

SSJ User's Guide

Package `randvar`

Generating Non-Uniform Random Numbers

Version: May 21, 2008

This package implements random number generators from various standard distributions. It also provides an interface to the C package UNURAN.

Contents

Overview	2
General Classes	4
RandomVariateGen	4
RandomVariateGenInt	5
RandomVariateGenWithCache	6
Generators for Discrete Distributions over the Integers	8
BinomialGen	8
BinomialConvolutionGen	9
GeometricGen	10
HypergeometricGen	11
LogarithmicGen	12
NegativeBinomialGen	13
PascalConvolutionGen	14
PoissonGen	15
PoissonTIACGen	16
UniformIntGen	17
Generators for Continuous Distributions	18
BetaGen	18
BetaRejectionLoglogisticGen	19
BetaStratifiedRejectionGen	20
BetaSymmetricalGen	21
BetaSymmetricalPolarGen	22
BetaSymmetricalBestGen	23
CauchyGen	24
ChiGen	25
ChiRatioOfUniformsGen	26
ChiSquareGen	27
ErlangGen	28
ErlangConvolutionGen	29
ExponentialGen	30

ExtremeValueGen	31
FatigueLifeGen	32
FisherFGen	33
GammaGen	34
GammaAcceptanceRejectionGen	35
GammaRejectionLoglogisticGen	36
HyperbolicSecantGen	37
InverseGaussianGen	38
KernelDensityGen	39
KernelDensityVarCorrectGen	41
LaplaceGen	42
LogisticGen	43
LoglogisticGen	44
LognormalGen	45
LognormalSpecialGen	46
NormalGen	47
NormalACRGen	48
NormalBoxMullerGen	49
NormalPolarGen	50
NormalKindermannRamageGen	51
ParetoGen	52
Pearson5Gen	53
Pearson6Gen	54
StudentGen	55
StudentPolarGen	56
TriangularGen	57
UniformGen	58
WeibullGen	59
UNURAN Interface	60
UnuranContinuous	60
UnuranDiscreteInt	61
UnuranEmpirical	62
UnuranException	63

Overview

This package provides a collection of classes for non-uniform random variate generation, primarily from standard distributions.

Each non-uniform random variate generator requires at least one `RandomStream` object (from package `rng`), used to generate the underlying uniform random numbers. With some variate generation methods (e.g., the *rejection* method), the number of uniforms required to get a single non-uniform variate varies from one call to the next. In that case, an auxiliary stream is often used to preserve the synchronization between random variates when implementing variance-reduction methods [23]. The main random number stream is called a fixed number of times per non-uniform variate generation. If more uniform random numbers are needed, they are obtained from the auxiliary stream. For these types of generators, two `RandomStream` objects should be passed to the constructor. Otherwise, by default, the same stream will be used for all uniforms.

The generic classes `RandomVariateGen` and `RandomVariateGenInt` permit one to construct a random variate generator from a random stream and an arbitrary distribution (from the package `Distribution`). To generate random variates by inversion from an arbitrary distribution over the real numbers, using a given random stream, one can construct a `RandomVariateGen` object with the desired (previously created) `Distribution` and `RandomStream` objects, and then call its `nextDouble` method. For discrete distributions over the integers, one can construct a `RandomVariateGenInt` object that contains the desired `DiscreteDistributionInt` and `RandomStream`, and call its `nextInt` method. By default, these generators simply call the `inverseF` method from the specified distribution object.

To generate random variates by other methods than inversion, one can use specialized classes that extend `RandomVariateGen` or `RandomVariateGenInt`. Such classes are provided for a variety of standard discrete and continuous distributions. For example, `NormalGen` extends `RandomVariateGen` and provides normal random variate generators based on inversion. Subclasses of `NormalGen` implement various non-inversion normal variate generation methods. To generate random variates with a specific method, it suffices to invoke the constructor of the appropriate subclass and then call its `nextDouble` method.

In most cases, the specialized classes maintain local copies of the distribution parameters and use them for variate generation. If the parameters of the contained distribution objects are later modified, this may lead to inconsistencies: the variate generator object will keep using the old values. In fact, the constructors of the specialized classes often precompute constants and tables based on these parameter values, which would have to be recomputed if the parameters are changed. On the other hand, the generic classes `RandomVariateGen` and `RandomVariateGenInt` call directly the `inverseF` method of the contained distribution object, so they will always use the new parameter values whenever the parameters in the distribution object are changed.

1

¹ From Pierre: It seems to me that in the future, only the constructors of `RandomVariateGen` and `RandomVariateGenInt` should require a distribution object. In the subclasses, we should directly pass the required parameters and there would not necessarily be a distribution object created. We should examine the implications of such a change.

Static methods in the specialized classes allow the generation of random variates from specific distributions without constructing a `RandomVariateGen` object.

This package also provides an interface to the *UNURAN* (Universal Non-Uniform Random number generators) package, a rich library of C functions designed and implemented by the ARVAG (Automatic Random Variate Generation) project group in Vienna [24]. This interface can be used to access distributions or generation methods not available directly in SSJ. To get a UNURAN generator, it suffices to instantiate one of the UNURAN interface classes: `UnuranDiscreteInt` for discrete random variates, `UnuranContinuous` for continuous ones (in one dimension), and `UnuranEmpirical` for quasi-empirical distributions based on experimental data. The type of distribution and its parameters are specified to UNURAN via its String API (see the UNURAN documentation). Only univariate distributions are supported because the UNURAN String API does not support the multivariate ones yet.

In the UNURAN interface classes, `nextDouble` and `nextInt` can be invoked as usual to generate variates, but these methods are slowed down significantly by the overhead in the interactions between code on the native side and on the Java side. When several random variates are needed, it is much more efficient to generate them in a single call, via the methods `nextArrayOfDouble` and `nextArrayOfInt`.

RandomVariateGen

This is the base class for all random variate generators over the real line. It specifies the signature of the `nextDouble` method, which is normally called to generate a real-valued random variate whose distribution has been previously selected. A random variate generator object can be created simply by invoking the constructor of this class with previously created `RandomStream` and `Distribution` objects, or by invoking the constructor of a subclass. By default, all random variates will be generated via inversion by calling the `inverseF` method for the distribution, even though this can be inefficient in some cases. For some of the distributions, there are subclasses with special and more efficient methods to generate the random variates.

For generating many random variates, creating an object and calling the non-static method is more efficient when the generating algorithm involves a significant setup. When no work is done at setup time, the static methods are usually slightly faster.

```
package umontreal.iro.lecuyer.randvar;

public class RandomVariateGen
```

Constructor

```
public RandomVariateGen (RandomStream s, Distribution dist)
```

Creates a new random variate generator from the distribution `dist`, using stream `s`.

Methods

```
public double nextDouble()
```

Generates a random number from the continuous distribution contained in this object. By default, this method uses inversion by calling the `inverseF` method of the distribution object. Alternative generating methods are provided in subclasses.

```
public void nextArrayOfDouble (double[] v, int start, int n)
```

Generates `n` random numbers from the continuous distribution contained in this object. These numbers are stored in the array `v`, starting from index `start`. By default, this method calls `nextDouble()` `n` times, but one can override it in subclasses for better efficiency.

```
public RandomStream getStream()
```

Returns the `RandomStream` used by this generator.

```
public void setStream (RandomStream stream)
```

Sets the `RandomStream` used by this generator to `stream`.

```
public Distribution getDistribution()
```

Returns the `Distribution` used by this generator.

RandomVariateGenInt

This is the base class for all generators of discrete random variates over the set of integers. Similar to `RandomVariateGen`, except that the generators produce integers, via the `nextInt` method, instead of real numbers.

```
package umontreal.iro.lecuyer.randvar;  
  
public class RandomVariateGenInt extends RandomVariateGen
```

Constructor

```
public RandomVariateGenInt (RandomStream s, DiscreteDistributionInt dist)
```

Creates a new random variate generator for the discrete distribution `dist`, using stream `s`.

Methods

```
public int nextInt()
```

Generates a random number (an integer) from the discrete distribution contained in this object. By default, this method uses inversion by calling the `inverseF` method of the distribution object. Alternative generating methods are provided in subclasses.

```
public void nextArrayOfInt (int[] v, int start, int n)
```

Generates `n` random numbers from the discrete distribution contained in this object. The results are stored into the array `v`, starting from index `start`. By default, this method calls `nextInt()` `n` times, but one can reimplement it in subclasses for better efficiency.

RandomVariateGenWithCache

This class represents a random variate generator whose values are cached for more efficiency when using common random numbers. An object from this class is constructed with a reference to a `RandomVariateGen` instance used to get the random numbers. These numbers are stored in an internal array to be retrieved later. The dimension of the array increases as the values are generated. If the `nextDouble` method is called after the object is reset, it gives back the cached values instead of computing new ones. If the cache is exhausted before the generator is reset, new values are computed and added to the cache.

Such caching allows for a better performance with common random numbers, when generating random variates is time-consuming. However, using such caching may lead to memory problems if a large quantity of random numbers are needed.

```
package umontreal.iro.lecuyer.randvar;  
  
public class RandomVariateGenWithCache extends RandomVariateGen
```

Constructors

```
public RandomVariateGenWithCache (RandomVariateGen rvg)
```

Constructs a new cached random variate generator with internal generator `rvg`.

```
public RandomVariateGenWithCache (RandomVariateGen rvg,  
                                  int initialCapacity)
```

Constructs a new cached random variate generator with internal generator `rvg`. The `initialCapacity` parameter is used to set the initial capacity of the internal array which can grow as needed; it does not limit the maximal number of cached values.

Methods

```
public boolean isCaching()
```

Determines if the random variate generator is caching values, default being `true`. When caching is turned OFF, the `nextDouble` method simply calls the corresponding method on the internal random variate generator, without storing the generated values.

```
public void setCaching (boolean caching)
```

Sets the caching indicator to `caching`. If caching is turned OFF, this method calls `clearCache` to clear the cached values.

```
public RandomVariateGen getCachedGen()
```

Returns a reference to the random variate generator whose values are cached.

```
public void setCachedGen (RandomVariateGen rvg)
```

Sets the random variate generator whose values are cached to `rvg`. If the generator is changed, the `clearCache` method is called.

```
public void clearCache()
```

Clears the cached values for this cached generator. Any subsequent call will then obtain new values from the internal generator.

```
public void initCache()
```

Resets this generator to recover values from the cache. Subsequent calls to `nextDouble` will return the cached random values until all the values are returned. When the array of cached values is exhausted, the internal random variate generator is used to generate new values which are added to the internal array as well. This method is equivalent to calling `setCacheIndex`.

```
public int getNumCachedValues()
```

Returns the total number of values cached by this generator.

```
public int getCacheIndex()
```

Return the index of the next cached value that will be returned by the generator. If the cache is exhausted, the returned value corresponds to the value returned by `getNumCachedValues`, and a subsequent call to `nextDouble` will generate a new variate rather than reading a previous one from the cache. If caching is disabled, this always returns 0.

```
public void setCacheIndex (int newIndex)
```

Sets the index, in the cache, of the next value returned by `nextDouble`. If `newIndex` is 0, this is equivalent to calling `initCache`. If `newIndex` is `getNumCachedValues`, subsequent calls to `nextDouble` will add new values to the cache.

```
public DoubleArrayList getCachedValues()
```

Returns an array list containing the values cached by this random variate generator.

```
public void setCachedValues (DoubleArrayList values)
```

Sets the array list containing the cached values to `values`. This resets the cache index to the size of the given array.

BinomialGen

This class implements random variate generators for the *binomial* distribution. It has parameters n and p with mass function

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x} = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x} \quad \text{for } x = 0, 1, 2, \dots, n \quad (1)$$

where n is a positive integer, and $0 \leq p \leq 1$.

No local copy of the parameters n and p is maintained in this class. The (non-static) `nextInt` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class BinomialGen extends RandomVariateGenInt
```

Constructors

```
public BinomialGen (RandomStream s, BinomialDist dist)
```

Creates a new random variate generator for the *binomial* distribution `dist` and the random stream `s`.

Methods

```
public static int nextInt (RandomStream s, int n, double p)
```

Generates a new integer from the *binomial* distribution with parameters $n = n$ and $p = p$, using the given stream `s`.

BinomialConvolutionGen

Implements binomial random variate generators using the convolution method. This method generates n Bernoulli random variates with parameter p and adds them up. Its advantages are that it requires little computer memory and no setup time. Its disadvantage is that it is very slow for large n . It makes sense only when n is small.

A local copy of the parameters n and p is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;  
public class BinomialConvolutionGen extends BinomialGen
```

Constructors

```
public BinomialConvolutionGen (RandomStream s, BinomialDist dist)
```

Creates a new random variate generator for distribution `dist` and stream `s`.

GeometricGen

This class implements a random variate generator for the *geometric* distribution. It has parameter p and mass function

$$p(x) = p(1 - p)^x \text{ for } x = 0, 1, 2, \dots, \quad (2)$$

where $0 \leq p \leq 1$. Random variates are generated by calling inversion on the distribution object.

```
package umontreal.iro.lecuyer.randvar;
public class GeometricGen extends RandomVariateGenInt
```

Constructors

```
public GeometricGen (RandomStream s, GeometricDist dist)
    Creates a new generator for the distribution dist, using stream s.
```

Methods

```
public static int nextInt (RandomStream s, double p)
    Generates a new geometric random variate with parameter  $p = p$ , using stream s, by inversion.
```

HypergeometricGen

This class implements random variate generators for the *hypergeometric* distribution. Its mass function is (see, e.g., [14, page 101])

$$p(x) = \frac{\binom{m}{x} \binom{l-m}{k-x}}{\binom{l}{k}} \quad \text{for } x = \max(0, k - l + m), \dots, \min(k, m), \quad (3)$$

where m , l and k are integers that satisfy $0 < m \leq l$ and $0 < k \leq l$.

The generation method is inversion using the chop-down algorithm [20]

```
package umontreal.iro.lecuyer.randvar;
public class HypergeometricGen extends RandomVariateGenInt
```

Constructors

```
public HypergeometricGen (RandomStream s, HypergeometricDist dist)
    Creates a new generator for distribution dist, using stream s.
```

Methods

```
public static int nextInt (RandomStream s, int m, int l, int k)
    Generates a new variate from the hypergeometric distribution with parameters  $m = m$ ,  $l = l$  and  $k = k$ , using stream s.
```

LogarithmicGen

This class implements random variate generators for the (discrete) *logarithmic* distribution. Its mass function is

$$p(x) = \frac{-\theta^x}{x \log(1 - \theta)} \quad \text{for } x = 1, 2, \dots, \quad (4)$$

where $0 < \theta < 1$. It uses inversion with the LS chop-down algorithm if $\theta < \theta_0$ and the LK transformation algorithm if $\theta \geq \theta_0$, as described in [21]. The threshold θ_0 can be specified when invoking the constructor. Its default value is $\theta_0 = 0.96$, as suggested in [21]. ²

A local copy of the parameter θ is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;
public class LogarithmicGen extends RandomVariateGenInt
```

Constructors

```
public LogarithmicGen (RandomStream s, LogarithmicDist dist)
```

Creates a new generator with distribution `dist` and stream `s`, with default value $\theta_0 = 0.96$.

```
public LogarithmicGen (RandomStream s, LogarithmicDist dist, double theta0)
```

Creates a new generator with distribution `dist` and stream `s`, with $\theta_0 = \text{theta0}$.

Methods

```
public static int nextInt (RandomStream s, double theta)
```

Uses stream `s` to generate a new variate from the *logarithmic* distribution with parameter $\theta = \text{theta}$.

² From Pierre: Does this work for any θ_0 ? Should we add constraints?

NegativeBinomialGen

This class implements random variate generators having the *negative binomial* distribution. Its mass function is

$$p(x) = \binom{n+x-1}{x} p^n (1-p)^x \quad \text{for } x = 0, 1, \dots, \quad (5)$$

where $n \geq 1$ and $0 \leq p \leq 1$.

No local copy of the parameters n and p is maintained in this class. The (non-static) `nextInt` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class NegativeBinomialGen extends RandomVariateGenInt
```

Constructors

```
public NegativeBinomialGen (RandomStream s, NegativeBinomialDist dist)
```

Creates a new generator for the distribution `dist`, using stream `s`.

Methods

```
public static int nextInt (RandomStream s, int n, double p)
```

Generates a new variate from the *negative binomial* distribution, with parameters $n = n$ and $p = p$, using stream `s`.

PascalConvolutionGen

Implements Pascal random variate generators by the *convolution* method (see [23]). The method generates n geometric variates with probability p and adds them up.

The algorithm is slow if n is large. A local copy of the parameters n and p is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;  
public class PascalConvolutionGen extends NegativeBinomialGen
```

Constructors

```
public PascalConvolutionGen (RandomStream s, PascalDist dist)  
    Creates a new generator for the distribution dist, using stream s.
```

PoissonGen

This class implements random variate generators having the *Poisson* distribution. Its mass function is

$$p(x) = \frac{e^{-\lambda} \lambda^x}{x!} \quad \text{for } x = 0, 1, \dots, \quad (6)$$

where $\lambda > 0$ is a real valued parameter equal to the mean.

No local copy of the parameter $\lambda = \text{lambda}$ is maintained in this class. The (non-static) `nextInt` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class PoissonGen extends RandomVariateGenInt
```

Constructor

```
public PoissonGen (RandomStream s, PoissonDist dist)
```

Creates a new random variate generator using the Poisson distribution `dist` and stream `s`.

Methods

```
public static int nextInt (RandomStream s, double lambda)
```

A static method for generating a random variate from a *Poisson* distribution with parameter $\lambda = \text{lambda}$.

PoissonTIACGen

This class implements random variate generators having the *Poisson* distribution (see `PoissonGen`). Uses the tabulated inversion combined with acceptance complement (*TIAC*) method of [2]. The implementation is adapted from UNURAN [24].

A local copy of the parameter `lambda` is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;  
  
public class PoissonTIACGen extends PoissonGen
```

Constructor

```
public PoissonTIACGen (RandomStream s, PoissonDist dist)
```

Creates a new random variate generator using the Poisson distribution `dist` and stream `s`.

UniformIntGen

This class implements a random variate generator for the *uniform* distribution over integers, over the interval $[i, j]$. Its mass function is

$$p(x) = \frac{1}{j - i + 1} \quad \text{for } x = i, i + 1, \dots, j \quad (7)$$

and 0 elsewhere.

```
package umontreal.iro.lecuyer.randvar;
public class UniformIntGen extends RandomVariateGenInt
```

Constructors

```
public UniformIntGen (RandomStream s, UniformIntDist dist)
```

Creates a new generator for the distribution `dist`, using stream `s`.

Methods

```
public static int nextInt (RandomStream s, int i, int j)
```

Generates a new *uniform* random variate over the interval $[i, j]$, using stream `s`, by inversion.

BetaGen

This class implements random variate generators with the *beta* distribution with shape parameters $\alpha > 0$ and $\beta > 0$, over the interval (a, b) , where $a < b$. The density function of this distribution is

$$f(x) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)(b - a)^{\alpha + \beta - 1}} (x - a)^{\alpha - 1} (b - x)^{\beta - 1} \quad \text{for } a < x < b, \quad (8)$$

and $f(x) = 0$ elsewhere, where $\Gamma(x)$ is the gamma function defined in (19).

Local copies of the parameters α , β , a , and b are maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class BetaGen extends RandomVariateGen
```

Constructors

```
public BetaGen (RandomStream s, BetaDist dist)
```

Creates a new generator for the distribution `dist`, using stream `s`.

Methods

```
public static double nextDouble (RandomStream s,
                                double alpha, double beta,
                                double a, double b)
```

Generates a variate from the *beta* distribution with parameters $\alpha = \text{alpha}$, $\beta = \text{beta}$, over the interval (a, b) , using stream `s`.

BetaRejectionLoglogisticGen

Implements *Beta* random variate generators using the rejection method with log-logistic envelopes from [10]. The method draws the first two uniforms from the main stream and uses the auxiliary stream for the remaining uniforms, when more than two are needed (i.e., when rejection occurs).

The current implementation is adapted from UNURAN.

```
package umontreal.iro.lecuyer.randvar;
public class BetaRejectionLoglogisticGen extends BetaGen
```

Constructors

```
public BetaRejectionLoglogisticGen (RandomStream s, RandomStream aux,
                                     BetaDist dist)
```

Creates a new generator for the distribution `dist`, using stream `s` and auxiliary stream `aux`. The main stream is used for the first uniforms (before a rejection occurs) and the auxiliary stream is used afterwards (after the first rejection).

```
public BetaRejectionLoglogisticGen (RandomStream s, BetaDist dist)
```

Same as `BetaRejectionLoglogisticGen (s, s, dist)`. The auxiliary stream used will be the same as the main stream.

Methods

```
public RandomStream getAuxStream()
```

Returns the auxiliary stream associated with that object.

BetaStratifiedRejectionGen

This class implements *Beta* random variate generators using the stratified rejection/patchwork rejection method from [26, 28]. This method draws one uniform from the main stream and uses the auxiliary stream for any additional uniform variates that might be needed.

```
package umontreal.iro.lecuyer.randvar;  
  
public class BetaStratifiedRejectionGen extends BetaGen
```

Constructors

```
public BetaStratifiedRejectionGen (RandomStream s, RandomStream aux,  
                                   BetaDist dist)
```

Creates a new generator for the distribution `dist`, using the given stream `s` and auxiliary stream `aux`. The auxiliary stream is used when a random number of variates must be drawn from the main stream.

```
public BetaStratifiedRejectionGen (RandomStream s, BetaDist dist)
```

Same as `BetaStratifiedRejectionGen(s, s, dist)`. The auxiliary stream used will be the same as the main stream.

Methods

```
public RandomStream getAuxStream()
```

Returns the auxiliary stream associated with this object.

BetaSymmetricalGen

This class implements random variate generators with the *symmetrical beta* distribution with shape parameters $\alpha = \beta$, over the interval $(0, 1)$.

```
package umontreal.iro.lecuyer.randvar;  
public class BetaSymmetricalGen extends BetaGen
```

Constructors

```
public BetaSymmetricalGen (RandomStream s, BetaSymmetricalDist dist)  
    Creates a new generator for the distribution dist, using stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double alpha)
```

BetaSymmetricalPolarGen

This class implements *symmetrical beta* random variate generators using Ulrich's polar method [29]. The method generates two uniform random variables $x \in [0, 1]$ and $y \in [-1, 1]$ until $x^2 + y^2 \leq 1$. Then it returns

$$\frac{1}{2} + \frac{xy}{S} \sqrt{1 - S^{2/(2\alpha-1)}} \quad (9)$$

where $S = x^2 + y^2$, and α is the shape parameter of the beta distribution. The method is valid only when $\alpha > 1/2$.

```
package umontreal.iro.lecuyer.randvar;
public class BetaSymmetricalPolarGen extends BetaSymmetricalGen
```

Constructors

```
public BetaSymmetricalPolarGen (RandomStream stream, RandomStream s2,
                                BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using stream `stream` to generate x , and stream `s2` to generate y as described in eq. (9) above. Restriction: `dist` must have $\alpha > 1/2$.

```
public BetaSymmetricalPolarGen (RandomStream stream, BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using only one stream `stream`. Restriction: `dist` must have $\alpha > 1/2$.

Methods

```
public static double nextDouble (RandomStream s1, RandomStream s2,
                                 double alpha)
```

Generates a random number using Ulrich's polar method. Stream `s1` generates x and stream `s2` generates y [see eq. (9)]. Restriction: $\alpha > 1/2$.

```
public static double nextDouble (RandomStream s, double alpha)
```

Generates a random number using Ulrich's polar method with only one stream `s`. Restriction: $\alpha > 1/2$.

```
public RandomStream getStream2()
```

Returns stream `s2` associated with this object.

BetaSymmetricalBestGen

This class implements *symmetrical beta* random variate generators using Devroye's one-liner method. It is based on Best's relation [6] between a Student- t variate and a symmetrical beta variate:

$$B_{\alpha,\alpha} \stackrel{\mathcal{L}}{=} \frac{1}{2} \left(1 + \frac{T_{2\alpha}}{\sqrt{2\alpha + T_{2\alpha}^2}} \right).$$

If S is a random sign and U_1, U_2 are two independent uniform $[0, 1]$ random variates, then the following gives a symmetrical beta variate [12]:

$$B_{\alpha,\alpha} \stackrel{\mathcal{L}}{=} \frac{1}{2} + \frac{S}{2 \sqrt{1 + \frac{1}{(U_1^{-1/\alpha} - 1) \cos^2(2\pi U_2)}}}} \quad (10)$$

valid for any shape parameter $\alpha > 0$.

```
package umontreal.iro.lecuyer.randvar;
```

```
public class BetaSymmetricalBestGen extends BetaSymmetricalGen
```

Constructors

```
public BetaSymmetricalBestGen (RandomStream stream, RandomStream s2,
                               RandomStream s3, BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using stream `stream` to generate U_1 , stream `s2` to generate U_2 and stream `s3` to generate S as given in equation (10).

```
public BetaSymmetricalBestGen (RandomStream stream,
                               BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using only one stream `stream`.

Methods

```
public static double nextDouble (RandomStream s1, RandomStream s2,
                                RandomStream s3, double alpha)
```

Generates a random number using Devroye's one-liner method. Restriction: $\alpha > 0$.

```
public static double nextDouble (RandomStream s, double alpha)
```

Generates a random number using Devroye's one-liner method with only one stream `s`. Restriction: $\alpha > 0$.

```
public RandomStream getStream2()
```

Returns stream `s2` associated with this object.

```
public RandomStream getStream3()
```

Returns stream `s3` associated with this object.

CauchyGen

This class implements random variate generators for the *Cauchy* distribution. The density is (see, e.g., [18] p. 299):

$$f(x) = \frac{\beta}{\pi[(x - \alpha)^2 + \beta^2]}, \quad \text{for } -\infty < x < \infty. \quad (11)$$

where $\beta > 0$.

No local copy of the parameters α and β is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class CauchyGen extends RandomVariateGen
```

Constructors

```
public CauchyGen (RandomStream s, CauchyDist dist)
```

Create a new generator for the distribution `dist`, using stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double alpha, double beta)
```

Generates a new variate from the *Cauchy* distribution with parameters $\alpha = \text{alpha}$ and $\beta = \text{beta}$, using stream `s`.

ChiGen

This class implements random variate generators for the *chi* distribution. It has $\nu > 0$ degrees of freedom and its density function is (see [18], page 417)

$$f(x) = \frac{e^{-x^2/2} x^{\nu-1}}{2^{(\nu/2)-1} \Gamma(\nu/2)} \quad \text{for } x > 0, \quad (12)$$

where $\Gamma(x)$ is the gamma function defined in (19).

No local copy of the parameter ν is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution (slow).

```
package umontreal.iro.lecuyer.randvar;
public class ChiGen extends RandomVariateGen
```

Constructors

```
public ChiGen (RandomStream s, ChiDist dist)
```

Create a new generator for the distribution `dist`, using stream `s`.

Methods

```
public static double nextDouble (RandomStream s, int nu)
```

Generates a random variate from the chi distribution with $\nu = \text{nu}$ degrees of freedom, using stream `s`.

ChiRatioOfUniformsGen

This class implements *Chi* random variate generators using the ratio of uniforms method with shift.

A local copy of the parameter ν is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;  
public class ChiRatioOfUniformsGen extends ChiGen
```

Constructors

```
public ChiRatioOfUniformsGen (RandomStream s, ChiDist dist)
```

Create a new generator for the distribution `dist`, using stream `s`.

ChiSquareGen

This class implements random variate generators with the *chi square* distribution with $n > 0$ degrees of freedom. Its density function is

$$f(x) = \frac{e^{-x/2} x^{n/2-1}}{2^{n/2} \Gamma(n/2)} \quad \text{for } x > 0, \quad (13)$$

where $\Gamma(x)$ is the gamma function defined in (19).

No local copy of the parameter n is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class ChiSquareGen extends RandomVariateGen
```

Constructors

```
public ChiSquareGen (RandomStream s, ChiSquareDist dist)
```

Create a new generator for the distribution `dist` and stream `s`.

Methods

```
public static double nextDouble (RandomStream s, int n)
```

Generates a new variate from the chi square distribution with n degrees of freedom, using stream `s`.

ErlangGen

This class implements random variate generators for the *Erlang* distribution with parameters $k > 0$ and $\lambda > 0$. This Erlang random variable is the sum of k exponentials with parameter λ and has mean k/λ .

No local copy of the parameters k and λ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
packageumontreal.iro.lecuyer.randvar;  
publicclassErlangGenextendsGammaGen
```

Constructors

```
publicErlangGen(RandomStream s, ErlangDist dist)  
    Creates a new generator for the distribution dist and stream s.
```

Methods

```
publicstaticdoublenextDouble(RandomStream s, int k, double lambda)  
    Generates a new variate from the Erlang distribution with parameters  $k = k$  and  $\lambda = lambda$ ,  
    using stream s.
```

ErlangConvolutionGen

This class implements *Erlang* random variate generators using the *convolution* method. This method uses inversion to generate k exponential variates with parameter λ and returns their sum.

A local copy of the parameters k and λ is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;  
public class ErlangConvolutionGen extends ErlangGen
```

Constructors

```
public ErlangConvolutionGen (RandomStream s, ErlangDist dist)  
    Creates a new generator for the distribution dist and stream s.
```

ExponentialGen

This class implements random variate generators for the *exponential* distribution. The density is

$$f(x) = \lambda e^{-\lambda x} \quad \text{for } x \geq 0, \quad (14)$$

where $\lambda > 0$.

No local copy of the parameter λ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class ExponentialGen extends RandomVariateGen
```

Constructors

```
public ExponentialGen (RandomStream s, ExponentialDist dist)
    Creates a new generator for the exponential distribution dist and stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double lambda)
    Uses inversion to generate a new exponential variate with parameter  $\lambda = \text{lambda}$ , using stream s.
```

ExtremeValueGen

This class implements random variate generators for the *Gumbel* (or *extreme value*) distribution. Its density is

$$f(x) = \lambda e^{-e^{-\lambda(x-\alpha)} - \lambda(x-\alpha)} \quad \text{for } x > 0, \quad (15)$$

where $\lambda > 0$.

No local copy of the parameters α and λ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class ExtremeValueGen extends RandomVariateGen
```

Constructors

```
public ExtremeValueGen (RandomStream s, ExtremeValueDist dist)
```

Creates a new generator object for distribution `dist` and stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double alpha,
                                double lambda)
```

Uses inversion to generate a new variate from the extreme value distribution with parameters $\alpha = \text{alpha}$ and $\lambda = \text{lambda}$, using stream `s`.

FatigueLifeGen

This class implements random variate generators for the *Fatigue Life* distribution with location parameter μ , scale parameter β and shape parameter γ . The density function of this distribution is

$$f(x) = \left[\frac{\sqrt{\frac{x-\mu}{\beta}} + \sqrt{\frac{\beta}{x-\mu}}}{2\gamma(x-\mu)} \right] \phi \left(\frac{\sqrt{\frac{x-\mu}{\beta}} - \sqrt{\frac{\beta}{x-\mu}}}{\gamma} \right) \quad (16)$$

where ϕ is the probability density of the standard normal distribution.

```
package umontreal.iro.lecuyer.randvar;
public class FatigueLifeGen extends RandomVariateGen
```

Constructors

```
public FatigueLifeGen (RandomStream s, FatigueLifeDist dist)
    Creates a new generator for the distribution dist, using stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double mu, double beta,
                                double gamma)
    Generates a variate from the Fatigue Life distribution with location parameter  $\mu$ , scale parameter  $\beta$  and shape parameter  $\gamma$ .
```

FisherFGen

This class implements random variate generators for the *Fisher F*-distribution with n and m degrees of freedom, where n and m are positive integers. The density function of this distribution is

$$f(x) = \frac{\Gamma(\frac{n+m}{2})n^{\frac{n}{2}}m^{\frac{m}{2}}}{\Gamma(\frac{n}{2})\Gamma(\frac{m}{2})} \frac{x^{\frac{n-2}{2}}}{(m+nx)^{\frac{n+m}{2}}}, \quad \text{for } x > 0 \quad (17)$$

```
package umontreal.iro.lecuyer.randvar;
public class FisherFGen extends RandomVariateGen
```

Constructors

```
public FisherFGen (RandomStream s, FisherFDist dist)
    Creates a new generator for the distribution dist, using stream s.
```

Methods

```
public static double nextDouble (RandomStream s, int n, int m)
    Generates a variate from the Fisher F-distribution with  $n$  and  $m$  degrees of freedom, using stream s.
```

GammaGen

This class implements random variate generators for the *gamma* distribution. Its parameters are $\alpha > 0$ and $\lambda > 0$. Its density function is

$$f(x) = \lambda^\alpha x^{\alpha-1} e^{-\lambda x} / \Gamma(\alpha) \quad \text{for } x > 0, \quad (18)$$

where Γ is the gamma function defined by

$$\Gamma(\alpha) = \int_0^\infty x^{\alpha-1} e^{-x} dx. \quad (19)$$

No local copy of the parameters α and λ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class GammaGen extends RandomVariateGen
```

Constructors

```
public GammaGen (RandomStream s, GammaDist dist)
    Creates a new generator object for the gamma distribution dist and stream s.
```

Methods

```
public static double nextDouble (RandomStream s,
                                double alpha, double lambda)
    Generates a new gamma random variate with parameters  $\alpha = \text{alpha}$  and  $\lambda = \text{lambda}$ , using stream s.
```

GammaAcceptanceRejectionGen

This class implements *gamma* random variate generators using a method that combines acceptance-rejection with acceptance-complement, and proposed in [1, 3]. It uses acceptance-rejection for $\alpha < 1$ and acceptance-complement for $\alpha \geq 1$. For each gamma variate, the first uniform required is taken from the main stream and all additional uniforms (after the first rejection) are obtained from the auxiliary stream.

```
package umontreal.iro.lecuyer.randvar;

public class GammaAcceptanceRejectionGen extends GammaGen
```

Constructors

```
public GammaAcceptanceRejectionGen (RandomStream s, GammaDist dist)
```

Creates a new generator object for the gamma distribution `dist` and stream `s` for both the main and auxiliary stream.

```
public GammaAcceptanceRejectionGen (RandomStream s, RandomStream aux,
                                     GammaDist dist)
```

Creates a new generator object for the gamma distribution `dist`, using main stream `s` and auxiliary stream `aux`. The auxiliary stream is used when a random number of uniforms is required for a rejection-type generation method.

Methods

```
public RandomStream getAuxStream()
```

Returns the auxiliary stream associated with this object.

```
public static double nextDouble (RandomStream s, RandomStream aux,
                                double alpha, double lambda)
```

Generates a new gamma variate with parameters $\alpha = \text{alpha}$ and $\lambda = \text{lambda}$, using main stream `s` and auxiliary stream `aux`.

```
public static double nextDouble (RandomStream s, double alpha,
                                double lambda)
```

Same as `nextDouble (s, s, alpha, lambda)`.

GammaRejectionLoglogisticGen

This class implements *gamma* random variate generators using a rejection method with loglogistic envelopes, from [9]. For each gamma variate, the first two uniforms are taken from the main stream and all additional uniforms (after the first rejection) are obtained from the auxiliary stream.

```
package umontreal.iro.lecuyer.randvar;

public class GammaRejectionLoglogisticGen extends GammaGen
```

Constructors

```
public GammaRejectionLoglogisticGen (RandomStream s, GammaDist dist)
```

Creates a new generator object for the gamma distribution `dist` and stream `s` for both the main and auxiliary stream.

```
public GammaRejectionLoglogisticGen (RandomStream s, RandomStream aux,
                                     GammaDist dist)
```

Creates a new generator object for the gamma distribution `dist`, using main stream `s` and auxiliary stream `aux`. The auxiliary stream is used when a random number of uniforms is required for a rejection-type generation method.

Methods

```
public RandomStream getAuxStream()
```

Returns the auxiliary stream associated with this object.

```
public static double nextDouble (RandomStream s, RandomStream aux,
                                double alpha, double lambda)
```

Generates a new gamma variate with parameters $\alpha = \text{alpha}$ and $\lambda = \text{lambda}$, using main stream `s` and auxiliary stream `aux`.

```
public static double nextDouble (RandomStream s, double alpha,
                                double lambda)
```

Same as `nextDouble (s, s, alpha, lambda)`.

HyperbolicSecantGen

This class implements random variate generators for the *Hyperbolic Secant* distribution with location parameter μ and scale parameter σ . The density function of this distribution is

$$f(x) = \frac{1}{2\sigma} \operatorname{sech}\left(\frac{\pi}{2} \frac{(x - \mu)}{\sigma}\right). \quad (20)$$

```
package umontreal.iro.lecuyer.randvar;
public class HyperbolicSecantGen extends RandomVariateGen
```

Constructors

```
public HyperbolicSecantGen (RandomStream s, HyperbolicSecantDist dist)
    Creates a new generator for the distribution dist, using stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double mu, double sigma)
    Generates a variate from the Hyperbolic Secant distribution with location parameter  $\mu$  and scale parameter  $\sigma$ .
```

InverseGaussianGen

This class implements random variate generators for the *inverse Gaussian* distribution with location parameter $\mu > 0$ and scale parameter $\lambda > 0$. The density function of this distribution is

$$f(x) = \sqrt{\frac{\lambda}{2\pi x^3}} \exp\left\{-\frac{\lambda(x-\mu)^2}{2\mu^2 x}\right\} \quad \text{for } x > 0. \quad (21)$$

```
package umontreal.iro.lecuyer.randvar;
public class InverseGaussianGen extends RandomVariateGen
```

Constructors

```
public InverseGaussianGen (RandomStream s, InverseGaussianDist dist)
    Creates a new generator for the distribution dist, using stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double mu, double lambda)
    Generates a variate from the inverse gaussian distribution with location parameter  $\mu > 0$  and scale parameter  $\lambda > 0$ .
```

KernelDensityGen

This class implements random variate generators for distributions obtained via *kernel density estimation* methods from a set of n individual observations x_1, \dots, x_n [13, 11, 16, 17, 27]. The basic idea is to center a copy of the same symmetric density at each observation and take an equally weighted mixture of the n copies as an estimator of the density from which the observations come. The resulting kernel density has the general form

$$f_n(x) = \frac{1}{nh} \sum_{i=1}^n k((x - x_i)/h), \quad (22)$$

where k is a fixed pre-selected density called the *kernel* and h is a positive constant called the *bandwidth* or *smoothing factor*. A difficult practical issue is the selection of k and h . Several approaches have been proposed for that; see, e.g., [5, 8, 17, 27].

The constructor of a generator from a kernel density requires a random stream s , the n observations in the form of an empirical distribution, a random variate generator for the kernel density k , and the value of the bandwidth h . The random variates are then generated as follows: select an observation x_I at random, by inversion, using stream s , then generate random variate Y with the generator provided for the density k , and return $x_I + hY$.

A simple formula for the bandwidth, suggested in [27, 17], is $h = \alpha_k h_0$, where

$$h_0 = 1.36374 \min(s_n, q/1.34)n^{-1/5}, \quad (23)$$

s_n and q are the empirical standard deviation and the interquartile range of the n observations, and α_k is a constant that depends on the type of kernel k . It is defined by

$$\alpha_k = \left(\sigma_k^{-4} \int_{-\infty}^{\infty} k(x) dx \right)^{1/5} \quad (24)$$

where σ_k is the standard deviation of the density k . The static method `getBaseBandwidth` permits one to compute h_0 for a given empirical distribution.

Table 1: Some suggested kernels

name	constructor	α_k	σ_k^2	efficiency
Epanechnikov	<code>BetaSymmetricDist (2.0, -1.0, 1.0)</code>	1.7188	1/5	1.000
triangular	<code>TriangularDist (-1.0, 1.0, 0.0)</code>	1.8882	1/6	0.986
Gaussian	<code>NormalDist()</code>	0.7764	1	0.951
boxcar	<code>UniformDist (-1.0, 1.0)</code>	1.3510	1/3	0.930
logistic	<code>LogisticDist()</code>	0.4340	3.2899	0.888
Student-t(3)	<code>StudentDist (3.0)</code>	0.4802	3	0.674

Table 1 gives the precomputed values of σ_k and α_k for selected (popular) kernels. The values are taken from [17]. The second column gives the name of a function (in this package)

that constructs the corresponding distribution. The *efficiency* of a kernel is defined as the ratio of its mean integrated square error over that of the Epanechnikov kernel, which has optimal efficiency and corresponds to the beta distribution with parameters $(2, 2)$ over the interval $(-1, 1)$.

```
package umontreal.iro.lecuyer.randvar;
public class KernelDensityGen extends RandomVariateGen
```

Constructors

```
public KernelDensityGen (RandomStream s, EmpiricalDist dist,
                        RandomVariateGen kGen, double h)
```

Creates a new generator for a kernel density estimated from the observations given by the empirical distribution `dist`, using stream `s` to select the observations, generator `kGen` to generate the added noise from the kernel density, and bandwidth `h`.

```
public KernelDensityGen (RandomStream s, EmpiricalDist dist,
                        NormalGen kGen)
```

This constructor uses a gaussian kernel and the default bandwidth $h = \alpha_k h_0$ with the α_k suggested in Table 1 for the gaussian distribution. This kernel has an efficiency of 0.951.

Kernel selection and parameters

```
public static double getBaseBandwidth (EmpiricalDist dist)
```

Computes and returns the value of h_0 in (23).

```
public void setBandwidth (double h)
```

Sets the bandwidth to `h`.

```
public void setPositiveReflection (boolean reflect)
```

After this method is called with `true`, the generator will produce only positive values, by using the *reflection method*: replace all negative values by their *absolute values*. That is, `nextDouble` will return $|x|$ if x is the generated variate. The mechanism is disabled when the method is called with `false`.

KernelDensityVarCorrectGen

This class is a variant of `KernelDensityGen`, but with a rescaling of the empirical distribution so that the variance of the density used to generate the random variates is equal to the empirical variance, as suggested by [27].

Let \bar{x}_n and s_n^2 be the sample mean and sample variance of the observations. The distance between each generated random variate and the sample mean \bar{x}_n is multiplied by the correcting factor $1/\sigma_e$, where $\sigma_e^2 = 1 + (h\sigma_k/s_n)^2$. The constant σ_k^2 must be passed to the constructor. Its value can be found in Table 1 for some popular kernels.

```
package umontreal.iro.lecuyer.randvar;
public class KernelDensityVarCorrectGen extends KernelDensityGen
```

Constructors

```
public KernelDensityVarCorrectGen (RandomStream s, EmpiricalDist dist,
                                   RandomVariateGen kGen, double h, double sigmaK2)
```

Creates a new generator for a kernel density estimated from the observations given by the empirical distribution `dist`, using stream `s` to select the observations, generator `kGen` to generate the added noise from the kernel density, bandwidth `h`, and $\sigma_k^2 = \text{sigmaK2}$ used for the variance correction.

```
public KernelDensityVarCorrectGen (RandomStream s, EmpiricalDist dist,
                                   NormalGen kGen)
```

This constructor uses a gaussian kernel and the default bandwidth suggested in Table 1 for the gaussian distribution.

LaplaceGen

This class implements methods for generating random variates from the *Laplace* distribution. Its density is (see [19, page 165])

$$f(x) = \frac{1}{2\phi} e^{-|x-\theta|/\phi} \quad \text{for } -\infty < x < \infty, \quad (25)$$

where $\phi > 0$.

No local copy of the parameters θ and ϕ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class LaplaceGen extends RandomVariateGen
```

Constructors

```
public LaplaceGen (RandomStream s, LaplaceDist dist)
```

Creates a new generator for the Laplace distribution `dist` and stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double theta, double phi)
```

Generates a new variate from the Laplace distribution with parameters $\theta = \text{theta}$ and $\phi = \text{phi}$, using stream `s`.

LogisticGen

This class implements random variate generators for the *logistic* distribution. Its parameters are α and $\lambda > 0$. Its density function is

$$f(x) = \frac{\lambda e^{-\lambda(x-\alpha)}}{(1 + e^{-\lambda(x-\alpha)})^2} \quad \text{for } -\infty < x < \infty. \quad (26)$$

No local copy of the parameters α and λ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class LogisticGen extends RandomVariateGen
```

Constructors

```
public LogisticGen (RandomStream s, LogisticDist dist)
    Creates a new generator for the logistic distribution dist and stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double alpha, double lambda)
    Generates a new variate from the logistic distribution with parameters  $\alpha = \text{alpha}$  and  $\lambda = \text{lambda}$ , using stream s.
```

LoglogisticGen

This class implements random variate generators for the *Log-Logistic* distribution with shape parameter $\alpha > 0$ and scale parameter $\beta > 0$. The density function of this distribution is

$$f(x) = \frac{\alpha(x/\beta)^{\alpha-1}}{\beta[1 + (x/\beta)^\alpha]^2} \quad \text{for } x > 0. \quad (27)$$

```
package umontreal.iro.lecuyer.randvar;
public class LoglogisticGen extends RandomVariateGen
```

Constructors

```
public LoglogisticGen (RandomStream s, LoglogisticDist dist)
    Creates a new generator for the distribution dist, using stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double alpha, double beta)
    Generates a variate from the Log-Logistic distribution with shape parameter  $\alpha > 0$  and scale parameter  $\beta > 0$ .
```

LognormalGen

This class implements methods for generating random variates from the *lognormal* distribution. Its density is

$$f(x) = \frac{1}{\sqrt{2\pi\sigma x}} e^{-(\ln(x)-\mu)^2/(2\sigma^2)} \quad \text{for } x > 0, \quad (28)$$

where $\sigma > 0$.

No local copy of the parameters μ and σ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the lognormal distribution object. One can also generate a lognormal random variate X via

```
X = Math.exp (NormalGen.nextDouble (s, mu, sigma)),
```

in which `NormalGen` can actually be replaced by any subclass of `NormalGen`.

```
package umontreal.iro.lecuyer.randvar;
public class LognormalGen extends RandomVariateGen
```

Constructors

```
public LognormalGen (RandomStream s, LognormalDist dist)
```

Create a random variate generator for the lognormal distribution `dist` and stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double mu, double sigma)
```

Generates a new variate from the *lognormal* distribution with parameters $\mu = \text{mu}$ and $\sigma = \text{sigma}$, using stream `s`.

LognormalSpecialGen

Implements methods for generating random variates from the *lognormal* distribution using an arbitrary normal random variate generator. The (non-static) `nextDouble` method calls the `nextDouble` method of the normal generator and takes the exponential of the result.

```
package umontreal.iro.lecuyer.randvar;  
public class LognormalSpecialGen extends RandomVariateGen
```

Constructors

```
public LognormalSpecialGen (NormalGen g)
```

Create a lognormal random variate generator using the normal generator `g` and with the same parameters.

NormalGen

This class implements methods for generating random variates from the *normal* distribution $N(\mu, \sigma)$. It has mean μ and variance σ^2 , where $\sigma > 0$. Its density function is

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/(2\sigma^2)} \quad (29)$$

No local copy of the parameters α and λ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class NormalGen extends RandomVariateGen
```

Constructors

```
public NormalGen (RandomStream s, NormalDist dist)
```

Creates a random variate generator for the normal distribution `dist` and stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double mu, double sigma)
```

Generates a variate from the normal distribution with parameters $\mu = \text{mu}$ and $\sigma = \text{sigma}$, using stream `s`.

NormalACRGen

This class implements *normal* random variate generators using the *acceptance-complement ratio* method [15]. For all the methods, the code was taken from UNURAN [24].

A local copy of the parameters μ and σ is maintained in this class.

```
packageumontreal.iro.lecuyer.randvar;  
publicclassNormalACRGenextendsNormalGen
```

Constructors

```
publicNormalACRGen(RandomStream s, NormalDist dist)
```

Creates a random variate generator for the normal distribution `dist` and stream `s`.

NormalBoxMullerGen

This class implements *normal* random variate generators using the *Box-Muller* method from [7]. Since the method generates two variates at a time, the second variate is returned upon the next call to the `nextDouble`.

A local copy of the parameters μ and σ is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;  
public class NormalBoxMullerGen extends NormalGen
```

Constructors

```
public NormalBoxMullerGen (RandomStream s, NormalDist dist)
```

Creates a random variate generator for the normal distribution `dist` and stream `s`.

NormalPolarGen

This class implements *normal* random variate generators using the *polar method with rejection* [25]. Since the method generates two variates at a time, the second variate is returned upon the next call to `nextDouble`.

A local copy of the parameters μ and σ is maintained in this class.

```
package umontreal.iro.lecuyer.randvar;  
public class NormalPolarGen extends NormalGen
```

Constructors

```
public NormalPolarGen (RandomStream s, NormalDist dist)
```

Creates a random variate generator for the normal distribution `dist` and stream `s`.

NormalKindermannRamageGen

This class implements *normal* random variate generators using the *Kindermann-Ramage* method [22]. The code was taken from UNURAN [24].

A local copy of the parameters μ and σ is maintained in this class.

```
packageumontreal.iro.lecuyer.randvar;  
publicclassNormalKindermannRamageGenextendsNormalGen
```

Constructors

```
publicNormalKindermannRamageGen(RandomStream s, NormalDist dist)  
    Creates a random variate generator for the normal distribution dist and stream s.
```

ParetoGen

This class implements random variate generators for one of the *Pareto* distributions, with parameters $\alpha > 0$ and $\beta > 0$. Its density function is

$$f(x) = \begin{cases} \alpha\beta^\alpha x^{-(\alpha+1)} & \text{for } x > \beta \\ 0 & \text{for } x \leq \beta \end{cases} \quad (30)$$

The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
packageumontreal.iro.lecuyer.randvar;
public class ParetoGen extends RandomVariateGen
```

Constructors

```
public ParetoGen (RandomStream s, ParetoDist dist)
```

Creates a new generator for the Pareto distribution `dist` and stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double alpha, double beta)
```

Generates a new variate from the Pareto distribution with parameters $\alpha = \text{alpha}$ and $\beta = \text{beta}$, using stream `s`.

Pearson5Gen

This class implements random variate generators for the *Pearson type V* distribution with shape parameter $\alpha > 0$ and scale parameter $\beta > 0$. The density function of this distribution is

$$f(x) = \begin{cases} \frac{x^{-(\alpha+1)} e^{-\beta/x}}{\beta^{-\alpha} \Gamma(\alpha)} & \text{for } x > 0 \\ 0 & \text{otherwise,} \end{cases} \quad (31)$$

where Γ is the gamma function.

```
package umontreal.iro.lecuyer.randvar;
```

```
public class Pearson5Gen extends RandomVariateGen
```

Constructors

```
public Pearson5Gen (RandomStream s, Pearson5Dist dist)
```

Creates a new generator for the distribution `dist`, using stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double alpha, double beta)
```

Generates a variate from the Pearson V distribution with shape parameter $\alpha > 0$ and scale parameter $\beta > 0$.

Pearson6Gen

This class implements random variate generators for the *Pearson type VI* distribution with shape parameters $\alpha_1 > 0$ and $\alpha_2 > 0$, and scale parameter $\beta > 0$. The density function of this distribution is

$$f(x) = \begin{cases} \frac{(x/\beta)^{\alpha_1-1}}{\beta\mathcal{B}(\alpha_1, \alpha_2)(1+x/\beta)^{\alpha_1+\alpha_2}} & \text{for } x > 0, \\ 0 & \text{otherwise,} \end{cases} \quad (32)$$

where \mathcal{B} is the beta function.

```
package umontreal.iro.lecuyer.randvar;
```

```
public class Pearson6Gen extends RandomVariateGen
```

Constructors

```
public Pearson6Gen (RandomStream s, Pearson6Dist dist)
```

Creates a new generator for the distribution `dist`, using stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double alpha1,
                                double alpha2, double beta)
```

Generates a variate from the Pearson VI distribution with shape parameters $\alpha_1 > 0$ and $\alpha_2 > 0$, and scale parameter $\beta > 0$.

StudentGen

This class implements methods for generating random variates from the *Student* distribution with $n > 0$ degrees of freedom. Its density function is

$$f(x) = \frac{\Gamma((n+1)/2)}{\Gamma(n/2)\sqrt{\pi n}} \left[1 + \frac{x^2}{n}\right]^{-(n+1)/2} \quad \text{for } -\infty < x < \infty, \quad (33)$$

where $\Gamma(x)$ is the gamma function defined in (19).

No local copy of the parameter n is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class StudentGen extends RandomVariateGen
```

Constructors

```
public StudentGen (RandomStream s, StudentDist dist)
    Creates a new generator for the Student distribution dist and stream s.
```

Methods

```
public static double nextDouble (RandomStream s, int n)
    Generates a new variate from the Student distribution with  $n = n$  degrees of freedom, using stream s.
```

StudentPolarGen

This class implements *Student* random variate generators using the *polar* method of [4]. The code is adapted from UNURAN (see [24]).

The non-static `nextDouble` method generates two variates at a time and the second one is saved for the next call. A pair of variates is generated every second call. In the static case, two variates are generated per call but only the first one is returned and the second is discarded.

```
packageumontreal.iro.lecuyer.randvar;  
publicclassStudentPolarGenextendsStudentGen
```

Constructors

```
publicStudentPolarGen(RandomStreams,StudentDistdist)
```

Creates a new generator for the Student distribution `dist` and stream `s`.

TriangularGen

This class implements random variate generators for the *triangular* distribution. Its density is

$$f(x) = \begin{cases} \frac{2(x-a)}{(b-a)(m-a)} & \text{for } a \leq x \leq m, \\ \frac{2(b-x)}{(b-a)(b-m)} & \text{for } m \leq x \leq b, \\ 0 & \text{elsewhere,} \end{cases} \quad (34)$$

where $a \leq m \leq b$ (see, e.g., [23]).

The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class TriangularGen extends RandomVariateGen
```

Constructors

```
public TriangularGen (RandomStream s, TriangularDist dist)
    Creates a new generator for the triangular distribution dist and stream s.
```

Methods

```
public static double nextDouble (RandomStream s, double a,
                                double b, double m)
    Generates a new variate from the triangular distribution with parameters  $a = \mathbf{a}$ ,  $b = \mathbf{b}$  and  $m = \mathbf{m}$  and stream s, using inversion.
```

UniformGen

This class implements random variate generators for the (continuous) *uniform* distribution over the interval (a, b) , where a and b are real numbers with $a < b$. The density is

$$f(x) = 1/(b - a) \quad \text{for } a \leq x \leq b. \quad (35)$$

The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class UniformGen extends RandomVariateGen
```

Constructors

```
public UniformGen (RandomStream s, UniformDist dist)
```

Creates a new generator for the uniform distribution `dist` and stream `s`.

Methods

```
static public double nextDouble (RandomStream s, double a, double b)
```

Generates a new uniform random variate over the interval (a, b) by inversion, using stream `s`.

WeibullGen

This class implements random variate generators for the *Weibull* distribution. Its density is

$$f(x) = \alpha \lambda^\alpha (x - \delta)^{\alpha-1} \exp[-(\lambda(x - \delta))^\alpha] \quad \text{for } x > \delta, \quad (36)$$

and $f(x) = 0$ elsewhere, where $\alpha > 0$, and $\lambda > 0$.

No local copy of the parameters λ and δ is maintained in this class. The (non-static) `nextDouble` method simply calls `inverseF` on the distribution.

```
package umontreal.iro.lecuyer.randvar;
public class WeibullGen extends RandomVariateGen
```

Constructors

```
public WeibullGen (RandomStream s, WeibullDist dist)
```

Creates a new generator for the Weibull distribution `dist` and stream `s`.

Methods

```
public static double nextDouble (RandomStream s, double alpha,
                                double lambda, double delta)
```

Uses inversion to generate a new variate from the Weibull distribution with parameters $\alpha = \text{alpha}$, $\lambda = \text{lambda}$, and $\delta = \text{delta}$, using stream `s`.

UnuranContinuous

This class permits one to create continuous univariate distribution using UNURAN via its string API.

```
package umontreal.iro.lecuyer.randvar;  
public class UnuranContinuous extends RandomVariateGen
```

Constructors

```
public UnuranContinuous (RandomStream s, String genStr)  
    Same as UnuranContinuous(s, s, genStr).
```

```
public UnuranContinuous (RandomStream s, RandomStream aux,  
                        String genStr)
```

Constructs a new continuous random number generator using the UNURAN generator specification string `genStr`, main stream `s`, and auxiliary stream `aux`.

Methods

```
public RandomStream getAuxStream()  
    Returns the auxiliary random number stream.
```

UnuranDiscreteInt

This class permits one to create a discrete univariate distribution using UNURAN via its string API.

```
package umontreal.iro.lecuyer.randvar;  
public class UnuranDiscreteInt extends RandomVariateGenInt
```

Constructors

```
public UnuranDiscreteInt (RandomStream s, String genStr)  
    Same as UnuranDiscreteInt (s, s, genStr).
```

```
public UnuranDiscreteInt (RandomStream s, RandomStream aux,  
                          String genStr)
```

Constructs a new discrete random number generator using the UNURAN generator specification string `genStr`, main stream `s`, and auxiliary stream `aux`.

Methods

```
public RandomStream getAuxStream()  
    Returns the auxiliary random number stream.
```

UnuranEmpirical

This class permits one to create generators for empirical and quasi-empirical univariate distributions using UNURAN via its string interface. The empirical data can be read from a file, from an array, or simply encoded into the generator specification string. When reading from a file or an array, the generator specification string must *not* contain a distribution specification string.

```
package umontreal.iro.lecuyer.randvar;
public class UnuranEmpirical extends RandomVariateGen
```

Constructors

```
public UnuranEmpirical (RandomStream s, String genStr)
```

Constructs a new empirical univariate generator using the specification string `genStr` and stream `s`.

```
public UnuranEmpirical (RandomStream s, RandomStream aux, String genStr)
```

Constructs a new empirical univariate generator using the specification string `genStr`, with main stream `s` and auxiliary stream `aux`.

```
public UnuranEmpirical (RandomStream s, PiecewiseLinearEmpiricalDist dist,
                        String genStr)
```

Same as `UnuranEmpirical(s, s, dist, genStr)`.

```
public UnuranEmpirical (RandomStream s, RandomStream aux,
                        PiecewiseLinearEmpiricalDist dist, String genStr)
```

Same as `UnuranEmpirical(s, aux, genStr)`, but reading the observations from the empirical distribution `dist`. The `genStr` argument must not contain a distribution part because the distribution will be generated from the input stream reader.

Methods

```
public RandomStream getAuxStream()
```

Returns the auxiliary random number stream.

UnuranException

This type of unchecked exception is thrown when an error occurs *inside* the UNURAN package. Usually, such an exception will come from the native side.

```
package umontreal.iro.lecuyer.randvar;  
public class UnuranException extends RuntimeException
```

Constructors

```
public UnuranException()
```

Constructs a new generic UNURAN exception.

```
public UnuranException (String message)
```

Constructs a UNURAN exception with the error message `message`

References

- [1] J. H. Ahrens and U. Dieter. Computer methods for sampling from gamma, beta, poisson and binomial distributions. *Computing*, 12:223–246, 1972.
- [2] J. H. Ahrens and U. Dieter. Computer generation of poisson deviates from modified normal distributions. *ACM Trans. Math. Software*, 8:163–179, 1982.
- [3] J. H. Ahrens and U. Dieter. Generating gamma variates by a modified rejection technique. *Communications of the ACM*, 25:47–54, 1982.
- [4] R. W. Bailey. Polar generation of random variates with the t -distribution. *Mathematics of Computation*, 62(206):779–781, 1994.
- [5] A. Berlinet and L. Devroye. A comparison of kernel density estimates. *Publications de l'Institut de Statistique de l'Université de Paris*, 38(3):3–59, 1994. available at <http://cgm.cs.mcgill.ca/~luc/np.html>.
- [6] D. J. Best. A simple algorithm for the computer generation of random samples from a Student's t or symmetric beta distribution. In L. C. A. Corsten and J. Hermans, editors, *COMPSTAT 1978: Proceedings in Computational statistics*, pages 341–347, Vienna, 1978. Physica-Verlag.
- [7] G. E. P. Box and M. E. Muller. A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29:610–611, 1958.
- [8] R. Cao, A. Cuevas, and W. González-Manteiga. A comparative study of several smoothing methods for density estimation. *Computational Statistics and Data Analysis*, 17:153–176, 1994.
- [9] R. C. H. Cheng. The generation of gamma variables with non-integral shape parameter. *Applied Statistics*, 26:71–75, 1977.
- [10] R. C. H. Cheng. Generating beta variates with nonintegral shape parameters. *Communications of the ACM*, 21:317–322, 1978.
- [11] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY, 1986.
- [12] L. Devroye. Random variate generation in one line of code. In *Proceedings of the 1996 Winter Simulation Conference*, pages 265–271. IEEE Press, 1996.
- [13] L. Devroye and L. Györfi. *Nonparametric Density Estimation: The L_1 View*. John Wiley, New York, NY, 1985.
- [14] J. E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, New York, NY, 1998.

- [15] W. Hörmann and G. Derflinger. The ACR method for generating normal random variables. *OR Spektrum*, 12:181–185, 1990.
- [16] W. Hörmann and J. Leydold. Automatic random variate generation for simulation input. In J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 675–682, Piscataway, NJ, Dec 2000. IEEE Press.
- [17] W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation*. Springer-Verlag, Berlin, 2004.
- [18] N. L. Johnson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, volume 1. Wiley, 2nd edition, 1994.
- [19] N. L. Johnson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, volume 2. Wiley, 2nd edition, 1995.
- [20] V. Kachitvichyanukul and B. Schmeiser. Computer generation of hypergeometric random variates. *J. Statist. Comput. Simul.*, 22:127–145, 1985.
- [21] A. W. Kemp. Efficient generation of logarithmically distributed pseudo-random variables. *Applied Statistics*, 30:249–253, 1981.
- [22] A. J. Kinderman and J. G. Ramage. Computer generation of normal random variables. *Journal of the American Statistical Association*, 71:893–898, 1976.
- [23] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, NY, third edition, 2000.
- [24] J. Leydold and W. Hörmann. *UNURAN—A Library for Universal Non-Uniform Random Number Generators*, 2002. Available at <http://statistik.wu-wien.ac.at/unuran>.
- [25] G. Marsaglia. Improving the polar method for generating a pair of random variables. Technical report, Boeing Scientific Research Laboratory, Seattle, Washington, 1962.
- [26] H. Sakasegawa. Stratified rejection and squeeze method for generating beta random numbers. *Annals of the Institute of Mathematical Statistics*, 35B:291–302, 1983.
- [27] B. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London, 1986.
- [28] E. Stadlober and H. Zechner. Generating beta variates via patchwork rejection. *Computing*, 50:1–18, 1993.
- [29] G. Ulrich. Computer generation of distributions on the m-sphere. *Applied Statistics*, 33:158–163, 1984.