 10^5, count, rn},
         Table[
          count = 0;
          rn = Range[n];
          Do[If[RandomSample[rn] == rn, count++], {trials}];
          {n, N[count / trials]},
          {n, 2, 8}]]
Out[1]= {{2, 0.50127}, {3, 0.1656}, {4, 0.04225}, {5, 0.00824}, {6, 0.00144}, {7, 0.0002}, {8, 0.00002}}
```

The results can be compared with the theoretical probabilities.

```
In[2]:= Table[{i, N[1 / i!]}, {i, 2, 8}]
Out[2]= {{2, 0.5}, {3, 0.166667}, {4, 0.0416667},
         {5, 0.00833333}, {6, 0.00138889}, {7, 0.000198413}, {8, 0.0000248016}}
```

Random number generation is at the heart of Monte Carlo estimates. An estimate of an expected value of a function f can be obtained by generating values from the desired distribution and finding the mean of f applied to those values.

This estimates the 6^{th} raw moment for a normal distribution.

```
In[3]:= Mean[RandomReal[NormalDistribution[0, 2], 10^6]^6]
Out[3]= 961.612
```

In this case, the estimate can be compared with an exact result.

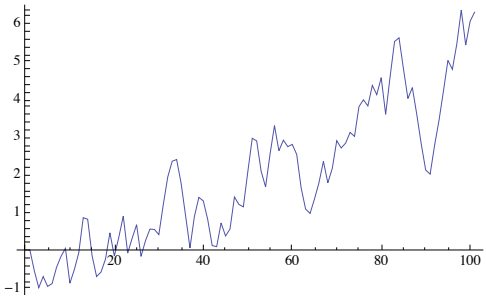
```
In[4]:= ExpectedValue[x^6, NormalDistribution[0, 2], x]
Out[4]= 960
```

Random processes can be simulated by generating a series of numbers with the desired properties. A random walk can be created by recursively summing pseudorandom numbers.

Here a random walk starting at 0 is created.

```
In[5]:= ListLinePlot[Join[{0.}, Accumulate[RandomReal[{-1, 1}, {100}]]]]
```

Out[5]=



Substitution of random numbers can be used to test the equivalence of symbolic expressions. For instance, the absolute difference between two expressions could be evaluated at randomly generated points to test for inequality of the expressions.

This provides no evidence that $\sqrt{x^2}$ and $|x|$ are different for real values.

```
In[6]:= Max[Abs[Sqrt[x^2] - Abs[x] /. x → RandomReal[{-10, 10}, 10 000]]]
Out[6]= 4.44089×10^-16
```

This provides evidence that $\sqrt{x^2}$ and $|x|$ differ for at least some complex values.

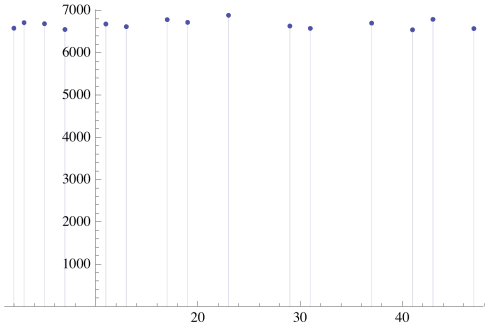
```
In[7]:= Max[Abs[Sqrt[x^2] - Abs[x] /. x → RandomComplex[{-10 + 10 I, 10 + 10 I}, 10 000]]]
Out[7]= 14.1415
```

RandomPrime chooses prime numbers with equal probability, which can be useful—for instance, to generate large primes for RSA encryption. The prime numbers are uniformly distributed on the primes in the range but are not uniformly distributed on the entire range because primes are in general not uniformly distributed over ranges of positive integers.

Primes in a given range are generated with equal probability.

```
In[8]:= ListPlot[Tally[RandomPrime[50, 10^5]], Filling → Axis, PlotRange → {0, Automatic}]
```

Out[8]=



Random Generation Functions

The main functions are RandomReal, RandomInteger, RandomComplex, RandomChoice, and RandomSample. RandomReal, RandomInteger, and RandomComplex generate numbers given some range of numeric values. RandomChoice and RandomSample generate elements from finite sets that may include non-numeric values.

Random Numbers

RandomReal generates pseudorandom real numbers over a specified range of real values. RandomInteger generates pseudorandom integer numbers over a specified range of integer values. RandomComplex generates pseudorandom complex numbers over a specified rectangular region in the complex plane. RandomPrime generates prime numbers with equal probability within a range.

RandomReal[]	give a pseudorandom real number in the range 0 to 1
RandomReal[{x_{min}, x_{max}}]	give a pseudorandom real number in the range x_{min} to x_{max}
RandomReal[x_{max}]	give a pseudorandom real number in the range 0 to x_{max}
RandomReal[*dist*]	give a random number from the continuous distribution *dist*
RandomReal[*domain*, n]	give a list of n pseudorandom reals
RandomReal[*domain*, {n_1, n_2, ...}]	give an $n_1 \times n_2 \times \ldots$ array of pseudorandom reals

Generation of random reals.

RandomInteger[{i_{min}, i_{max}}]	give a pseudorandom integer in the range {i_{min}, ..., i_{max}}
RandomInteger[i_{max}]	give a pseudorandom integer in the range {0, ..., i_{max}}
RandomInteger[]	pseudorandomly give 0 or 1 with probability $\frac{1}{2}$
RandomInteger[*dist*]	give a pseudorandom integer from the discrete distribution *dist*
RandomInteger[*domain*, n]	give a list of n pseudorandom integers
RandomInteger[*domain*, {n_1, n_2, ...}]	give an $n_1 \times n_2 \times \ldots$ array of pseudorandom integers

Generation of random integers.

RandomComplex[]	give a pseudorandom complex number in the unit square
RandomComplex[{z_{min}, z_{max}}]	give a pseudorandom complex number in the rectangle bounded by z_{min} and z_{max}
RandomComplex[z_{max}]	give a pseudorandom complex number in the rectangle bounded by 0 and z_{max}
RandomComplex[*domain*, n]	give a list of n pseudorandom complex numbers
RandomComplex[*domain*, {n_1, n_2, ...}]	give an $n_1 \times n_2 \times \ldots$ array of pseudorandom complex numbers

Generation of random complex numbers.

RandomPrime[{i_{min}, i_{max}}]	give a pseudorandom prime in the range {i_{min}, ..., i_{max}}
RandomPrime[i_{max}]	give a pseudorandom prime in the range 2 to i_{max}
RandomPrime[*domain*, n]	give a list of n pseudorandom primes
RandomPrime[*domain*, {n_1, n_2, ...}]	give an $n_1 \times n_2 \times \ldots$ array of pseudorandom primes

Generation of random primes.

When the domain is specified in terms of x_{min} and x_{max}, `RandomReal` and `RandomInteger` generate uniformly distributed numbers over the specified range. When the domain is specified as a distribution, rules defined for the distribution are used. Additionally, mechanisms are included for defining new methods and distributions.

The two-argument interface provides a convenient way to obtain multiple random numbers at once. Even more importantly, there is a significant efficiency advantage to generating a large number of pseudorandom numbers at once.

Generating 10^7 numbers between 0 and 1 takes a fraction of a second.

```
In[9]:= Timing[RandomReal[1, 10^7];]
Out[9]= {0.791, Null}
```

Generating 10^7 numbers one at a time takes roughly five times as long.

```
In[10]:= Timing[Table[RandomReal[1], {10^7}];]
Out[10]= {2.835, Null}
```

For multidimensional arrays with dimensions n_1 through n_k, the total number of required pseudorandom numbers $n_{total} = \prod_{i=1}^{k} n_i$ is generated and then partitioned. This makes the multidimensional array generation as efficient as possible because the total number of random values is generated as efficiently as possible and the time required for partitioning is negligible.

The time required for a 100×100×100×10 array is about the same as for a vector of 10^7 numbers.

```
In[11]:= Timing[RandomInteger[100, {100, 100, 100, 10}];]
Out[11]= {0.22, Null}
```

```
In[12]:= Timing[RandomInteger[100, 10^7];]
Out[12]= {0.23, Null}
```

An array of the same dimensions generated 10 numbers at a time takes several times as long.

```
In[13]:= Timing[Table[RandomInteger[100, {10}], {100}, {100}, {100}];]
Out[13]= {0.541, Null}
```

For statistical distributions, the speed advantage of generating many numbers at once can be even greater. In addition to the efficiency benefit inherited from the uniform number generators used, many statistical distributions also benefit from vectorized evaluation of elementary and special functions. For instance, `WeibullDistribution` benefits from vector evaluations of the elementary functions `Power`, `Times`, and `Log`.

Generation of 10^5 Weibull numbers takes virtually no time.

```
In[14]:= Timing[RandomReal[WeibullDistribution[2, 1], 10^5] // Length]
Out[14]= {0.02, 100000}
```

Several seconds are required when 10^5 Weibulls are generated one at a time.

```
In[15]:= Timing[Table[RandomReal[WeibullDistribution[2, 1]], {10^5}] // Length]
Out[15]= {4.737, 100000}
```

Random number generation can be useful in exploratory investigations. For instance, you might look for occurrences of a random sequence of digits in a longer sequence of digits.

This converts a list of 5 random decimal digits to a string.

```
In[16]:= digits = Apply[StringJoin, Map[ToString, RandomInteger[9, 5]]]
Out[16]= 64141
```

The following converts the first million digits of π to a string of integers.

```
In[17]:= pistring = StringJoin[Map[ToString, RealDigits[N[Pi, 10^6]][[1]]]];
```

This gives the positions where the string of five digits appears in the first million digits of π.

```
In[18]:= StringPosition[pistring, digits]
Out[18]= {{157883, 157887}, {516599, 516603}, {883250, 883254}, {901136, 901140}}
```

Random number generation is also highly useful in estimating distributions for which closed-form results are not known or known to be computationally difficult. Properties of random matrices provide one example.

This estimates the probability that a 5×5 matrix of uniform reals will have real eigenvalues.

```
In[19]:= Block[{count = 0, ev},
          Do[ev = Eigenvalues[RandomReal[{0, 1}, {5, 5}]];
           If[Re[ev] == ev, count++], {10^5}];
          N[count / 10^5]]
Out[19]= 0.11925
```

The following does the same for a matrix of standard normal numbers.

```
In[20]:= Block[{count = 0, ev},
          Do[ev = Eigenvalues[RandomReal[NormalDistribution[0, 1], {5, 5}]];
           If[Re[ev] == ev, count++], {10^5}];
          N[count / 10^5]]
Out[20]= 0.03186
```

An example of simulating a multivariate distribution is the Gibbs sampler used in Bayesian statistics [1]. The Gibbs sampler provides a means by which to simulate values from multivariate distributions provided the distributions of each coordinate conditional on the other coordinates are known. Under some restrictions, the distribution of random vectors constructed by iteratively sampling from the conditional distributions will converge to the true multivariate distribution.

The following example will construct a Gibbs sampler for an example given by Casella and George [2]. The distribution of interest is bivariate. The conditional distribution of x given y is a binomial, and the conditional distribution of y given x is a beta. As Casella and George mention, various strategies for detecting convergence and sampling using the Gibbs sampler have been suggested. For simplicity, assume that convergence will occur within 1000 iterations. A sample of size n from the distribution will be taken as the n values following the 1000^{th} iteration. It should be noted that these n values will, however, be dependent.

This defines the sampler with a binomial and a beta conditional distribution.

```
In[21]:= sampler[len_] := Block[{y0, dist1, dist2, x0},
           y0 = .5;
           dist1[y_] := RandomInteger[BinomialDistribution[16, y]];
           dist2[x_] := RandomReal[BetaDistribution[x + 2, 16 - x + 4]];
           Do[{x0 = dist1[y0], y0 = dist2[x0]}, {1000}];
           Table[{x0 = dist1[y0], y0 = dist2[x0]}, {len}]]
```

A Gibbs sampler could also be defined as a distribution object within the distribution framework for random number generation. An example of this particular Gibbs sampler as a distribution object is provided in the section "Defining Distributions".

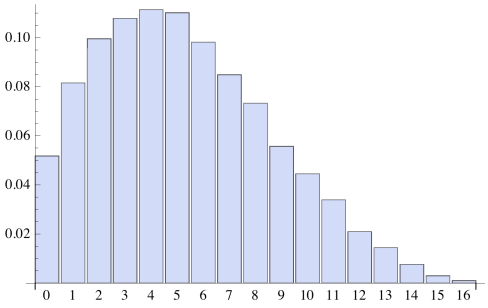
`data` is a sample of length 10^4.

```
In[22]:= data = sampler[10^4];
```

The following bar chart shows the marginal distribution of the first dimension.

In[23]:=
```
frqs = Sort[Tally[data[[All, 1]]]];
BarChart[frqs[[All, 2]] / 10^4, ChartLabels → frqs[[All, 1]]]
```

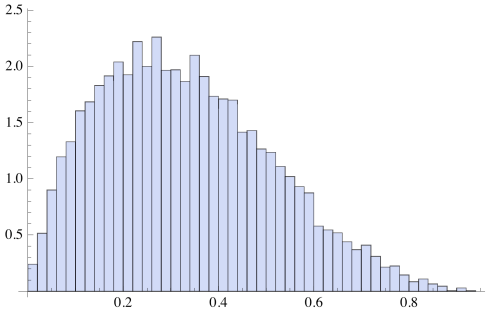
Out[24]=



The marginal distribution of the second coordinate can be visualized with a histogram.

In[25]:=
```
Histogram[data[[All, -1]], Automatic, "ProbabilityDensity"]
```

Out[25]=

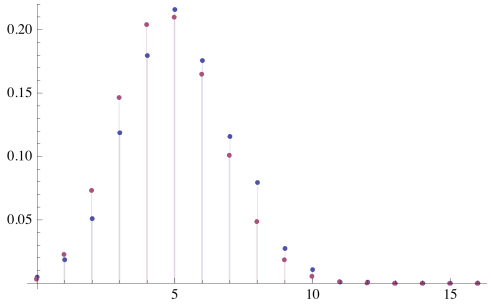


Conditional distributions should closely match the assumed binomial and beta distributions provided there is enough data for the conditional distribution. The greatest amount of data occurs when the densities of the marginal distributions are highest, so those values can be used for comparisons. The following graphics compare the empirical and assumed conditional distributions, using bins of width .05 for estimating probabilities of continuous values.

This compares the empirical and theoretical distributions of x for $0.3 \le y < 0.35$.

In[28]:=
```
Block[{y = .3, bcounts, probs, cdata},
 cdata = Select[data, y <= #[[2]] < y + .05 &][[All, 1]];
 bcounts = BinCounts[cdata, {0, 17}] / Length[cdata];
 probs = Table[{x - .025, PDF[BinomialDistribution[16, y], x]}, {x, 0, 16}];
 ListPlot[{Transpose[{Range[0, 16], bcounts}], probs}, Filling → Axis]]
```

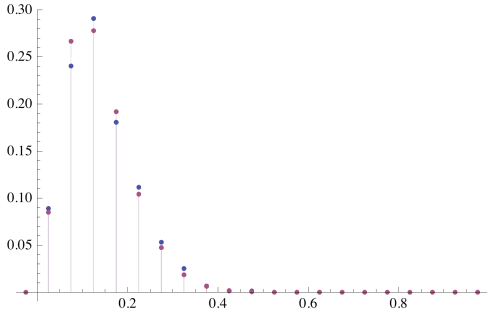
Out[28]=



This compares the empirical and theoretical distributions of y for $x = 1$.

In[29]:=
```
Block[{x = 1, bcounts, probs, cdata},
 cdata = Select[data, #[[1]] == x &][[All, -1]];
 bcounts = BinCounts[cdata, {0, 1, .05}] / Length[cdata];
 probs = Table[{y - .025, CDF[BetaDistribution[x + 2, 16 - x + 4], y] -
     CDF[BetaDistribution[x + 2, 16 - x + 4], y - .05]}, {y, 0, 1, .05}];
 ListPlot[{Transpose[{Range[.025, .975, .05], bcounts}], probs},
  Filling → Axis, PlotRange → All]]
```

Out[29]=



Arbitrary-Precision Reals and Complexes

By default, `RandomReal` and `RandomComplex` generate machine-precision numbers. Arbitrary-precision numbers can be obtained by setting the `WorkingPrecision` option.

option name	*default value*	
`WorkingPrecision`	`MachinePrecision`	precision of the arithmetic to use in calculations

Option for `RandomReal` and `RandomComplex`.

The option is valid for uniformly distributed reals, complexes, and reals from built-in distributions. WorkingPrecision can also be incorporated into user-defined distributions.

Here is a precision-25 real number between 5 and 50.

```
In[30]:= RandomReal[{5, 50}, WorkingPrecision -> 25]
Out[30]= 47.9195529823230900769 7519
```

This gives a precision-50 t-distributed number.

```
In[31]:= RandomReal[StudentTDistribution[10], WorkingPrecision -> 50]
Out[31]= 0.6365727113185606616201553815902390637883656600072 2
```

Increased WorkingPrecision can be useful in simulations where loss of precision can be expected and highly accurate results are necessary. Increased precision can also be used to estimate the precision loss in computations.

This estimates the worst precision loss in computing J_1 on the interval [0, 1000].

```
In[32]:= 100 - Precision[BesselJ[1, RandomReal[{0, 1000}, 1000, WorkingPrecision → 100]]]
Out[32]= 5.56284
```

If the precision of the input is less than the specified WorkingPrecision, the function will warn of the problem. The precision of the input will then be artificially increased to generate a pseudorandom number of the desired precision.

A warning is generated because the machine number 7.5 has precision less than 50.

```
In[33]:= RandomComplex[7.5 + I, WorkingPrecision -> 50]

RandomComplex::precw:
  The precision of the argument function ({0, 7.5 + i}) is less than WorkingPrecision (50.`). ≫

Out[33]= 6.37109205700991775983711920961705164715027122895 10 +
 0.34151554408746740924954028235202796686840490009223 i
```

WorkingPrecision is not an option for RandomInteger. Integers have infinite precision, so the precision is completely specified by the function name.

WorkingPrecision is not meaningful for pseudorandom integers.

```
In[34]:= RandomInteger[10, WorkingPrecision -> 50]

RandomInteger::array: The array dimensions WorkingPrecision → 50
    given in position 2 of RandomInteger[10, WorkingPrecision → 50] should be a
    list of non−negative machine−sized integers giving the dimensions for the result.

Out[34]= RandomInteger[10, WorkingPrecision → 50]
```

Random Elements

RandomChoice and RandomSample generate pseudorandom selections from a list of possible elements. The elements can be numeric or non-numeric.

RandomChoice[{e_1,e_2,…}]	give a pseudorandom choice of one of the e_i
RandomChoice[*list*,n]	give a list of n pseudorandom choices from *list*
RandomChoice[*list*,{n_1,n_2,…}]	give $n_1 \times n_2 \times \ldots$ pseudorandom choices from *list*
RandomChoice[{w_1,w_2,…}->{e_1,e_2,…}]	give a pseudorandom choice weighted by the w_i
RandomChoice[*wlist*->*elist*,n]	give a list of n weighted choices
RandomChoice[*wlist*->*elist*,{n_1,n_2,…}]	give an array of $n_1 \times n_2 \times \ldots$ array of weighted choices

Random choice from a list.

RandomSample[{e_1,e_2,…},n]	give a pseudorandom sample of n of the e_i
RandomSample[{w_1,w_2,…}->{e_1,e_2,…},n]	give a pseudorandom sample of n of the e_i chosen using weights w_i
RandomSample[{e_1,e_2,…}]	give a pseudorandom permutation of the e_i
RandomSample[*wlist*->*elist*]	give a pseudorandom permutation of *elist* using initial weights *wlist*

Random sample from a list.

The main difference between RandomChoice and RandomSample is that RandomChoice selects from the e_i with replacement, while RandomSample samples without replacement. The number of elements chosen by RandomChoice is not limited by the number of elements in *elist*, and an element e_i may be chosen more than once. The size of a sample returned by RandomSample is limited by the number of elements in *elist*, and the number of occurrences of a distinct element in that sample is limited by the number of occurrences of that element in *elist*.

If the first argument to RandomChoice or RandomSample is a list, elements are selected with equal probability. The weight specification defines a distribution on the set of the e_i. The

weights must be positive, but need not sum to 1. For weights $\{w_1, \ldots, w_n\}$ the probability of e_i in the initial distribution is $w_i / \sum_{j=1}^{n} w_j$. Since RandomSample samples without replacement, weights are updated internally based on the total remaining weight after each selection.

RandomChoice can be used for simulation of independent identically distributed events with a finite list of possible outcomes.

This gives 15 simulated fair coin tosses.

```
In[35]:= RandomChoice[{"heads", "tails"}, 15]
Out[35]= {heads, heads, heads, heads, tails, heads,
          tails, tails, tails, tails, heads, tails, heads, heads, tails}
```

This gives 20 rolls of a die loaded toward 5s.

```
In[36]:= RandomChoice[{.15, .1, .15, .15, .3, .15} -> Range[6], 20]
Out[36]= {1, 3, 5, 5, 5, 1, 3, 4, 5, 5, 1, 6, 3, 2, 4, 6, 6, 1, 6, 5}
```

RandomChoice can be used to generate observations from any discrete distribution with finite support.

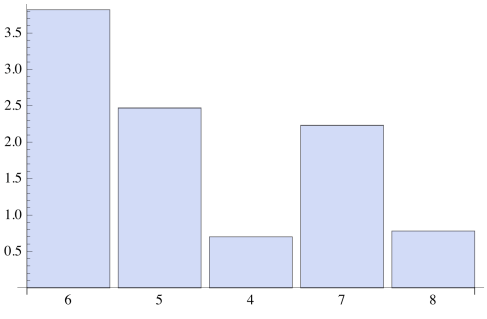
The following generates a random observation from a discrete analog of a TriangularDistribution.

```
In[37]:= RandomChoice[{1, 3, 5, 3, 1} -> Range[4, 8]]
Out[37]= 5
```

Here is the empirical PDF for 1000 simulated points.

```
In[26]:= frqs = Tally[RandomChoice[{1, 3, 5, 3, 1} -> Range[4, 8], 1000]];
         BarChart[frqs[[All, 2]] / 100, ChartLabels → frqs[[All, 1]]]
```

Out[27]=



RandomSample can be used to simulate observations from a finite set of outcomes in which each element in the list of outcomes can only be observed once. There may be more than one occurrence of distinct values in the list.

This simulates 7 draws from a container of 80 blue and 45 red objects.

```
In[41]:= RandomSample[Join[Table["blue", {80}], Table["red", {45}]], 7]
Out[41]= {blue, blue, blue, blue, blue, blue, blue}
```

Randomly sampling all elements in the list results in a random permutation.

The following is a random permutation of the integers from 1 to 10.

```
In[42]:= RandomSample[Range[10]]
Out[42]= {4, 2, 10, 1, 6, 3, 8, 9, 7, 5}
```

Assigning weights to the elements results in a random permutation in which values with greater weight tend to appear earlier in the permutation than values with lesser weight.

Here is a random permutation weighted by the squares of the data values.

```
In[43]:= RandomSample[Range[10]^2 -> Range[10]]
Out[43]= {9, 10, 6, 7, 5, 4, 8, 2, 3, 1}
```

For the same list of weighted or unweighted elements, `RandomSample[#, 1] &` is distributionally equivalent to `RandomChoice`.

This gives an empirical PDF for 10^5 random samples of size 1.

```
In[44]:= data = Range[10];

In[45]:= tallies = Tally[Table[RandomSample[data^2 -> data, 1], {10^5}]]
Out[45]= {{{3}, 2420}, {{4}, 4314}, {{10}, 26091}, {{7}, 12508},
          {{9}, 21109}, {{8}, 16509}, {{2}, 1029}, {{5}, 6336}, {{6}, 9406}, {{1}, 278}}

In[46]:= Sort[tallies][[All, 2]] / 10^5.
Out[46]= {0.00278, 0.01029, 0.0242, 0.04314, 0.06336, 0.09406, 0.12508, 0.16509, 0.21109, 0.26091}
```

Here is an empirical distribution for a distributionally equivalent `RandomChoice`.

```
In[47]:= Sort[Tally[RandomChoice[data^2 -> data, 10^5]]][[All, 2]] / 10^5.
Out[47]= {0.00258, 0.01011, 0.02324, 0.0411, 0.06343, 0.09369, 0.12762, 0.16485, 0.21333, 0.26005}
```

The probabilities for the two examples are very close to each other and to the theoretical values.

These are the theoretical probabilities.

```
In[48]:= N[data^2 / Total[data^2]]
Out[48]= {0.0025974, 0.0103896, 0.0233766, 0.0415584,
          0.0649351, 0.0935065, 0.127273, 0.166234, 0.21039, 0.25974}
```

RandomSample can also be used for random assignments to groups, such as in clinical trials. The following uses integers, but other identifying values such as name or identification number could be used instead.

The following randomly places 20 elements into four groups of equal size.

```
In[49]:= Partition[RandomSample[Range[20]], 5]
Out[49]= {{2, 11, 14, 9, 18}, {19, 10, 15, 3, 16}, {17, 7, 4, 6, 8}, {20, 1, 13, 5, 12}}
```

RandomChoice and RandomSample can be affected by changes to the Method option to SeedRandom. Built-in methods are described in "Methods". Additionally, mechanisms for defining new methods are described in "Defining Your Own Generator".

Seeding and Localization

Pseudorandom number generators algorithmically create numbers that have some apparent level of randomness. Methods for pseudorandom number generation typically use a recurrence relation to generate a number from the current state and to establish a new state from which the next number will be generated. The state can be set by seeding the generator with an integer that will be used to initialize the recurrence relation in the algorithm.

Given an initial starting point, called a seed, pseudorandom number generators are completely deterministic. In many cases it is desirable to locally or globally set the seed for a random number generator to obtain a constant sequence of "random" values. If set globally, the seed will affect future pseudorandom numbers unless a new seed is explicitly set. If set locally, the seed will only affect random number and element generation within the localized code.

BlockRandom[*expr*]	evaluate *expr* with all pseudorandom generators localized
SeedRandom[*n*]	reset the pseudorandom generator using *n* as a seed
SeedRandom[]	reset the generator using as a seed the time of day and certain attributes of the current *Mathematica* session

Localization and seeding functions.

The SeedRandom function provides a means by which to seed the random generator. Used on its own, SeedRandom will globally set the seed for random generators. The BlockRandom function provides a means by which to locally set or change the seed for random generators without affecting the global state.

The following seeds the random generator globally.

```
In[50]:= {SeedRandom[1]; RandomReal[], SeedRandom[1]; RandomReal[]}
Out[50]= {0.817389, 0.817389}
```

The following gives two different numbers because the first RandomReal is generated within BlockRandom, while the second is generated outside of BlockRandom.

The second RandomReal is not generated using the seed 1.

```
In[51]:= {BlockRandom[SeedRandom[1]; RandomReal[]],
           BlockRandom[SeedRandom[1]]; RandomReal[]}
Out[51]= {0.817389, 0.11142}
```

SeedRandom also provides the mechanism for switching the random generator.

option name	*default value*	
Method	Automatic	method to be seeded and used

Option for SeedRandom.

An individual generator can be seeded directly by specifying that generator via the Method option. All generators can be seeded by setting Method -> All.

Here the default generator is seeded with 1, but the "Rule30CA" generator is not.

```
In[52]:= BlockRandom[SeedRandom[1];
           SeedRandom[Method -> "Rule30CA"];
           RandomReal[]]
Out[52]= 0.164277
```

Seeding the "Rule30CA" generator with 1 gives a different random number.

```
In[53]:= BlockRandom[SeedRandom[1, Method -> "Rule30CA"];
           RandomReal[]]
Out[53]= 0.46345
```

Methods

Five pseudorandom generator methods are available on all systems. A sixth platform-dependent method is available on Intel-based systems. A framework for defining new methods, described in the section "Defining Your Own Generator", is also included.

"Congruential"	linear congruential generator (low-quality randomness)
"ExtendedCA"	extended cellular automaton generator (default)
"Legacy"	default generators prior to *Mathematica* 6.0
"MersenneTwister"	Mersenne Twister shift register generator
"MKL"	Intel MKL generator (Intel-based systems)
"Rule30CA"	Wolfram rule 30 generator

Built-in methods.

This gives pseudorandom integers from each method with seed 2020.

```
In[54]:= Map[BlockRandom[SeedRandom[2020, Method -> #]; RandomInteger[10^20]] &,
           {"Congruential", "ExtendedCA", "Legacy", "MersenneTwister", "MKL", "Rule30CA"}]
Out[54]= {55 649 265 348 960 921 658, 459 120 772 313 493 841, 50 876 346 696 796 959 169,
          77 391 724 740 010 742 551, 58 128 025 990 681 059 425, 74 027 343 124 503 736 203}
```

This gives pseudorandom reals from the same seed.

```
In[55]:= Map[BlockRandom[SeedRandom[2020, Method -> #]; RandomReal[]] &,
           {"Congruential", "ExtendedCA", "Legacy", "MersenneTwister", "MKL", "Rule30CA"}]
Out[55]= {0.688547, 0.00311112, 0.874893, 0.524427, 0.393891, 0.501629}
```

Congruential

"Congruential" uses a linear congruential generator. This is one of the simplest types of pseudorandom number generators, with pseudorandom numbers between 0 and 1 obtained from x_i/m, where x_i is given by the modular recurrence relation

$$x_i \equiv (b x_{i-1} + c) \bmod m$$

for some fixed integers b, c, and m called the multiplier, increment, and modulus respectively. If the increment is 0, the generator is a multiplicative congruential generator. The values of b, c, and m can be set via options to the "Congruential" method.

option name	*default value*	
"Bits"	Automatic	specify range of bits to use for numbers constructed from bits
"Multiplier"	1 283 839 219 676 404 755	multiplier value
"Increment"	0	increment value
"Modulus"	2 305 843 009 213 693 951	modulus value
"ConvertToRealsDirectly"	True	whether reals should be constructed directly from the congruence relation

Options for Method "Congruential".

Linear congruential generators are periodic and tend to give a lower quality of randomness, especially when a large number of random values is needed. If reals are generated directly from the congruence relation, the period is less than or equal to m.

The default option values are chosen to have a large period and for 64-bit efficiency. With the default options, the "Congruential" generator passes many standard tests of randomness despite the inherent issues with congruential number generators.

This generates 40 numbers from a multiplicative congruential generator.

```
In[56]:= lcdata = BlockRandom[SeedRandom[1, Method -> {"Congruential", "Multiplier" -> 11,
            "Increment" -> 0, "Modulus" -> 63}]; RandomReal[1, 40]];
```

The period of a multiplicative congruential generator is bounded above by the number of positive integers less than or equal to the modulus that are relatively prime to the modulus. This upper bound is Euler's totient function of the modulus.

With a modulus of 63, the period of the cycle is at most 36.

```
In[57]:= EulerPhi[63]
Out[57]= 36
```

The actual period can be determined by finding the smallest integer i such that $i \equiv b^i \bmod m$.

The period with multiplier 11 and modulus 63 is 6.

```
In[58]:= First[Select[Range[36], (Mod[11^#, 63] === 1 &)]]
Out[58]= 6
```

Partitioning the data into sets of 6 elements shows the recursion.

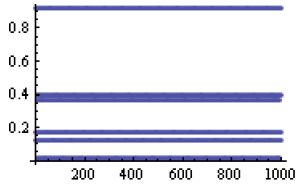
```
In[59]:= Partition[lcdata, 6]

Out[59]= {{0.174603, 0.920635, 0.126984, 0.396825, 0.365079, 0.015873},
          {0.174603, 0.920635, 0.126984, 0.396825, 0.365079, 0.015873},
          {0.174603, 0.920635, 0.126984, 0.396825, 0.365079, 0.015873},
          {0.174603, 0.920635, 0.126984, 0.396825, 0.365079, 0.015873},
          {0.174603, 0.920635, 0.126984, 0.396825, 0.365079, 0.015873},
          {0.174603, 0.920635, 0.126984, 0.396825, 0.365079, 0.015873}}
```

The distinct numbers can also be seen graphically by plotting a sequence of generated numbers.

Here is a plot of 1000 values from the congruential generator.

```
In[60]:= ListPlot[BlockRandom[SeedRandom[1, Method -> {"Congruential", "Multiplier" -> 11,
           "Increment" -> 0, "Modulus" -> 63}]; RandomReal[1, 1000]]]
```

Out[60]=



If "ConvertToRealsDirectly" is set to False, reals are generated by taking eight bits at a time from elements of the sequence to construct a 52-bit machine-precision number. Congruential numbers generated in this fashion will still cycle, but cycling will depend on repetition in the bit pattern rather than in the initial congruence relation.

The "Bits" option can be Automatic, a nonzero integer, or a list of two nonzero integers specifying the range of bits in the modulus m used for constructing numbers from bits. Automatic uses {2, -1} unless m is a power of 2, in which case {1, -1} is used.

ExtendedCA

The default "ExtendedCA" method makes use of cellular automata to generate high-quality pseudorandom numbers. This generator uses a particular five-neighbor rule, so each new cell depends on five nonadjacent cells from the previous step.

Cellular-automata-based random number generators evolve a state vector of 0s and 1s according to a deterministic rule. For a given cellular automaton, an element (or cell) at a given position in the new state vector is determined by certain neighboring cells of that cell in the old state vector. A subset of cells in the state vectors is then output as random bits from which the pseudorandom numbers are generated.

The cellular automaton used by "ExtendedCA" produces an extremely high level of randomness. It is so high that even using every single cell in output will give a stream of bits that passes many randomness tests, in spite of the obvious correlation between one cell and five previous ones.

Two options are included for modifying the size of the state vector and the cells skipped. The defaults are chosen for quality and speed and there is typically no need to modify these options.

option name	*default value*	
"Size"	80	state vector size as a multiplier of 64
"Skip"	4	number of cells to skip

Options for Method "ExtendedCA".

The length of the state vectors used is by default set to 80×64=5120 cells. The multiple of 64 can be controlled by the "Size" option.

In practice using every fourth cell in each state vector proves to be sufficient to pass very stringent randomness tests. This is the default used for the "Skip" option. For even faster random number generation, a "Skip" setting of 2 or even 1 could be used, but the quality of the random numbers will then decline.

"ExtendedCA" is the default number generator.

```
In[61]:= BlockRandom[SeedRandom[1]; RandomReal[1, 5]]
Out[61]= {0.817389, 0.11142, 0.789526, 0.187803, 0.241361}
```

```
In[62]:= BlockRandom[SeedRandom[1, Method -> "ExtendedCA"]; RandomReal[1, 5]]
Out[62]= {0.817389, 0.11142, 0.789526, 0.187803, 0.241361}
```

Legacy

The "Legacy" method uses the generator called by Random in versions of *Mathematica* prior to Version 6.0. A Marsaglia-Zaman subtract-with-borrow generator is used for reals. The integer generator is based on a Wolfram rule 30 cellular automaton generator. The rule 30 generator is used directly for small integers and used to generate certain bits for large integers.

Here are RandomReal and RandomInteger values obtained via the "Legacy" method.

```
In[63]:= BlockRandom[SeedRandom[31, Method -> "Legacy"]; {RandomReal[], RandomInteger[50]}]
Out[63]= {0.210596, 8}
```

The same values are given by equivalent Random calls.

```
In[64]:= BlockRandom[SeedRandom[31]; {Random[], Random[Integer, {0, 50}]}]
Out[64]= {0.210596, 8}
```

To guarantee consistency with sequences generated prior to Version 6.0, seeds set for the Automatic method are also applied to the "Legacy" method.

The "Legacy" method has no options.

MersenneTwister

"MersenneTwister" uses the Mersenne Twister generator due to Matsumoto and Nishimura [3][4]. The Mersenne Twister is a generalized feedback shift register generator with period $2^{19937} - 1$.

This gives 5 random numbers from a Mersenne Twister generator.

```
In[65]:= BlockRandom[SeedRandom[1, Method -> "MersenneTwister"]; RandomReal[1, 5]]
Out[65]= {0.393562, 0.701033, 0.966231, 0.221456, 0.436768}
```

The "MersenneTwister" method has no options.

MKL

The "MKL" method uses the random number generators provided in Intel's MKL libraries. The MKL libraries are platform dependent. The "MKL" method is available on Microsoft Windows (32-bit, 64-bit), Linux x86 (32-bit, 64-bit), and Linux Itanium systems.

option name	*default value*	
Method	Automatic	MKL generator to use

Option for Method "MKL".

`"MCG31"`	31-bit multiplicative congruential generator
`"MCG59"`	59-bit multiplicative congruential generator
`"MRG32K3A"`	combined multiple recursive generators with two components of order 3
`"MersenneTwister"`	Mersenne Twister shift register generator
`"R250"`	generalized feedback shift register generator
`"WichmannHill"`	Wichmann-Hill combined multiplicative congruential generators
`"Niederreiter"`	Niederreiter low-discrepancy sequence
`"Sobol"`	Sobol low-discrepancy sequence

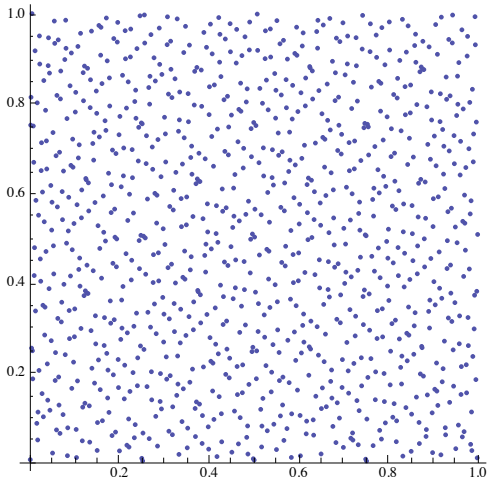
`"MKL"` methods.

The first six methods are uniform generators. `"Niederreiter"` and `"Sobol"` generate Niederreiter and Sobol sequences. These sequences are nonuniform and have underlying structure which is sometimes useful in numerical methods. For instance, these sequences typically provide faster convergence in multidimensional Monte Carlo integration.

The following shows the structure of a Niederreiter sequence in dimension 2.

In[66]:=
```
ListPlot[BlockRandom[
  SeedRandom[Method -> {"MKL", Method -> {"Niederreiter", "Dimension" -> 2}}];
  RandomReal[1, {1000, 2}]],
 AspectRatio -> 1]
```

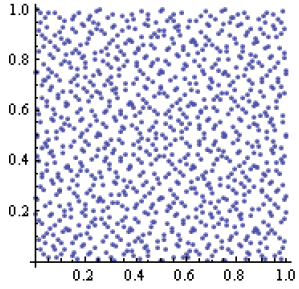
Out[66]=



This shows the structure of a Sobol sequence in dimension 2.

```
In[67]:= ListPlot[BlockRandom[
           SeedRandom[Method -> {"MKL", Method -> {"Sobol", "Dimension" -> 2}}];
           RandomReal[1, {1000, 2}]],
         AspectRatio -> 1]
```

Out[67]=



Rule30CA

The `"Rule30CA"` method uses a Wolfram rule 30 cellular automaton generator. Bits are obtained by evolving a state vector of 0s and 1s using the relation

$$f(i,\ t+1) = f(i-1,\ t) \veebar (f(i,\ t) \vee f(i+1,\ t)),$$

where $f(i, t)$ is the value of cell i at time t.

option name	*default value*	
`"Size"`	9	state vector size as a multiplier of 29

Option for `Method` `"Rule30CA"`.

The length of the state vectors used is by default set to $9 \times 29 = 261$ cells. The multiplier for 29 can be controlled by the `"Size"` option.

This gives a 2×3×4 tensor of random integers using `"Rule30CA"`.

```
In[68]:= BlockRandom[SeedRandom[1, Method -> "Rule30CA"]; RandomInteger[10^5, {2, 3, 4}]]
Out[68]= {{{60 745, 40 991, 22 336, 76 623}, {57 042, 92 146, 91 746, 18 972}, {70 251, 62 829, 78 488, 82 331}},
          {{93 628, 53 079, 11 476, 55 013}, {7702, 96 543, 99 411, 79 327}, {42 137, 83 772, 56 154, 6410}}}
```

The `"Rule30CA"` method uses only the first bit from each state vector, making it slower than the `"ExtendedCA"` method, which uses multiple bits from each state vector.

Defining Your Own Generator

Methods can be plugged into the random framework as long as they follow the correct template. A generator object is of the form $gsym[data]$ where $gsym$ is the symbol that identifies the generator and to which rules are attached. $data$ is effectively private to the top-level evaluations associated with the generator definitions.

Generator initialization is handled by a call to `Random`InitializeGenerator`.

`Random`InitializeGenerator[`*gsym*`,`*opts*`]`	initializes the generator *gsym* with options *opts*

Generator initialization function.

`Random`InitializeGenerator` is expected to return a generator object *gobj* of the form $gsym[data]$.

Generators can support generation of random bit streams, random integers, and random reals. If the generator supports bit streams, reals and integers can be generated by conversion of the bit stream. At method setup time, properties are queried to determine what is supported and how.

`GeneratesBitsQ`	set to `True` if the method generates bits
`GeneratesIntegersQ`	set to `True` if the method generates integers for a given range
`GeneratesRealsQ`	set to `True` if the method generates reals for a given range and precision

Generator properties.

If bit streams are supported, then $gobj$`["GenerateBits"[`$nbits$`]]` is expected to return an integer comprised of n random bits or a list of length $nbits$ with entries that are 0 or 1.

If random integers are supported, then $gobj$`["GenerateIntegers"[`n`, {`a`, `b`}]]` is expected to return a list of n random integers in the range $[a, b]$. A warning message will be issued when results are out of range.

If random reals are supported, then $gobj$`["GenerateReals"[`a`, {`a`, `b`}, `$prec$`]]` is expected to return a list of n random reals with precision $prec$ in the range $[a, b]$. A warning message will be issued when results are out of range or of the wrong precision.

For any of the generation functions, the return can be {*res*, *gobj*}, where *res* is the result of the correct type and *gobj* is a new generator object (reflecting any state change).

Seeding is done by *gobj*["SeedGenerator"[*seed*]] for an integer *seed*. *gobj*["SeedGenerator"[*seed*]] is expected to return a new generator object.

Example: Multiplicative Congruential Generator

In the following example a multiplicative congruential generator will be defined. A multiplicative congruential generator follows the recurrence relation

$$x_i \equiv b x_{i-1} \bmod m.$$

The generator, as defined below, will allow only for generation of real numbers.

This sets default options for the generator MultiplicativeCongruential.

```
In[69]:= Options[MultiplicativeCongruential] =
           {"Multiplier" → 123 456 789, "Modulus" → 2^35 - 1};
```

Initialization of the generator will extract the values of the multiplier and modulus. Initialization will fail if either of these values is not a positive integer.

The following initializes the generator.

```
In[70]:= MultiplicativeCongruential /:
           Random`InitializeGenerator[MultiplicativeCongruential, opts___] := Module[
            {mult, mod, flops = Flatten[{opts, Options[MultiplicativeCongruential]}]},
            mult = "Multiplier" /. flops;
            If[! (IntegerQ[mult] && Positive[mult]),
             Throw[$Failed]
            ];
            mod = "Modulus" /. flops;
            If[! (IntegerQ[mod] && Positive[mult]),
             Throw[$Failed]
            ];
            MultiplicativeCongruential[mult, mod, 1]];
```

Calls from the kernel to Random`IntializeGenerator are effectively wrapped in Catch. Throw can be used in the initialization code to easily exit in case of problems.

This establishes that MultiplicativeCongruential generates reals.

```
In[71]:= MultiplicativeCongruential[___]["GeneratesRealsQ"] := True;
```

The following seeds the generator using the recurrence relation.

```
In[72]:= MultiplicativeCongruential[mult_, mod_, ___]["SeedGenerator"[seed_]] :=
           MultiplicativeCongruential[mult, mod, Mod[(mult * seed), mod]];
```

The real number generator will return the desired number of reals n and a new `MultiplicativeCongruential` generator. The seed for the new generator is updated based on the recurrence relation.

This defines the real number generator.

```
In[73]:= MultiplicativeCongruential[mult_, mod_, s_][
           "GenerateReals"[n_, {a_, b_}, prec_]] :=
          Module[{x = s},
           {a + (b - a) Table[x = mult * x; Mod[x, mod], {n}] / mod,
            MultiplicativeCongruential[mult, mod, x]}
          ]
```

This generates 10 reals using the `MultiplicativeCongruential` generator.

```
In[74]:= BlockRandom[
          SeedRandom[Method -> MultiplicativeCongruential];
          RandomReal[{5, 50}, 10]]
Out[74]= {33.1547, 47.5694, 45.0011, 31.632, 25.1043, 37.7568, 16.2839, 48.1744, 17.5352, 48.7686}
```

The generator is not defined for integers.

```
In[75]:= BlockRandom[
          SeedRandom[Method -> MultiplicativeCongruential];
          RandomInteger[{5, 50}]]

         RandomInteger::unstyp :
           The current random generator does not support generation of random integers in the given range.

Out[75]= RandomInteger[{5, 50}]
```

Example: Blum-Blum-Shub Generator

The Blum-Blum-Shub generator is a quadratic congruential method for generating pseudorandom bits for cryptographic purposes [5]. The congruence is mod $p \times q$ for specified primes p and q.

This sets default options for the generator `BlumBlumShub`.

```
In[76]:= Options[BlumBlumShub] = {"BlumPrimes" → {1 267 650 600 228 229 401 496 703 981 519,
            1 267 650 600 228 229 401 496 704 318 359}, "BitWidth" → Automatic};
```

The following define an auxiliary function and error messages for the generator.

```
In[77]:= SpecialBlumPrimeQ[x_] := (Positive[x] && (Mod[x, 4] == 3) && PrimeQ[x])

In[78]:= BlumBlumShub::bprime = "`1` is not a list of two distinct special Blum primes.";

In[79]:= BlumBlumShub::bw =
           "Warning: the value of the option BitWidth->`1` exceeds the number
             `2` that has been proved cyptographically secure.";

In[80]:= BlumBlumShub::bw1 = "The value of the option BitWidth->`1`
             should be a positive machine-sized integer or Automatic.";
```

The generator initialization will extract option values and issue error messages if necessary before calling the actual generator.

The following initializes the generator.

```
In[81]:= BlumBlumShub /: Random`InitializeGenerator[BlumBlumShub, opts___] :=
           Module[{n, abw, bw, flops = Flatten[{opts, Options[BlumBlumShub]}]},
            n = "BlumPrimes" /. flops;
            If[
             ! And[VectorQ[n, SpecialBlumPrimeQ], Length[n] == 2, Not[Apply[Equal, n]]],
             Message[BlumBlumShub::"bprime", n];
             Throw[$Failed]
            ];
            n = Apply[Times, n];
            abw = Max[1, Floor[Log[2., Log[2., n]]]];
            bw = "BitWidth" /. flops;
            If[bw === Automatic,
             bw = abw,
             If[! (IntegerQ[bw] && Positive[bw]),
              Message[BlumBlumShub::bwi, bw];
              Throw[$Failed]];
             If[bw > abw,
              Message[BlumBlumShub::bw, bw, abw]];
            ];
            BlumBlumShub[n, bw, 2^bw - 1, 2]];
```

This establishes that `BlumBlumShub` is a bit generator and determines the bit width.

```
In[82]:= BlumBlumShub[___]["GeneratesBitsQ"] := True;
```

```
In[83]:= BlumBlumShub[n_, bw_, __]["BitWidth"] := bw;
```

The following seeds the generator.

```
In[84]:= BlumBlumShub[n_, bw_, mask_, ___]["SeedGenerator"[seed_]] :=
          Module[{x , i = 0, state = {}},
           While[Length[Union[state]] < 10,
            x = seed + i;
            state = NestList[PowerMod[#, 2, n] &, x, 9];
           ];
           BlumBlumShub[n, bw, mask,  Last[state]]]
```

This defines the bit generator.

```
In[85]:= BlumBlumShub[n_, bw_, mask_, s_]["GenerateBits"[bits_]] :=
          Module[{x = PowerMod[s, 2, n]},
           {BitAnd[x, mask], BlumBlumShub[n, bw, mask, x]}]
```

This generates 5 integers and 5 reals using the `BlumBlumShub` generator.

```
In[86]:= BlockRandom[
          SeedRandom[Method -> BlumBlumShub];
          {RandomInteger[{0, 10}, 5], RandomReal[4, 5]}]
Out[86]= {{1, 5, 7, 3, 5}, {2.37406, 1.59922, 1.11636, 3.70079, 0.29338}}
```

Statistical Distributions

The general idea behind generating random variates from a nonuniform statistical distribution is to generate a random uniform variate between 0 and 1 and then compute the inverse CDF of that random value in the desired distribution. In practice, however, following this recipe directly can be very computationally intensive if a large number of random variates is desired, particularly when the inverse CDF is complicated or cannot be expressed in a closed form.

In such cases, table lookups, direct construction based on distributional relationships, or acceptance-rejection methods are often more efficient alternatives to direct inversion of the CDF. On some level, these methodologies will all still rely on uniformly distributed `RandomReal` values, uniformly distributed `RandomInteger` values, observations from a weighted `RandomChoice`, or a combination of these values. As a result, methods set via `SeedRandom` will have an effect on random observations from statistical distributions.

The methods used by `RandomReal` and `RandomInteger` for many of the distributions in *Mathematica* follow methods suggested or described in Gentle [6], and are not necessarily the same methods used by `Random` and `RandomArray` in the standard add-ons included with versions of *Mathematica* prior to Version 6.0.

Random observations from all built-in statistical distributions can be generated using either `RandomReal` or `RandomInteger`.

`RandomReal[`*dist*`]`	give a random number from the continuous distribution *dist*
`RandomReal[`*dist*`,`*n*`]`	give a list of *n* pseudorandom reals from *dist*
`RandomReal[`*dist*`,{`n_1`,`n_2`,…}]`	give an $n_1 \times n_2 \times \ldots$ array of pseudorandom reals from *dist*

Generation of random values from continuous distributions.

`RandomInteger[`*dist*`]`	give a random number from the discrete distribution *dist*
`RandomInteger[`*dist*`,`*n*`]`	give a list of *n* pseudorandom integers from *dist*
`RandomInteger[`*dist*`,{`n_1`,`n_2`,…}]`	give an $n_1 \times n_2 \times \ldots$ array of pseudorandom integers from *dist*

Generation of random values from discrete distributions.

Observations from continuous univariate and multivariate distributions are obtained via `RandomReal`, while discrete univariate and multivariate distributions are obtained via `RandomInteger`. If `RandomInteger` is used on a continuous distribution, or `RandomReal` on a discrete distribution, an error is generated and the input is returned unevaluated.

This fails because the χ^2 distribution is continuous.

```
In[87]:= RandomInteger[ChiSquareDistribution[8]]

RandomInteger::unsdst:
  The distribution ChiSquareDistribution[8] is defined on a set of real values. Use RandomReal instead.

Out[87]= RandomInteger[ChiSquareDistribution[8]]
```

`WorkingPrecision` is an option to `RandomReal` for continuous distributions just as it is for uniform numbers over ranges.

Here is a precision-30 beta-distributed variate.

```
In[88]:= RandomReal[BetaDistribution[3, 7], WorkingPrecision -> 30]
Out[88]= 0.453568522862231707895720399466
```

Multivariate distributions and distributions derived from the multivariate normal distribution are included in the Multivariate Statistics Package. Generators for those distributions can be accessed by loading the package.

This loads the package.

```
In[89]:= Needs["MultivariateStatistics`"]
```

Here is a random vector from a bivariate normal distribution.

```
In[90]:= RandomReal[MultinormalDistribution[{1, 2}, {{2, .5}, {.5, 3}}]]
Out[90]= {1.8453, 3.59417}
```

This is a random vector from a multinomial distribution.

```
In[91]:= RandomInteger[MultinomialDistribution[20, {1 / 3, 1 / 2, 1 / 6}]]
Out[91]= {8, 9, 3}
```

Continuous Distributions

For distributions whose inverse CDFs contain only elementary functions, direct computation of the inverse CDF for a random uniform is generally used. This can be seen as a direct construction from a uniformly distributed random variable. Continuous distributions falling in this category include `CauchyDistribution`, `ExponentialDistribution`, `ExtremeValueDistribution`, `GumbelDistribution`, `LaplaceDistribution`, `LogisticDistribution`, `ParetoDistribution`, `RayleighDistribution`, `TriangularDistribution`, and `WeibullDistribution`.

Direct construction of a single random variate from multiple uniform variates, or from variates other than the uniform distribution are also employed. Normal variates are generated in pairs from pairs of random uniforms using the Box-Müller method. `HalfNormalDistribution` and `LogNormalDistribution` variates are obtained by direct transformation of normal variates. `MultinormalDistribution` and `QuadraticFormDistribution` from the Multivariate Statistics Package also use direct construction from normal variates.

`InverseGaussianDistribution` uses an acceptance-complement method involving normal and uniform variates. The method is due to Michael, Schucany and Haas and described in Gentle [6]. `MaxwellDistribution` variates are constructed from `ChiDistribution` variates. The chi variates themselves are obtained from `ChiSquareDistribution` variates, which are special cases of `GammaDistribution` variates.

In most cases `FRatioDistribution` constructs each random value from a single random beta variate. For small degrees of freedom, `FRatioDistribution` variates are instead generated from pairs of gamma variates to avoid possible divisions by 0 that may arise in the beta construction.

`NoncentralChiSquareDistribution[`ν`, `λ`]`, $\chi^2_\nu(\lambda)$, variate generation uses additive properties of χ^2 distributions to avoid expensive inverse CDF computations for nonintegral ν. The additive properties are given in, for instance, Johnson, Kotz, and Balakrishnan [7]. For $\nu = 1$ a noncentral χ^2 variate can be generated as the square of a normal variate with mean $\sqrt{\lambda}$ and variance 1. For $\nu \neq 1$ noncentral χ^2 variates are obtained as the sum of a central and a noncentral χ^2 random variable. For $\nu > 1$, $X + Y$ is distributed $\chi^2_\nu(\lambda)$ if $X \sim \chi^2_1(\lambda)$ and $Y \sim \chi^2_{\nu-1}$. This relationship cannot be used for $\nu < 1$. In that case the construction is $X + Y$ with $X \sim \chi^2_0(\lambda)$ and $Y \sim \chi^2_{\nu-1}$, where $\chi^2_0(\lambda)$ is the

limiting noncentral χ^2 distribution as ν goes to 0. The limiting distribution $\chi_0^2(\lambda)$ is a mixture of Poisson and χ^2 variables, which has a nonzero probability mass at 0 and a continuous density for positive values. `NoncentralFRatioDistribution` variates are obtained from one central and one noncentral χ^2 variate.

For the `WishartDistribution` from the Multivariate Statistics Package, matrices are generated via Smith and Hocking's method [8]. This method constructs Wishart matrices from matrices with chi-distributed diagonal entries and normally distributed off-diagonal entries.

`NoncentralStudentTDistribution`, and the `HotellingTSquareDistribution` and `MultivariateTDistribution` from the Multivariate Statistics Package each use direct construction from univariate random variates.

`GammaDistribution`, `BetaDistribution`, and `StudentTDistribution` use acceptance-rejection methods to some extent.

For `GammaDistribution[`α`,` β`]` exponential variates are generated when $\alpha = 1$. Otherwise, methods due to Cheng and Feast [9] and Ahrens and Dieter [10] are used.

Beta variates are constructed by switching between multiple methods depending on the values of the beta parameters α and β. If both parameters are 1, uniform random variates will be generated. If one of the beta parameters is 1, then a closed-form inverse CDF evaluation is used. Otherwise, `RandomReal` switches between acceptance-rejection methods due to Jöhnk [11], Cheng [12], and Atkinson [13]. An example of the advantage of using an acceptance-rejection method over construction from two gammas can be seen in the following. The direct acceptance-rejection method is nearly twice as fast as the gamma-pair construction.

This shows a comparison of direct construction and acceptance-rejection methods for beta variates.

```
In[92]:= Timing[With[{x1 = RandomReal[GammaDistribution[7, 1], 10^6]},
           x1 / (x1 + RandomReal[GammaDistribution[3, 1], 10^6])];]
Out[92]= {1.422, Null}

In[93]:= Timing[RandomReal[BetaDistribution[7, 3], 10^6];]
Out[93]= {1.021, Null}
```

For `StudentTDistribution` the method used by `RandomReal` is a polar rejection method due to Bailey [14]. This method is more efficient than direct construction from normal and χ^2 variates as can be seen in the following. The direct construction takes roughly 1.5 times as long as the polar method for a million Student t variates.

This shows a comparison of direct construction and Bailey's polar rejection method for Student t.

```
In[94]:= Timing[RandomReal[NormalDistribution[0, 1], 10^6] /
           Sqrt[RandomReal[ChiSquareDistribution[6], 10^6] / 6];]
Out[94]= {1.282, Null}

In[95]:= Timing[RandomReal[StudentTDistribution[6], 10^6];]
Out[95]= {0.611, Null}
```

Discrete Distributions

`GeometricDistribution`, `BetaBinomialDistribution`, and `BetaNegativeBinomialDistribution` use direct construction. `GeometricDistribution` variates are generated as $\left\lfloor \frac{\log(U)}{\log(1-p)} \right\rfloor$ where U follows `UniformDistribution[0, 1]`. `BetaBinomialDistribution` and `BetaNegativeBinomialDistribution` are constructed from `BinomialDistribution` and `NegativeBinomialDistribution` variates with probability parameters taken as random `BetaDistribution` variates.

When used, table lookups for random integer generation are implemented via `RandomChoice` using the distribution's probability mass function for the weights. Most discrete distributions switch to other methods whenever construction of the list of weights is expected to be expensive given the desired sampled size. For example, as p approaches 1 `LogSeriesDistribution[p]` switches to the direct construction $\left\lfloor 1 + \frac{\log(V)}{\log\left(1-(1-p)^U\right)} \right\rfloor$, where U and V are uniformly distributed on the interval $[0, 1]$ [15]. Depending on parameters and sample size `NegativeBinomialDistribution[n, p]` may switch to construction as a Poisson-gamma mixture, which is a Poisson variable with mean following a gamma distribution [6].

`BinomialDistribution`, `HypergeometricDistribution`, and `PoissonDistribution` rely on direct sampling from the density function if the computational overhead of computing the PDF values is small relative to the number of desired random values. Otherwise they switch to acceptance-rejection methods. The acceptance-rejection methods also allow for generation of variates when overflows or underflows would occur in directly computing the PDF values, thus extending the range of parameter values for which random numbers can be generated.

The binomial and hypergeometric distributions switch to acceptance-rejection methods due to Kachitvichyanukul and Schmeiser with small modifications. The binomial method, based on the acceptance-rejection portion of their BTPE (Binomial, Triangle, Parallelogram, Exponential) algorithm [16], effectively uses a piecewise majorizing function with three regions and a triangular minorizing function for a quick acceptance test. The majorizing and minorizing functions create a two-parallelogram envelope around the center of the rescaled binomial density, and the tails of the majorizing function form exponential envelopes on the tails of the scaled binomial distribution. One case where it is clearly better to use BTPE rather than to construct a lookup table is when few observations are desired and the lookup table would be large.

The hypergeometric method, based on the acceptance-rejection portion of Kachitvichyanukul and Schmeiser's H2PE algorithm [17], uses a majorizing function with three regions around a scaled hypergeometric density. The middle portion of the density is enveloped by a rectangular region and the tails of the distribution are bounded by exponentials.

The acceptance-rejection method used by `PoissonDistribution` is due to Ahrens and Dieter [18]. The acceptance and rejection is carried out using discrete normal variates, taking advantage of the tendency of `PoissonDistribution[`μ`]` toward `NormalDistribution[`μ`,` $\sqrt{\mu}$`]` as μ increases.

Random values from the `ZipfDistribution` are generated via an acceptance-rejection method described by Devroye [15]. The method uses pairs of uniform variates and a test involving only a `Floor` and noninteger powers, aside from basic arithmetic, to efficiently obtain Zipf-distributed values.

Defining Distributions

Definitions for distributions are supported through rules for `Random`DistributionVector`. `DistributionVector` is expected to return a vector of the given length with numbers of the given precision.

`Random`DistributionVector[`*dist*`,`*n*`,`*prec*`]`
: defines rules for generating n observations from *dist* with precision *prec*

Function for defining random generation from distributions.

Rules for generating random values from distributions are generally defined via a `TagSet` on the head of the distribution. The distribution itself may contain parameters. As a simple example, the following defines rules for `NegativeOfUniform[a, b]`, which represents a uniform distribution on the interval $[-b, -a]$.

```
In[96]:= NegativeOfUniform /:
          Random`DistributionVector[NegativeOfUniform[a_, b_], n_Integer,
           prec_?Positive] := -RandomReal[{a, b}, n, WorkingPrecision -> prec] /;
           VectorQ[{a, b}, NumericQ] && Element[{a, b}, Reals]
```

Random numbers from `NegativeOfUniform` can now be generated via `RandomReal` just like any built-in continuous distribution.

The following gives a machine-precision number and a precision-20 number from `NegativeOfUniform`.

```
In[97]:= {RandomReal[NegativeOfUniform[1, 3]],
          RandomReal[NegativeOfUniform[1, 3], WorkingPrecision -> 20]}
Out[97]= {-2.90662, -2.0231113301407817182}
```

Matrices and higher-dimensional tensors can also be generated directly via `RandomReal`. `RandomReal` uses the definition given to `Random`DistributionVector` to generate the total number of random values desired, and partitions that total number into the specified dimensions.

Here is a 3×4 array of `NegativeOfUniform` numbers.

```
In[98]:= RandomReal[NegativeOfUniform[7, 21], {3, 4}]
Out[98]= {{-7.44952, -7.12576, -15.6616, -13.9985},
          {-18.3775, -11.6597, -17.3955, -15.4164}, {-13.2334, -17.4337, -20.5442, -15.0836}}
```

Discrete distributions can be defined in a similar way. The main difference is that the precision argument to `Random`DistributionVector` will now be `Infinity`. The discrete version of `NegativeOfUniform` provides a simple example.

```
In[99]:= NegativeOfDiscreteUniform /:
          Random`DistributionVector[NegativeOfDiscreteUniform[a_Integer, b_Integer],
           n_Integer, Infinity] := -RandomInteger[{a, b}, n]
```

Random values from `NegativeOfDiscreteUniform` can now be obtained from `RandomInteger`.

Here are 10 `NegativeOfDiscreteUniform` numbers.

```
In[100]:= RandomInteger[NegativeOfDiscreteUniform[1, 3], 10]
Out[100]= {-2, -3, -2, -3, -3, -3, -1, -3, -3, -1}
```

While the previous examples show the basic framework for defining distributions, the distributions themselves are not particularly interesting. In fact, it would have been easier in these two cases to just generate values from `RandomReal` and `RandomInteger` and multiply the end result by -1 instead of attaching definitions to a new distribution symbol. The following examples will demonstrate slightly more complicated distributions, in which case attaching definitions to a distribution will be more useful.

Example: Normal Distribution by Inversion

The textbook definition for generating random values from a generic univariate statistical distribution involves two steps:

- generate a uniform random number q on the interval $[0, 1]$
- compute the inverse cumulative distribution function of q for the distribution of interest

To demonstrate the process, use the normal distribution. To generate random normal variates using this method, start with the `Quantile` for the normal distribution at a point q and replace q with a random uniform between 0 and 1.

Here is the `Quantile` function for a normal with mean μ and standard deviation σ.

```
In[101]:= Quantile[NormalDistribution[μ, σ], q]
Out[101]= μ + √2 σ InverseErf[-1 + 2 q]
```

A new distribution object can now be used to define a normal random number generator that uses inversion of the cumulative distribution function.

This defines generation of normals by inversion.

```
In[102]:= NormalByInversion /:
           Random`DistributionVector[
            NormalByInversion[mu_?(NumericQ[#] && Im[#] === 0 &), sigma_?Positive],
            n_Integer, prec_?Positive] :=
           mu + √2 sigma InverseErf[-1 + 2 RandomReal[1, n, WorkingPrecision -> prec]]
```

Here are 10 random normals generated by inversion.

```
In[103]:= RandomReal[NormalByInversion[1, 2], 10]
Out[103]= {-0.457637, -1.59854, 1.53492, 1.85251, 5.06223, -0.765828, -2.86778, -0.512359, 1.55514, 4.64143}
```

Here is a sample of 10^4 random normals along with the sample mean and standard deviation.

```
In[104]:= (ninv = RandomReal[NormalByInversion[1, 2], 10^4]); // Timing
Out[104]= {3.385, Null}
```

```
In[105]:= {Mean[ninv], StandardDeviation[ninv]}
Out[105]= {0.989365, 2.01145}
```

The normal distribution is one example where methods other than direct inversion are generally preferred. While inversion of the CDF is a perfectly valid method for generating pseudorandom normal numbers, it is not particularly efficient. Numeric evaluation of `InverseErf` is computationally much more expensive than the sinusoid and logarithmic evaluations required by the Box-Müller method used by `NormalDistribution`.

The built-in method takes almost no time for the same number of values.

```
In[106]:= (ndist = RandomReal[NormalDistribution[1, 2], 10^4]); // Timing
Out[106]= {0., Null}

In[107]:= {Mean[ndist], StandardDeviation[ndist]}
Out[107]= {1.02922, 1.99552}

In[108]:= Clear[ninv, ndist]
```

Example: Uniform Distribution on a Disk

`Random`DistributionVector` can also be used to define generators for multidimensional distributions. For instance, suppose a random point from a uniform distribution on the unit disk, the set of real points $\{x, y\}$ with $\sqrt{x^2 + y^2} \leq 1$, is desired. Such a random point can be constructed as follows:

- generate a random angle ϕ uniformly distributed on $[0, 2\pi)$
- generate a random vector u uniformly distributed on $[0, 1]$
- return $\left\{\sqrt{u}\ \sin\phi, \sqrt{u}\ \cos\phi\right\}$

The returned ordered pair can be multiplied by r to generate points uniformly distributed on a disk of radius r.

The following defines a generator for a uniform disk of radius r.

```
In[109]:= UniformDisk /:
           Random`DistributionVector[
            UniformDisk[r_?Positive], n_Integer, prec_?Positive] :=
           r * Sqrt[RandomReal[1, n, WorkingPrecision → prec]] *
            Transpose[{Cos[#], Sin[#]} &[RandomReal[2 Pi, n, WorkingPrecision → prec]]]
```

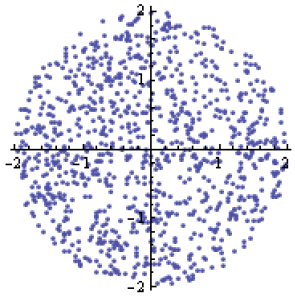
Here is one random ordered pair from a disk of radius 2.

```
In[110]:= RandomReal[UniformDisk[2]]
Out[110]= {0.232325, 0.660046}
```

The following visualizes the distribution of 10^4 generated points on this disk.

```
In[111]:= ListPlot[RandomReal[UniformDisk[2], 1000], AspectRatio → 1]
```

Out[111]=



Example: Dirichlet Distribution

The n-dimensional Dirichlet distribution is parameterized by a vector of positive values $\{\alpha_1, \ldots, \alpha_n\}$ and has support on the set of vectors $\{x_1, \ldots, x_n\}$ such that $\sum_{i=1}^{n} x_i = 1$ and $x_i \in [0, 1]$ for i from 1 to n. Thus, the Dirichlet distribution is defined on an $n-1$-dimensional subspace of the n-dimensional unit hypercube $[0, 1]^n$. The Dirichlet distribution is the multivariate extension of the beta distribution. If y follows `BetaDistribution[α, β]`, then $\{y, 1-y\}$ follows a Dirichlet distribution with parameter vector $\{\alpha, \beta\}$.

An n-dimensional Dirichlet variate can be generated from n gamma variates. With parameter vector $\{\alpha_1, \ldots, \alpha_n\}$, the process is as follows:

- generate a random number γ_i from `GammaDistribution[α_i, 1]` for i from 1 to n
- return $\{\gamma_1, \ldots, \gamma_n\} / \sum_{i=1}^{n} \gamma_i$

This defines a Dirichlet generator attached to the symbol `DirichletDistribution`.

```
In[112]:= DirichletDistribution /: Random`DistributionVector[
            DirichletDistribution[alpha_?(VectorQ[#, Positive] &)],
            n_Integer, prec_?Positive] :=
           Block[{gammas},
            gammas =
             Map[RandomReal[GammaDistribution[#, 1], n, WorkingPrecision → prec] &, alpha];
            Transpose[gammas] / Total[gammas]]
```

Here is a three-dimensional Dirichlet vector with precision 25.

```
In[113]:= RandomReal[DirichletDistribution[{1, 3, 5 / 2}], WorkingPrecision -> 25]
Out[113]= {0.018396041373213696377711 29, 0.699794890174493389846 0560, 0.281809068452229137762327}
```

Example: Gibbs Sampler

Gibbs samplers can also be defined as distributions. As an example consider a Gibbs sampler that mixes beta and binomial distributions. A specific case of this sampler was explored in a previous example. Here, the distribution will be defined with two parameters m and α.

This defines a Gibbs sampler `BinomialBetaSampler`.

```
In[114]:= BinomialBetaSampler /: Random`DistributionVector[
            BinomialBetaSampler[m_Integer, α_?Positive], n_Integer, prec_?Positive] :=
           Block[{y0, dist1, dist2, x0},
            y0 = .5;
            dist1[y_] := RandomInteger[BinomialDistribution[m, y]]; dist2[x_] :=
             RandomReal[BetaDistribution[x + α, m - x + 4], WorkingPrecision → prec];
            Do[{x0 = dist1[y0], y0 = dist2[x0]}, {1000}];
            Table[{x0 = dist1[y0], y0 = dist2[x0]}, {n}]]
```

For the specific Gibbs sampler constructed earlier, m was 16 and α was 2.

Here are 5 vectors from the sampler with $m = 16$ and $\alpha = 2$.

```
In[115]:= RandomReal[BinomialBetaSampler[16, 2], 5]
Out[115]= {{2, 0.0474404}, {1, 0.192054}, {6, 0.299769}, {3, 0.113683}, {1, 0.0480714}}
```

References

[1] Geman, S. and D. Geman. "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6, no. 6 (1984): 721-741.

[2] Casella, G. and E. I. George. "Explaining the Gibbs Sampler." *The American Statistician* 46, no. 3 (1992): 167-174.

[3] Matsumoto, M. and T. Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator." *ACM Transactions on Modeling and Computer Simulation* 8, no. 1 (1998): 3-30.

[4] Nishimura, T. "Tables of 64-Bit Mersenne Twisters." *ACM Transactions on Modeling and Computer Simulation* 10, no. 4 (2000): 348-357.

[5] Junod, P. "Cryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator." August 1999. http://crypto.junod.info/bbs.pdf

[6] Gentle, J. E. *Random Number Generation and Monte Carlo Methods*, (2nd ed.) Springer-Verlag, 2003.

[7] Johnson, N. L., S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions, Volume 2*, (2nd ed.) John Wiley & Sons, 1995.

[8] Smith, W. B. and R. R. Hocking. "Algorithm AS 53: Wishart Variate Generator." *Applied Statistics* 21, no. 3 (1972): 341-345.

[9] Cheng, R. C. H. and G. M. Feast. "Some Simple Gamma Variate Generators." *Applied Statistics* 28, no. 3 (1979): 290-295.

[10] Johnson, M. E. *Multivariate Statistical Simulation*. John Wiley & Sons, 1987.

[11] Jöhnk, M. D. "Erzeugung von Betaverteilten und Gammaverteilten Zufallszahlen." *Metrika* 8 (1964): 5-15.

[12] Cheng, R. C. H. "Generating Beta Variables with Nonintegral Shape Parameters." *Communications of the ACM* 21, no. 4 (1978): 317-322.

[13] Atkinson, A. C. "A Family of Switching Algorithms for the Computer Generation of Beta Random Variables." *Biometrika* 66, no. 1 (1979): 141-145.

[14] Bailey, R. W. "Polar Generation of Random Variates with the *t*-Distribution." *Mathematics of Computation* 62, no. 206 (1994): 779-781.

[15] Devroye, L. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.

[16] Kachitvichyanukul, V. and B. W. Schmeiser. "Binomial Random Variate Generation." *Communications of the ACM* 31, no. 2 (1988): 216-223.

[17] Kachitvichyanukul, V. and B. W. Schmeiser. "Computer Generation of Hypergeometric Random Variates." *Journal of Statistical Computation and Simulation* 22, no. 2 (1985): 127-145.

[18] Ahrens, J. H. and U. Dieter "Computer Generation of Poisson Deviates from Modified Normal Distributions." *ACM Transactions on Mathematical Software* 8, no. 2 (1982): 163-179.