SSJ User's Guide

Package `randvar`

Generating Non-Uniform Random Numbers

Version: February 15, 2012

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

Contents

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.

The generic classes `RandomVariateGen` and `RandomVariateGenInt` permit one to construct a random variate generator from a random stream and an arbitrary distribution (see interface `Distribution`). To generate random variates by inversion from an arbitrary distribution over the real numbers using a given random stream, it suffices to construct a `RandomVariateGen` object with the desired (previously created) `Distribution` and `RandomStream` objects, and then call its `nextDouble` method as many times as needed. For discrete distributions over the integers, one can construct a `RandomVariateGenInt` object containing the desired `DiscreteDistributionInt` and `RandomStream`, and call its `nextInt` method. By default, these generators simply call the `inverseF` method from the specified distribution object. These two classes suffice as long as we are willing to use inversion. Here is a simple example in which we create three parallel streams of normal random variates using inversion.

Listing 1: Using three parallel streams of random normal variates

```
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;

public class normaltest
{
   public static void main (String[] args) {
      // Create 3 parallel streams of random numbers
      RandomStream stream1 = new MRG31k3p();
      RandomStream stream2 = new MRG31k3p();
      RandomStream stream3 = new MRG31k3p();

      // Create 3 parallel streams of normal random variates
      RandomVariateGen gen1 = new NormalGen (stream1);
      RandomVariateGen gen2 = new NormalGen (stream2);
      RandomVariateGen gen3 = new NormalGen (stream3);

      final int n = 5;
      genere (gen1, n);
      genere (gen2, n);
      genere (gen3, n);
   }

   private static void genere (RandomVariateGen gen, int n) {
      double u;
      for (int i = 0; i < n; i++) {
         u = gen.nextDouble();
         System.out.printf ("%12.6f%n", u);
      }
      System.out.println ("----------------------");
```

```
    }
}
```

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, five different subclasses implement normal random variate generators, using five different methods. One of them, the class `NormalGen`, extends `RandomVariateGen` directly and provides normal random variate generators based on inversion, so it does the same thing as using `RandomVariateGen` with the normal distribution. The others are subclasses of `NormalGen`; they implement various non-inversion normal variate generation methods. To generate random variates with a specific method, it suffices to construct an object 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.

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 [28]. 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.

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 [29]. 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.

`public String toString ()`

Returns a `String` containing information about the current 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.

`public DiscreteDistributionInt getDistribution()`

Returns the `DiscreteDistributionInt` used by this generator.

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 (by calling `setCachedValues`), 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.

InverseFromDensityGen

Implements a method for generating random variates by numerical inversion of an *arbitrary continuous* distribution when only the probability density is known [12]. The cumulative probabilities (cdf) are pre-computed by numerical quadrature of the density using Gauss-Lobatto integration over suitably small intervals to satisfy the required precision, and these values are kept in tables. Then the algorithm uses polynomial interpolation over the tabulated values to get the inverse cdf. The user can select the desired precision and the degree of the interpolating polynomials.

The algorithm may fail for some distributions for which the density becomes infinite at a point (for ex. the Gamma and the Beta distributions with $\alpha < 1$) if one requires too high a precision (a too small `eps`, for ex. $\epsilon \sim 10^{-15}$). However, it should work also for continuous densities with finite discontinuities.

While the setup time is relatively slow, the generation of random variables is extremely fast and practically independent of the required precision and of the specific distribution. The following table shows the time needed (in seconds) to generate 10^8 random numbers using inversion from a given class, then the numerical inversion with Gauss-Lobatto integration implemented here, and finally the speed ratios between the two methods. The speed ratio is the speed of the latter over the former. Thus for the beta distribution with parameters (5, 500), generating random variables with the Gauss-Lobatto integration implemented in this class is more than 1700 times faster than using inversion from the `BetaDist` class. These tests were made on a machine with processor AMD Athlon 4000, running Red Hat Linux, with clock speed at 2403 MHz.

Distribution	Inversion	Gauss-Lobatto	speed ratio
`NormalDist(10.5, 5)`	9.19	8.89	1.03
`ExponentialDist(5)`	17.72	8.82	2.0
`CauchyDist(10.5, 5)`	18.30	8.81	2.1
`BetaSymmetricalDist(10.5)`	242.80	8.85	27.4
`GammaDist(55)`	899.50	8.89	101
`ChiSquareNoncentralDist(10.5, 5)`	5326.90	8.85	602
`BetaDist(5, 500)`	15469.10	8.86	1746

The following table gives the time (in sec.) needed to create an object (setup time) and to generate one random variable for this class compared to the same for the inversion method specific to each class, and the ratios of the times (init + one random variable) of the two methods. For inversion, we initialized 10^8 times; for this class, we initialized 10^4 times.

Distribution	Inversion 10^8 init	Gauss-Lobatto 10^4 init	time ratio for 1 init
`NormalDist(10.5, 5)`	5.30	38.29	26426
`ExponentialDist(5)`	3.98	27.05	12466
`CauchyDist(10.5, 5)`	5.05	58.39	25007
`BetaSymmetricalDist(10.5)`	90.66	68.33	2049
`GammaDist(55)`	13.15	58.34	639
`ChiSquareNoncentralDist(10.5, 5)`	190.48	248.98	451
`BetaDist(5, 500)`	63.60	116.57	75

If only a few random variables are needed, then using this class is not efficient because of the slow set-up. But if one wants to generate large samples from the same distribution with fixed parameters, then this class will be very efficient. The following table gives the number of random variables generated beyond which, using this class will be worthwhile.

Distribution	number of generated variables
`NormalDist(10.5, 5)`	41665
`ExponentialDist(5)`	15266
`CauchyDist(10.5, 5)`	31907
`BetaSymmetricalDist(10.5)`	2814
`GammaDist(55)`	649
`ChiSquareNoncentralDist(10.5, 5)`	467
`BetaDist(5, 500)`	75

Thus, for example, if one needs to generate less than 15266 exponential random variables, then using the `InverseFromDensityGen` class is not wortwhile: it will be faster to use inversion from the `ExponentialGen` class.

```
package umontreal.iro.lecuyer.randvar;
   import umontreal.iro.lecuyer.functions.MathFunction;
   import umontreal.iro.lecuyer.rng.RandomStream;
   import umontreal.iro.lecuyer.probdist.ContinuousDistribution;


public class InverseFromDensityGen extends RandomVariateGen
```

Constructors

```
public InverseFromDensityGen (RandomStream stream,
                              ContinuousDistribution dis,
                              double xc, double eps, int order)
```

Creates a new generator for the *continuous* distribution `dis`, using stream `stream`. `dis` must have a well-defined density method; its other methods are unused. For a non-standard distribution `dis`, the user may wish to set the left and the right boundaries between which the density is non-zero by calling methods `setXinf` and `setXsup` of `dis`, for better efficiency. Argument `xc` can be the mean, the mode or any other x for which the density is relatively large. The u-resolution `eps` is the desired absolute error in the CDF, and `order` is the degree of the Newton interpolating polynomial over each interval. An `order` of 3 or 5, and an `eps` of 10^{-6} to 10^{-12} are usually good choices. Restrictions: $3 \le$ `order` ≤ 12.

```
public InverseFromDensityGen (RandomStream stream, MathFunction dens,
                              double xc, double eps, int order,
                              double xleft, double xright)
```

Creates a new generator from the *continuous* probability density `dens`. The left and the right boundaries of the density are `xleft` and `xright` (the density is 0 outside the interval `[xleft, xright]`). See the description of the other constructor.

Methods

```
public double nextDouble()
```

Generates a new random variate.

```
public double getXc()
```

Returns the `xc` given in the constructor.

```
public double getEpsilon()
```

Returns the u-resolution `eps`.

```
public int getOrder()
```

Returns the order of the interpolating polynomial.

BernoulliGen

This class implements random variate generators for the *Bernoulli* distribution (see class `BernoulliDist`).

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

public class BernoulliGen extends RandomVariateGenInt
```

Constructors

```
public BernoulliGen (RandomStream s, double p)
```

Creates a Bernoulli random variate generator with parameter p, using stream `s`.

```
public BernoulliGen (RandomStream s, BernoulliDist dist)
```

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

Methods

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

Generates a new integer from the *Bernoulli* distribution with parameter $p =$ `p`, using the given stream `s`.

```
public double getP()
```

Returns the parameter p of this object.

BinomialGen

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

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

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

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, int n, double p)
```

Creates a binomial random variate generator with parameters n and p, using stream `s`.

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

Creates a 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`.

```
public int getN()
```

Returns the parameter n of this object.

```
public double getP()
```

Returns the parameter p of this object.

BinomialConvolutionGen

Implements binomial random variate generators using the convolution method. This method generates n Bernouilli 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.

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

public class BinomialConvolutionGen extends BinomialGen
```

Constructors

```
public BinomialConvolutionGen (RandomStream s, int n, double p)
```

Creates a *binomial* random variate generator with parameters n and p, using stream `s`.

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

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

GeometricGen

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

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

where $0 \le p \le 1$. 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, double p)
```

Creates a geometric random variate generator with parameter p, using stream `s`.

```
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 *geometric* random variate with parameter p = `p`, using stream `s`, by inversion.

```
public double getP()
```

Returns the parameter p of this object.

HypergeometricGen

This class implements random variate generators for the *hypergeometric* distribution. Its mass function is (see, e.g., [17, page 101])

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

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

The generation method is inversion using the chop-down algorithm [23]

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

public class HypergeometricGen extends RandomVariateGenInt
```

Constructors

`public HypergeometricGen (RandomStream s, int m, int l, int k)`

Creates a hypergeometric generator with parameters $m =$ `m`, $l =$ `l` and $k =$ `k`, using stream `s`.

`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`.

`public int getM()`

Returns the m associated with this object.

`public int getL()`

Returns the l associated with this object.

`public int getK()`

Returns the k associated with this object.

LogarithmicGen

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

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

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 [25]. The threshold θ_0 can be specified when invoking the constructor. Its default value is $\theta_0 = 0.96$, as suggested in [25].

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, double theta)
```

Creates a logarithmic random variate generator with parameters $\theta =$ `theta` and default value $\theta_0 = 0.96$, using stream `s`.

```
public LogarithmicGen (RandomStream s, double theta, double theta0)
```

Creates a logarithmic random variate generator with parameters $\theta =$ `theta` and $\theta_0 =$ `theta0`, using stream `s`.

```
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 $\theta_0 =$ `theta0`.

Methods

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

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

```
public double getTheta()
```

Returns the θ associated with this object.

```
public double getTheta0()
```

Returns the θ_0 associated with this object.

NegativeBinomialGen

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

$$p(x) = \frac{\Gamma(\gamma + x)}{x!\,\Gamma(\gamma)}\, p^\gamma (1-p)^x, \qquad \text{for } x = 0, 1, 2, \ldots \tag{5}$$

where Γ is the gamma function, $\gamma > 0$ and $0 \le p \le 1$. No local copy of the parameters γ 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, double gamma, double p)
```

Creates a negative binomial random variate generator with parameters $\gamma =$ `gamma` and p, using stream `s`.

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

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

Methods

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

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

```
public double getGamma()
```

Returns the parameter γ of this object.

```
public double getP()
```

Returns the parameter p of this object.

PascalGen

Implements Pascal random variate generators, which is a special case of the negative binomial generator with parameter γ equal to a positive integer. See `NegativeBinomialGen` for a description.

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

public class PascalGen extends RandomVariateGenInt
```

Constructors

`public PascalGen (RandomStream s, int n, double p)`

Creates a Pascal random variate generator with parameters n and p, using stream `s`.

`public PascalGen (RandomStream s, PascalDist 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 *Pascal* distribution, with parameters $n =$ `n` and $p =$ `p`, using stream `s`.

`public int getN()`

Returns the parameter n of this object.

`public double getP()`

Returns the parameter p of this object.

PascalConvolutionGen

Implements *Pascal* random variate generators by the *convolution* method (see [28]). The method generates n geometric variates with probability p and adds them up.

The algorithm is slow if n is large.

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

public class PascalConvolutionGen extends PascalGen
```

Constructors

`public PascalConvolutionGen (RandomStream s, int n, double p)`

Creates a *Pascal* random variate generator with parameters n and p, using stream `s`.

`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

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

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

No local copy of the parameter $\lambda =$ `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
```

Constructors

```
public PoissonGen (RandomStream s, double lambda)
```

Creates a Poisson random variate generator with parameter $\lambda =$ `lambda`, using stream `s`.

```
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 $\lambda =$ `lambda`.

```
public double getLambda()
```

Returns the λ associated with this object.

PoissonTIACGen

This class implements random variate generators having the *Poisson* distribution (see `PoissonGen`). Uses the tabulated inversion combined with the acceptance complement (*TIAC*) method of [2]. The implementation is adapted from UNURAN [29].

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

```
public class PoissonTIACGen extends PoissonGen
```

Constructors

```
public PoissonTIACGen (RandomStream s, double lambda)
```

Creates a Poisson random variate generator with parameter λ = `lambda`, using stream `s`.

```
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

$$p(x) = \frac{1}{j - i + 1} \qquad \text{for } x = i, i + 1, \ldots, j \tag{7}$$

and 0 elsewhere.

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

public class UniformIntGen extends RandomVariateGenInt
```

Constructors

```
public UniformIntGen (RandomStream s, int i, int j)
```

Creates a uniform random variate generator over the integers in the closed interval $[i, j]$, using stream `s`.

```
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.

```
public int getI()
```

Returns the parameter i.

```
public int getJ()
```

Returns the parameter j.

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

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

and $f(x) = 0$ elsewhere, where $\Gamma(x)$ is the gamma function defined in (21).

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, double alpha, double beta,
                                double a, double b)
```

Creates a new beta generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval (`a`, `b`), using stream `s`.

```
public BetaGen (RandomStream s, double alpha, double beta)
```

Creates a new beta generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval $(0, 1)$, using stream `s`.

```
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 $\alpha =$ `alpha`, $\beta =$ `beta`, over the interval (a, b), using stream `s`.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getBeta()
```

Returns the parameter β of this object.

`public double getA()`

Returns the parameter a of this object.

`public double getB()`

Returns the parameter b of this object.

BetaRejectionLoglogisticGen

Implements *Beta* random variate generators using the rejection method with log-logistic envelopes from [11]. 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,
                                    double alpha, double beta)
```

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta` over the interval $(0, 1)$, 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.

```
public BetaRejectionLoglogisticGen (RandomStream s,
                                    double alpha, double beta)
```

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval $(0, 1)$, using stream `s`.

```
public BetaRejectionLoglogisticGen (RandomStream s, RandomStream aux,
          double alpha, double beta, double a, double b)
```

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta` over the interval (`a`, `b`), 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.

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

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval (`a`, `b`), using stream `s`.

```
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 [32, 34]. 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,
                                   double alpha, double beta)
```

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval $(0, 1)$, 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.

```
public BetaStratifiedRejectionGen (RandomStream s,
                                   double alpha, double beta)
```

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval $(0, 1)$, using stream `s`.

```
public BetaStratifiedRejectionGen (RandomStream s, RandomStream aux,
       double alpha, double beta, double a, double b)
```

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval (`a`, `b`), 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.

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

Creates a beta random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, over the interval (`a`, `b`), using stream `s`.

```
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, double alpha)
```

Creates a new symmetrical beta generator with parameters $\alpha =$ `alpha`, over the interval $(0, 1)$, using stream `s`.

```
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 [36]. The method generates two uniform random variables $x \in [0, 1]$ and $y \in [-1, 1]$ until $x^2 + y^2 \le 1$. Then it returns

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

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 s1, RandomStream s2,
                                double alpha)
```

Creates a symmetrical beta random variate generator with parameter $\alpha =$ `alpha`, using stream `s1` to generate x and stream `s2` to generate y, as in (9) above. Restriction: $\alpha > 1/2$.

```
public BetaSymmetricalPolarGen (RandomStream s1, double alpha)
```

Creates a symmetrical beta random variate generator with parameter $\alpha =$ `alpha`, using stream `s1` to generate x and y, as in (9) above. Restriction: $\alpha > 1/2$.

```
public BetaSymmetricalPolarGen (RandomStream s1, RandomStream s2,
                                BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using stream `s1` to generate x and stream `s2` to generate y, as in (9) above. Restriction: `dist` must have $\alpha > 1/2$.

```
public BetaSymmetricalPolarGen (RandomStream s1,
                                BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using only one stream `s1`. 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 by Ulrich's polar method using 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 [14]:

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

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

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

public class BetaSymmetricalBestGen extends BetaSymmetricalGen
```

Constructors

```
public BetaSymmetricalBestGen (RandomStream s1, RandomStream s2,
                               RandomStream s3, double alpha)
```

Creates a symmetrical beta random variate generator with parameter $\alpha =$ `alpha`, using stream `s1` to generate U_1, stream `s2` to generate U_2 and stream `s3` to generate S, as given in equation (10).

```
public BetaSymmetricalBestGen (RandomStream s1, double alpha)
```

Creates a symmetrical beta random variate generator with parameter $\alpha =$ `alpha`, using only one stream `s1` to generate U_1, U_2, and S as given in equation (10).

```
public BetaSymmetricalBestGen (RandomStream s1, RandomStream s2,
                               RandomStream s3, BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using stream `s1` to generate U_1, stream `s2` to generate U_2 and stream `s3` to generate S as given in equation (10).

```
public BetaSymmetricalBestGen (RandomStream s1, BetaSymmetricalDist dist)
```

Creates a new generator for the distribution `dist`, using only one stream `s1`.

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., [21] p. 299):

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

where $\beta > 0$.

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, double alpha, double beta)
```

Creates a Cauchy random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, using stream `s`.

```
public CauchyGen (RandomStream s)
```

Creates a Cauchy random variate generator with parameters $\alpha = 0$ and $\beta = 1$, using stream `s`.

```
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 $\alpha =$ `alpha` and $\beta =$ `beta`, using stream `s`.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getBeta()
```

Returns the parameter β of this object.

ChiGen

This class implements random variate generators for the *chi* distribution. It has $\nu > 0$ degrees of freedom and its density function is (see [21], page 417)

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

where $\Gamma(x)$ is the gamma function defined in (21).

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, int nu)
```

Creates a *chi* random variate generator with $\nu =$ `nu` degrees of freedom, using stream `s`.

```
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 $\nu =$ `nu` degrees of freedom, using stream `s`.

```
public int getNu()
```

Returns the value of ν for this object.

ChiRatioOfUniformsGen

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

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

public class ChiRatioOfUniformsGen extends ChiGen
```

Constructors

`public ChiRatioOfUniformsGen (RandomStream s, int nu)`

Creates a *chi* random variate generator with $\nu =$ `nu` degrees of freedom, using stream `s`.

`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

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

where $\Gamma(x)$ is the gamma function defined in (21).

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, int n)
```

Creates a *chi square* random variate generator with n degrees of freedom, using stream `s`.

```
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`.

```
public int getN()
```

Returns the value of n for this object.

ChiSquareNoncentralGen

This class implements random variate generators for the *noncentral chi square* distribution with $\nu > 0$ degrees of freedom and noncentrality parameter $\lambda > 0$. See the definition in `ChiSquareNoncentralDist`.

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

public class ChiSquareNoncentralGen extends RandomVariateGen
```

Constructors

```
public ChiSquareNoncentralGen (RandomStream s, double nu, double lambda)
```

Creates a *noncentral chi square* random variate generator with `nu` $= \nu > 0$ degrees of freedom and noncentrality parameter `lambda` $= \lambda > 0$, using stream `s`.

```
public ChiSquareNoncentralGen (RandomStream s,
                               ChiSquareNoncentralDist dist)
```

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

Methods

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

Generates a new variate from the noncentral chi square distribution with `nu` $= \nu$ degrees of freedom and noncentrality parameter `lambda` $= \lambda$, using stream `s`.

```
public double getNu()
```

Returns the value of ν of this object.

```
public double getLambda()
```

Returns the value of λ for this object.

ChiSquareNoncentralGamGen

This class implements *noncentral chi square* random variate generators using the additive property of the noncentral chi square distribution [27]. It uses the following algorithm: generate a real $X \sim N(\sqrt{\lambda}, 1)$ from a normal distribution with variance 1, generate a real $Y \sim \Gamma((\nu-1)/2, 1/2)$ from a gamma distribution, then return X^2+Y. Here ν is the number of degrees of freedom and λ is the noncentrality parameter.

To generate the normal variates, one uses the fast *acceptance-complement ratio* method in [18] (see class `NormalACRGen`). To generate the gamma variates, one uses acceptance-rejection for $\alpha < 1$, and acceptance-complement for $\alpha \geq 1$, as proposed in [1, 3] (see class `GammaAcceptanceRejectionGen`).

This noncentral chi square generator is faster than the generator `ChiSquareNoncentral-PoisGen` on the next page of this guide. For small λ, it is nearly twice as fast. As λ increases, it is still faster but not as much.

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

public class ChiSquareNoncentralGamGen extends ChiSquareNoncentralGen
```

Constructor

```
public ChiSquareNoncentralGamGen (RandomStream stream,
                                  double nu, double lambda)
```

Creates a noncentral chi square random variate generator with with $\nu =$ `nu` degrees of freedom and noncentrality parameter $\lambda =$ `lambda` using stream `stream`, as described above.

Methods

```
public static double nextDouble (RandomStream stream,
                                 double nu, double lambda)
```

Generates a variate from the noncentral chi square distribution with parameters $\nu =$ `nu` and $\lambda =$ `lambda` using stream `stream`, as described above.

ChiSquareNoncentralPoisGen

This class implements *noncentral chi square* random variate generators using Poisson and central chi square generators. It uses the following algorithm: generate a random integer $J \sim \text{Poisson}(\lambda/2)$ from a Poisson distribution, generate a random real $X \sim \Gamma(j + \nu/2, 1/2)$ from a gamma distribution, then return X. Here ν is the number of degrees of freedom and λ is the noncentrality parameter.

To generate the Poisson variates, one uses tabulated inversion for $\lambda < 10$, and the acceptance complement method for $\lambda \geq 10$, as in [2] (see class `PoissonTIACGen`). To generate the gamma variates, one uses acceptance-rejection for $\alpha < 1$, and acceptance-complement for $\alpha \geq 1$, as proposed in [1, 3] (see class `GammaAcceptanceRejectionGen`).

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

public class ChiSquareNoncentralPoisGen extends ChiSquareNoncentralGen
```

Constructor

```
public ChiSquareNoncentralPoisGen (RandomStream stream,
                                   double nu, double lambda)
```

Creates a noncentral chi square random variate generator with $\nu =$ `nu` degrees of freedom and noncentrality parameter $\lambda =$ `lambda` using stream `stream`, as described above.

Methods

```
public static double nextDouble (RandomStream stream,
                                 double nu, double lambda)
```

Generates a variate from the noncentral chi square distribution with parameters $\nu =$ `nu` and $\lambda =$ `lambda` using stream `stream`, as described above.

ConstantGen

This class implements a random variate generator that returns a constant value. Its mass function is

$$p(x) = \begin{cases} 1, & \text{for } x = c, \\ 0, & \text{elsewhere.} \end{cases} \tag{14}$$

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

public class ConstantGen extends RandomVariateGen
```

Constructor

```
public ConstantGen (double val)
```

Constructs a new constant generator returning the given value `val`.

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.

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

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

public class ErlangGen extends GammaGen
```

Constructors

```
public ErlangGen (RandomStream s, int k, double lambda)
```

Creates an Erlang random variate generator with parameters `k` and λ = `lambda`, using stream `s`.

```
public ErlangGen (RandomStream s, int k)
```

Creates an Erlang random variate generator with parameters `k` and $\lambda = 1$, using stream `s`.

```
public ErlangGen (RandomStream s, ErlangDist dist)
```

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

Methods

```
public static double nextDouble (RandomStream s, int k, double lambda)
```

Generates a new variate from the Erlang distribution with parameters k = `k` and λ = `lambda`, using stream `s`.

```
public int getK()
```

Returns the parameter k of this object.

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.

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

public class ErlangConvolutionGen extends ErlangGen
```

Constructors

`public ErlangConvolutionGen (RandomStream s, int k, double lambda)`

Creates an Erlang random variate generator with parameters `k` and λ = `lambda`, using stream `s`.

`public ErlangConvolutionGen (RandomStream s, int k)`

Creates an Erlang random variate generator with parameters `k` and $\lambda = 1$, using stream `s`.

`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

$$f(x) = \lambda e^{-\lambda x} \qquad \text{for } x \geq 0, \tag{15}$$

where $\lambda > 0$.

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, double lambda)`

Creates an exponential random variate generator with parameter λ = `lambda`, using stream `s`.

`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 λ = `lambda`, using stream `s`.

`public double getLambda()`

Returns the λ associated with this object.

ExponentialInverseFromDensityGen

This class implements *exponential* random variate generators using numerical inversion of the *exponential* density as described in [12]. It makes use of the class `InverseDistFromDensity`. Generating exponential random variables by inversion usually requires the computation of a logarithm for each generated random number. Numerical inversion precomputes a set of tables that will speed up the generation of random variables. This is useful if one wants to generate a large number of random variables.

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

public class ExponentialInverseFromDensityGen extends ExponentialGen
```

Constructors

```
public ExponentialInverseFromDensityGen (RandomStream stream,
                                         double lambda,
                                         double ueps, int order)
```

Creates an exponential random variate generator with parameter $\lambda =$ `lambda`, using stream `stream`. It uses numerical inversion with precomputed tables. The u-resolution `ueps` is the desired absolute error in the `cdf`, and `order` is the degree of the Newton interpolating polynomial over each interval.

```
public ExponentialInverseFromDensityGen (RandomStream stream,
                                         ExponentialDist dist,
                                         double ueps, int order)
```

Similar to the above constructor, with the exponential distribution `dist`.

```
public ExponentialInverseFromDensityGen (RandomStream stream,
                                         InverseDistFromDensity dist)
```

Creates a new exponential generator using the *exponential* distribution `dist` and stream `stream`. `dist` may be obtained by calling method `getDistribution`, after using one of the other constructors to create the precomputed tables. This is useful when one needs many generators using the same exponential distribution (same λ). Precomputing tables for numerical inversion is costly; thus using only one set of tables for many generators is more efficient. The first `ExponentialInverseFromDensityGen` generator using the other constructors creates the precomputed tables. Then all other streams use this constructor with the same set of tables.

Methods

```
public double getUepsilon()
```

Returns the u-resolution `ueps`.

```
public int getOrder()
```

Returns the order of the interpolating polynomial.

ExtremeValueGen

This class has been replaced by `GumbelGen`.

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

$$f(x) = \lambda e^{-e^{-\lambda(x-\alpha)}-\lambda(x-\alpha)} \tag{16}$$

where $\lambda > 0$.

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

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

@Deprecated
public class ExtremeValueGen extends RandomVariateGen
```

Constructors

```
public ExtremeValueGen (RandomStream s, double alpha, double lambda)
```

Creates an *extreme value* random variate generator with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
public ExtremeValueGen (RandomStream s)
```

Creates an *extreme value* random variate generator with parameters $\alpha = 0$ and $\lambda = 1$, using stream `s`.

```
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 $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getLambda()
```

Returns the parameter λ of this object.

FatigueLifeGen

This class implements random variate generators for the *fatigue life* distribution [7] with location parameter μ, scale parameter β and shape parameter γ. The density function of this distribution is

$$f(x) = \left[\frac{\sqrt{(x-\mu)/\beta} + \sqrt{\beta/(x-\mu)}}{2\gamma(x-\mu)}\right] \phi\left(\frac{\sqrt{(x-\mu)/\beta} - \sqrt{\beta/(x-\mu)}}{\gamma}\right), \qquad x > \mu \qquad (17)$$

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, double mu, double beta,
                                       double gamma)
```

Creates a *fatigue life* random variate generator with parameters $\mu =$ `mu`, $\beta =$ `beta` and $\gamma =$ `gamma`, using stream `s`.

```
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 μ, scale parameter β and shape parameter γ.

```
public double getBeta()
```

Returns the parameter β of this object.

```
public double getGamma()
```

Returns the parameter γ of this object.

```
public double getMu()
```

Returns the parameter μ of this object.

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

$$f(x) = \frac{\Gamma((n+m)/2)n^{n/2}m^{m/2}}{\Gamma(n/2)\Gamma(m/2)}\frac{x^{(n-2)/2}}{(m+nx)^{(n+m)/2}}, \qquad \text{for } x > 0 \tag{18}$$

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

public class FisherFGen extends RandomVariateGen
```

Constructors

```
public FisherFGen (RandomStream s, int n, int m)
```

Creates a *Fisher F* random variate generator with n and m degrees of freedom, using stream `s`.

```
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`.

```
public int getN()
```

Returns the parameter n of this object.

```
public int getM()
```

Returns the parameter p of this object.

FNoncentralGen

This class implements random variate generators for the *noncentral* F-distribution. If X is a noncentral chi-square random variable with $\nu_1 > 0$ degrees of freedom and noncentrality parameter $\lambda > 0$, and Y is a chi-square random variable (statistically independent of X) with $\nu_2 > 0$ degrees of freedom, then

$$F' = \frac{X/\nu_1}{Y/\nu_2}$$

has a noncentral F-distribution.

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

public class FNoncentralGen extends RandomVariateGen
```

Constructors

```
public FNoncentralGen (ChiSquareNoncentralGen ncgen, ChiSquareGen cgen)
```

Creates a *noncentral-F* random variate generator using noncentral chi-square generator `ncgen` and chi-square generator `cgen`.

Methods

```
public void setChiSquareNoncentralGen (ChiSquareNoncentralGen ncgen)
```

Sets the noncentral chi-square generator to `ncgen`.

```
public void setChiSquareGen (ChiSquareGen cgen)
```

Sets the chi-square generator to `cgen`.

FoldedNormalGen

This class implements methods for generating random variates from the *folded normal* distribution with parameters $\mu \geq 0$ and $\sigma > 0$. The density is

$$f(x) = \phi\left(\frac{x-\mu}{\sigma}\right) + \phi\left(\frac{-x-\mu}{\sigma}\right) \qquad \text{for } x \geq 0, \tag{19}$$

where ϕ denotes the density function of a standard normal distribution.

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

public class FoldedNormalGen extends RandomVariateGen
```

Constructors

```
public FoldedNormalGen (RandomStream s, double mu, double sigma)
```

Creates a new *folded normal* generator with parameters $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

```
public FoldedNormalGen (RandomStream s, FoldedNormalDist 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 *folded normal* distribution with parameters $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

```
public double getMu()
```

Returns the parameter μ of this object.

```
public double getSigma()
```

Returns the parameter σ of this object.

FrechetGen

This class implements methods for generating random variates from the *Fréchet* distribution, with location parameter δ, scale parameter $\beta > 0$, and shape parameter $\alpha > 0$, where we use the notation $z = (x - \delta)/\beta$. It has density

$$f(x) = \frac{\alpha e^{-z^{-\alpha}}}{\beta z^{\alpha+1}}, \qquad \text{for } x > \delta.$$

The density is 0 for $x \le \delta$.

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

public class FrechetGen extends RandomVariateGen
```

Constructors

```
public FrechetGen (RandomStream s, double alpha)
```

Creates a *Fréchet* random number generator with $\alpha =$ `alpha`, $\beta = 1$ and $\delta = 0$ using stream `s`.

```
public FrechetGen (RandomStream s, double alpha, double beta,
                   double delta)
```

Creates a *Fréchet* random number generator with parameters $\alpha =$ `alpha`, $\beta =$ `beta` and δ = `delta` using stream `s`.

```
public FrechetGen (RandomStream s, FrechetDist dist)
```

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

Methods

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

Generates a new variate from the *Fréchet* distribution with parameters $\alpha =$ `alpha`, $\beta =$ `beta` and $\delta =$ `delta` using stream `s`.

```
public double getAlpha()
```

Returns the parameter α.

```
public double getBeta()
```

Returns the parameter β.

```
public double getDelta()
```

Returns the parameter δ.

GammaGen

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

$$f(x) = \lambda^\alpha x^{\alpha-1} e^{-\lambda x} / \Gamma(\alpha) \qquad \text{for } x > 0, \tag{20}$$

where Γ is the gamma function defined by

$$\Gamma(\alpha) = \int_0^\infty x^{\alpha-1} e^{-x}\, dx. \tag{21}$$

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, double alpha, double lambda)
```

Creates a gamma random variate generator with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
public GammaGen (RandomStream s, double alpha)
```

Creates a gamma random variate generator with parameters $\alpha =$ `alpha` and $\lambda = 1$, using stream `s`.

```
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 $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getLambda()
```

Returns the parameter λ of this object.

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, RandomStream aux,
                                    double alpha, double lambda)
```

Creates a gamma random variate generator with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, 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.

```
public GammaAcceptanceRejectionGen (RandomStream s,
                                    double alpha, double lambda)
```

Creates a gamma random variate generator with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
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.

```
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.

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 $\alpha =$ `alpha` and $\lambda =$ `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 [10]. 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, RandomStream aux,
                                     double alpha, double lambda)
```

Creates a gamma random variate generator with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, 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.

```
public GammaRejectionLoglogisticGen (RandomStream s,
                                     double alpha, double lambda)
```

Creates a gamma random variate generator with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
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.

```
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.

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 $\alpha =$ `alpha` and $\lambda =$ `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)`.

GumbelGen

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

$$f(x) = \frac{e^{-z}e^{-e^{-z}}}{|\beta|}, \qquad \text{for } -\infty < x < \infty, \tag{22}$$

where we use the notation $z = (x - \delta)/\beta$. The scale parameter β can be positive (for the Gumbel distribution) or negative (for the reverse Gumbel distribution), but not 0.

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

public class GumbelGen extends RandomVariateGen
```

Constructors

```
public GumbelGen (RandomStream s)
```

Creates a Gumbel random number generator with $\beta = 1$ and $\delta = 0$ using stream `s`.

```
public GumbelGen (RandomStream s, double beta, double delta)
```

Creates a Gumbel random number generator with parameters $\beta =$ `beta` and $\delta =$ `delta` using stream `s`.

```
public GumbelGen (RandomStream s, GumbelDist dist)
```

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

Methods

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

Generates a new variate from the Gumbel distribution with parameters $\beta =$ `beta` and $\delta =$ `delta` using stream `s`.

```
public double getBeta()
```

Returns the parameter β.

```
public double getDelta()
```

Returns the parameter δ.

HalfNormalGen

This class implements methods for generating random variates from the *half-normal* distribution with parameters μ and $\sigma > 0$. Its density is

$$
\begin{aligned}
f(x) &= \frac{1}{\sigma}\sqrt{\frac{2}{\pi}}\, e^{-(x-\mu)^2/2\sigma^2}, \qquad \text{for } x \ge \mu. \\
f(x) &= 0, \qquad \text{for } x < \mu.
\end{aligned}
$$

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

public class HalfNormalGen extends RandomVariateGen
```

Constructors

```
public HalfNormalGen (RandomStream s, double mu, double sigma)
```

Creates a new *half-normal* generator with parameters $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

```
public HalfNormalGen (RandomStream s, HalfNormalDist 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 *half-normal* distribution with parameters $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

```
public double getMu()
```

Returns the parameter μ of this object.

```
public double getSigma()
```

Returns the parameter σ of this object.

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

$$f(x) = \frac{1}{2\sigma} \operatorname{sech}\left(\frac{\pi}{2}\frac{(x-\mu)}{\sigma}\right), \qquad -\infty < x < \infty. \tag{23}$$

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

public class HyperbolicSecantGen extends RandomVariateGen
```

Constructors

`public HyperbolicSecantGen (RandomStream s, double mu, double sigma)`

Creates a *hyperbolic secant* random variate generator with parameters $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

`public HyperbolicSecantGen (RandomStream s)`

Creates a *hyperbolic secant* random variate generator with parameters $\mu = 0$ and $\sigma = 1$, using stream `s`.

`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 μ and scale parameter σ.

`public double getMu()`

Returns the parameter μ of this object.

`public double getSigma()`

Returns the parameter σ of this object.

HypoExponentialGen

This class implements random variate generators for the *hypoexponential* distribution (see classes `HypoExponentialDist` and `HypoExponentialDistQuick` in package `probdist` for the definition).

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

public class HypoExponentialGen extends RandomVariateGen
```

Constructors

```
public HypoExponentialGen (RandomStream stream, double[] lambda)
```

Creates a hypoexponential random variate generator with rates $\lambda_i =$ `lambda`$[i-1]$, $i = 1,\ldots,k$, using stream `stream`.

```
public HypoExponentialGen (RandomStream stream, HypoExponentialDist dist)
```

Creates a new generator for the hypoexponential distribution `dist` with stream `stream`.

Methods

```
public static double nextDouble (RandomStream stream, double[] lambda)
```

Uses inversion to generate a new hypoexponential variate with rates $\lambda_i =$ `lambda`$[i-1]$, $i = 1,\ldots,k$, using stream `stream`. The inversion uses a root-finding method and is very slow.

```
public double[] getLambda()
```

Returns the λ_i associated with this object.

```
public void setLambda (double[] lambda)
```

Sets the rates $\lambda_i =$ `lam`$[i-1]$, $i = 1,\ldots,k$ of this object.

InverseGammaGen

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

$$f(x) = \begin{cases} \dfrac{\beta^\alpha}{\Gamma(\alpha)} \dfrac{e^{-\beta/x}}{x^{\alpha+1}} & \text{for } x > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{24}$$

where Γ is the gamma function.

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

```
public class InverseGammaGen extends RandomVariateGen
```

Constructors

```
public InverseGammaGen (RandomStream s, double alpha, double beta)
```

Creates an inverse gamma random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, using stream `s`.

```
public InverseGammaGen (RandomStream s, double alpha)
```

Creates an inverse gamma random variate generator with parameters $\alpha =$ `alpha` and $\beta = 1$, using stream `s`.

```
public InverseGammaGen (RandomStream s, InverseGammaDist 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 inverse gamma distribution with shape parameter $\alpha > 0$ and scale parameter $\beta > 0$.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getBeta()
```

Returns the parameter β of this object.

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

$$f(x) = \sqrt{\frac{\lambda}{2\pi x^3}}\, e^{-\lambda(x-\mu)^2/(2\mu^2 x)} \qquad \text{for } x > 0. \tag{25}$$

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

public class InverseGaussianGen extends RandomVariateGen
```

Constructors

```
public InverseGaussianGen (RandomStream s, double mu, double lambda)
```

Creates an *inverse Gaussian* random variate generator with parameters $\mu =$ `mu` and $\lambda =$ `lambda`, using stream `s`.

```
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 $\lambda > 0$.

```
public double getMu()
```

Returns the parameter μ of this object.

```
public double getLambda()
```

Returns the parameter λ of this object.

InverseGaussianMSHGen

This class implements *inverse gaussian* random variate generators using the many-to-one transformation method of Michael, Schucany and Haas (MHS) [31, 15].

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

public class InverseGaussianMSHGen extends InverseGaussianGen
```

Constructors

```
public InverseGaussianMSHGen (RandomStream s, NormalGen sn,
                              double mu, double lambda)
```

Creates an *inverse gaussian* random variate generator with parameters $\mu =$ `mu` and $\lambda =$ `lambda`, using streams `s` and `sn`.

```
public InverseGaussianMSHGen (RandomStream s, NormalGen sn,
                              InverseGaussianDist dist)
```

Creates a new generator for the distribution `dist` using streams `s` and `sn`.

Methods

```
public static double nextDouble (RandomStream s, NormalGen sn,
                                 double mu, double lambda)
```

Generates a new variate from the *inverse gaussian* distribution with parameters $\mu =$ `mu` and $\lambda =$ `lambda`, using streams `s` and `sn`.

JohnsonSBGen

This class implements random variate generators for the *Johnson* S_B distribution.

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

```
public class JohnsonSBGen extends RandomVariateGen
```

Constructors

```
public JohnsonSBGen (RandomStream s, double gamma, double delta,
                     double xi, double lambda)
```

Creates a JohnsonSB random variate generator.

```
public JohnsonSBGen (RandomStream s, JohnsonSBDist dist)
```

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

Methods

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

Uses inversion to generate a new JohnsonSB variate, using stream `s`.

```
public double getGamma()
```

Returns the γ associated with this object.

```
public double getDelta()
```

Returns the δ associated with this object.

```
public double getXi()
```

Returns the ξ associated with this object.

```
public double getLambda()
```

Returns the λ associated with this object.

JohnsonSUGen

This class implements random variate generators for the *Johnson* S_U distribution.

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

public class JohnsonSUGen extends RandomVariateGen
```

Constructors

```
public JohnsonSUGen (RandomStream s, double gamma, double delta,
                        double xi, double lambda)
```

Creates a JohnsonSU random variate generator.

```
public JohnsonSUGen (RandomStream s, JohnsonSUDist dist)
```

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

Methods

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

Uses inversion to generate a new JohnsonSU variate, using stream `s`.

```
public double getGamma()
```

Returns the γ associated with this object.

```
public double getDelta()
```

Returns the δ associated with this object.

```
public double getXi()
```

Returns the ξ associated with this object.

```
public double getLambda()
```

Returns the λ associated with this object.

KernelDensityGen

This class implements random variate generators for distributions obtained via *kernel density estimation* methods from a set of n individual observations $x_1, \ldots, x_n$ [16, 13, 19, 20, 33]. 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

$$f_n(x) = \frac{1}{nh} \sum_{i=1}^{n} k((x - x_i)/h), \tag{26}$$

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, 9, 20, 33].

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 [33, 20], is $h = \alpha_k h_0$, where

$$h_0 = 1.36374 \min(s_n, q/1.34) n^{-1/5}, \tag{27}$$

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

$$\alpha_k = \left(\sigma_k^{-4} \int_{-\infty}^{\infty} k(x) dx \right)^{1/5} \tag{28}$$

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	`BetaSymmetricalDist(2, -1, 1)`	1.7188	1/5	1.000
triangular	`TriangularDist(-1, 1, 0)`	1.8882	1/6	0.986
Gaussian	`NormalDist()`	0.7764	1	0.951
boxcar	`UniformDist(-1, 1)`	1.3510	1/3	0.930
logistic	`LogisticDist()`	0.4340	3.2899	0.888
Student-t(3)	`StudentDist(3)`	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 [20]. 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 (27).

```
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 mecanism 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 [33].

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 $\sigma_k^2 =$ `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 [22, page 165])

$$f(x) = \frac{1}{2\beta} e^{-|x-\mu|/\beta} \qquad \text{for } -\infty < x < \infty, \tag{29}$$

where $\beta > 0$.

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, double mu, double beta)`

Creates a Laplace random variate generator with parameters $\mu =$ `mu` and $\beta =$ `beta`, using stream `s`.

`public LaplaceGen (RandomStream s)`

Creates a Laplace random variate generator with parameters $\mu = 0$ and $\beta = 1$, using stream `s`.

`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 mu, double beta)`

Generates a new variate from the Laplace distribution with parameters $\mu =$ `mu` and $\beta =$ `beta`, using stream `s`.

`public double getMu()`

Returns the parameter μ.

`public double getBeta()`

Returns the parameter β.

LogisticGen

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

$$f(x) = \frac{\lambda e^{-\lambda(x-\alpha)}}{(1 + e^{-\lambda(x-\alpha)})^2} \qquad \text{for } -\infty < x < \infty. \tag{30}$$

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, double alpha, double lambda)
```

Creates a logistic random variate generator with parameters $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
public LogisticGen (RandomStream s)
```

Creates a logistic random variate generator with parameters $\alpha = 0$ and $\lambda = 1$, using stream `s`.

```
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 $\alpha =$ `alpha` and $\lambda =$ `lambda`, using stream `s`.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getLambda()
```

Returns the parameter λ of this object.

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

$$f(x) = \frac{\alpha (x/\beta)^{\alpha-1}}{\beta[1 + (x/\beta)^\alpha]^2} \qquad \text{for } x > 0. \tag{31}$$

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

public class LoglogisticGen extends RandomVariateGen
```

Constructors

```
public LoglogisticGen (RandomStream s, double alpha, double beta)
```

Creates a log-logistic random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, using stream `s`.

```
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 $\beta > 0$.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getBeta()
```

Returns the parameter β of this object.

LognormalGen

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

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma x} e^{-(\ln(x)-\mu)^2/(2\sigma^2)} \qquad \text{for } x > 0, \tag{32}$$

where $\sigma > 0$.

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, double mu, double sigma)`

Creates a lognormal random variate generator with parameters $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

`public LognormalGen (RandomStream s)`

Creates a lognormal random variate generator with parameters $\mu = 0$ and $\sigma = 1$, using stream `s`.

`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 $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

`public double getMu()`

Returns the parameter μ of this object.

`public double getSigma()`

Returns the parameter σ of this object.

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.

NakagamiGen

This class implements random variate generators for the *Nakagami* distribution. See the definition in `NakagamiDist` of package `probdist`.

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

public class NakagamiGen extends RandomVariateGen
```

Constructors

```
public NakagamiGen (RandomStream s, double a, double lambda, double c)
```

Creates a new Nakagami generator with parameters $a =$ `a`, $\lambda =$ `lambda` and $c =$ `c`, using stream `s`.

```
public NakagamiGen (RandomStream s, NakagamiDist dist)
```

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

Methods

```
public static double nextDouble (RandomStream s, double a, double lambda,
                                 double c)
```

Generates a variate from the *Nakagami* distribution with parameters $a =$ `a`, $\lambda =$ `lambda` and $c =$ `c`, using stream `s`.

```
public double getA()
```

Returns the location parameter a of this object.

```
public double getLambda()
```

Returns the scale parameter λ of this object.

```
public double getC()
```

Returns the shape parameter c of this object.

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

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{(x-\mu)^2/(2\sigma^2)} \tag{33}$$

The `nextDouble` method simply calls `inverseF` on the distribution.

The following table gives the CPU time needed to generate 10^8 standard normal random variates using the different implementations available in SSJ. The first time is for a generator object (non-static method), and the second time is for the static method where no object is created. These tests were made on a machine with processor AMD Athlon 4000, running Red Hat Linux, with clock speed at 2403 MHz. The static method `nextDouble()` for `NormalBoxMullerGen` and `NormalPolarGen` uses only one number out of two that are generated; thus they are twice slower than the non-static method.

Generator	time in seconds (object)	time in seconds (static)
`NormalGen`	7.67	7.72
`NormalACRGen`	4.71	4.76
`NormalBoxMullerGen`	16.07	31.45
`NormalPolarGen`	7.31	13.74
`NormalKindermannRamageGen`	5.38	5.34

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

public class NormalGen extends RandomVariateGen
```

Constructors

```
public NormalGen (RandomStream s, double mu, double sigma)
```

Creates a normal random variate generator with mean `mu` and standard deviation `sigma`, using stream `s`.

```
public NormalGen (RandomStream s)
```

Creates a standard normal random variate generator with mean `0` and standard deviation `1`, using stream `s`.

```
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 $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `s`.

`public double getMu()`

Returns the parameter μ of this object.

`public double getSigma()`

Returns the parameter σ of this object.

NormalACRGen

This class implements *normal* random variate generators using the *acceptance-complement ratio* method [18]. For all the methods, the code was taken from UNURAN [29].

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

public class NormalACRGen extends NormalGen
```

Constructors

`public NormalACRGen (RandomStream s, double mu, double sigma)`

Creates a normal random variate generator with mean `mu` and standard deviation `sigma`, using stream `s`.

`public NormalACRGen (RandomStream s)`

Creates a standard normal random variate generator with mean `0` and standard deviation `1`, using stream `s`.

`public NormalACRGen (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 [8]. Since the method generates two variates at a time, the second variate is returned upon the next call to the `nextDouble`.

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

```
public class NormalBoxMullerGen extends NormalGen
```

Constructors

```
public NormalBoxMullerGen (RandomStream s, double mu, double sigma)
```

Creates a normal random variate generator with mean `mu` and standard deviation `sigma`, using stream `s`.

```
public NormalBoxMullerGen (RandomStream s)
```

Creates a standard normal random variate generator with mean `0` and standard deviation `1`, using stream `s`.

```
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* [30]. Since the method generates two variates at a time, the second variate is returned upon the next call to `nextDouble`.

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

public class NormalPolarGen extends NormalGen
```

Constructors

`public NormalPolarGen (RandomStream s, double mu, double sigma)`

Creates a normal random variate generator with mean `mu` and standard deviation `sigma`, using stream `s`.

`public NormalPolarGen (RandomStream s)`

Creates a standard normal random variate generator with $\mu = 0$ and $\sigma = 1$, using stream `s`.

`public NormalPolarGen (RandomStream s, NormalDist dist)`

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

NormalInverseFromDensityGen

This class implements *normal* random variate generators using numerical inversion of the *normal* density as described in [12]. It makes use of the class `InverseDistFromDensity`. A set of tables are precomputed to speed up the generation of normal random variables by numerical inversion. This will be useful if one wants to generate a large number of random variables.

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

public class NormalInverseFromDensityGen extends NormalGen
```

Constructors

```
public NormalInverseFromDensityGen (RandomStream stream, double mu,
                                    double sigma, double ueps, int order)
```

Creates a normal random variate generator with parameters $\mu =$ `mu` and $\sigma =$ `sigma`, using stream `stream`. It uses numerical inversion with precomputed tables. The u-resolution `ueps` is the desired absolute error in the `cdf`, and `order` is the degree of the Newton interpolating polynomial over each interval.

```
public NormalInverseFromDensityGen (RandomStream stream, NormalDist dist,
                                    double ueps, int order)
```

Similar to the first constructor, with the normal distribution `dist`.

```
public NormalInverseFromDensityGen (RandomStream stream,
                                    InverseDistFromDensity dist)
```

Creates a new normal generator using the *normal* distribution `dist` and stream `stream`. `dist` may be obtained by calling method `getDistribution`, after using one of the other constructors to create the precomputed tables. This is useful when one needs many generators using the same normal distribution. Precomputing tables for numerical inversion is costly; thus using only one set of tables for many generators is more efficient. The first `NormalInverseFromDensityGen` generator using the other constructors creates the precomputed tables. Then all other streams use this constructor with the same set of tables.

Methods

```
public double getUepsilon()
```

Returns the u-resolution `ueps`.

```
public int getOrder()
```

Returns the order of the interpolating polynomial.

NormalKindermannRamageGen

This class implements *normal* random variate generators using the *Kindermann-Ramage* method [26]. The code was taken from UNURAN [29]. It includes the correction of the error in the original *Kindermann-Ramage* method found by the authors in [35].

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

public class NormalKindermannRamageGen extends NormalGen
```

Constructors

```
public NormalKindermannRamageGen (RandomStream s,
                                  double mu, double sigma)
```

Creates a normal random variate generator with mean `mu` and standard deviation `sigma`, using stream `s`.

```
public NormalKindermannRamageGen (RandomStream s)
```

Creates a standard normal random variate generator with mean `0` and standard deviation `1`, using stream `s`.

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

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

NormalInverseGaussianGen

This class implements random variate generators for the *normal inverse gaussian* ($\mathcal{NIG}$) distribution. See the definition of `NormalInverseGaussianDist` in package `probdist`.

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

public class NormalInverseGaussianGen extends RandomVariateGen
```

Constructors

```
public NormalInverseGaussianGen (RandomStream s, double alpha,
                                 double beta, double mu, double delta)
```

Creates an *normal inverse gaussian* random variate generator with parameters $\alpha =$ `alpha`, $\beta =$ `beta`, $\mu =$ `mu` and $\delta =$ `delta`, using stream `s`.

```
public NormalInverseGaussianGen (RandomStream s,
                                 NormalInverseGaussianDist dist)
```

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

Methods

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

NOT IMPLEMENTED. Use the daughter classes.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getBeta()
```

Returns the parameter β of this object.

```
public double getMu()
```

Returns the parameter μ of this object.

```
public double getDelta()
```

Returns the parameter δ of this object.

```
public void setParams (double alpha, double beta, double mu,
                       double delta)
```

Sets the parameters α, β, μ and δ of this object.

NormalInverseGaussianIGGen

This class implements a *normal inverse gaussian* (NIG) random variate generator by using a normal generator (N) and an inverse gaussian generator (IG), as described in the following [37, 24]

$$\begin{aligned} Y &\sim IG(\delta/\gamma, \delta^2) \\ X \mid (Y = y) &\sim N(\mu + \beta y, y). \end{aligned} \tag{34}$$

The normal $N(\mu, \sigma^2)$ has mean μ and variance σ^2, while the inverse gaussian has the parametrization described in equation (25) on page 62. If $\gamma = \sqrt{\alpha^2 - \beta^2}$ with $0 \le |\beta| < \alpha$ and $\delta > 0$, then $X \sim NIG(\alpha, \beta, \mu, \delta)$.

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

public class NormalInverseGaussianIGGen extends NormalInverseGaussianGen
```

Constructors

```
public NormalInverseGaussianIGGen (InverseGaussianGen ig, NormalGen ng,
                                   double beta, double mu)
```

Creates a *normal inverse gaussian* random variate generator with parameters α, $\beta =$ `beta`, $\mu =$ `mu` and δ, using generators `ig` and `ng`, as described in eq. (34). The parameters α and δ are included in generator `ig`.

Methods

```
public static double nextDouble (InverseGaussianGen ig, NormalGen ng,
                                 double beta, double mu)
```

Generates a new variate from the *normal inverse gaussian* distribution with parameters α, β $=$ `beta`, $\mu =$ `mu` and δ, using generators `ig` and `ng`, as described in eq. (34). The parameters α and δ are included in generator `ig`.

ParetoGen

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

$$f(x) = \begin{cases} \dfrac{\alpha\beta^\alpha}{x^{\alpha+1}} & \text{for } x > \beta \\ 0 & \text{for } x \le \beta \end{cases} \tag{35}$$

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

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

public class ParetoGen extends RandomVariateGen
```

Constructors

```
public ParetoGen (RandomStream s, double alpha, double beta)
```

Creates a Pareto random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, using stream `s`.

```
public ParetoGen (RandomStream s, double alpha)
```

Creates a Pareto random variate generator with parameters $\alpha =$ `alpha` and $\beta = 1$, using stream `s`.

```
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 $\alpha =$ `alpha` and $\beta =$ `beta`, using stream `s`.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getBeta()
```

Returns the parameter β of this object.

Pearson5Gen

THIS CLASS HAS BEEN RENAMED `InverseGammaGen`.

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

$$f(x) = \begin{cases} \dfrac{x^{-(\alpha+1)}e^{-\beta/x}}{\beta^{-\alpha}\Gamma(\alpha)} & \text{for } x > 0 \\ 0 & \text{otherwise,} \end{cases} \tag{36}$$

where Γ is the gamma function.

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


@Deprecated
public class Pearson5Gen extends RandomVariateGen
```

Constructors

```
public Pearson5Gen (RandomStream s, double alpha, double beta)
```

THIS CLASS HAS BEEN RENAMED `InverseGammaGen`. Creates a Pearson5 random variate generator with parameters $\alpha =$ `alpha` and $\beta =$ `beta`, using stream `s`.

```
public Pearson5Gen (RandomStream s, double alpha)
```

Creates a Pearson5 random variate generator with parameters $\alpha =$ `alpha` and $\beta = 1$, using stream `s`.

```
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 $\beta > 0$.

```
public double getAlpha()
```

Returns the parameter α of this object.

```
public double getBeta()
```

Returns the parameter β of this object.

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

$$f(x) = \begin{cases} \dfrac{(x/\beta)^{\alpha_1 - 1}}{\beta \mathcal{B}(\alpha_1, \alpha_2)(1 + x/\beta)^{\alpha_1 + \alpha_2}} & \text{for } x > 0, \\ 0 & \text{otherwise,} \end{cases} \tag{37}$$

where $\mathcal{B}$ is the beta function.

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

```
public class Pearson6Gen extends RandomVariateGen
```

Constructors

```
public Pearson6Gen (RandomStream s, double alpha1, double alpha2,
                                    double beta)
```

Creates a Pearson6 random variate generator with parameters α_1 = `alpha1`, α_2 = `alpha2` and β = `beta`, using stream `s`.

```
public Pearson6Gen (RandomStream s, double alpha1, double alpha2)
```

Creates a Pearson6 random variate generator with parameters α_1 = `alpha1`, α_2 = `alpha2` and $\beta = 1$, using stream `s`.

```
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 $\beta > 0$.

```
public double getAlpha1()
```

Returns the α_1 parameter of this object.

```
public double getAlpha2()
```

Returns the α_2 parameter of this object.

```
public double getBeta()
```

Returns the β parameter of this object.

PowerGen

This class implements random variate generators for the *power* distribution with shape parameter $c > 0$, over the interval $[a, b]$. Its density is

$$f(x) = \frac{c(x-a)^{c-1}}{(b-a)^c}, \tag{38}$$

for $a \le x \le b$, and 0 elsewhere.

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

public class PowerGen extends RandomVariateGen
```

Constructors

```
public PowerGen (RandomStream s, double a, double b, double c)
```

Creates a Power random variate generator with parameters $a =$ `a`, $b =$ `b` and $c =$ `c`, using stream `s`.

```
public PowerGen (RandomStream s, double c)
```

Creates a Power random variate generator with parameters $a = 0$, $b = 1$ and $c =$ `c`, using stream `s`.

```
public PowerGen (RandomStream s, PowerDist dist)
```

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

Methods

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

Uses inversion to generate a new variate from the power distribution with parameters $a =$ `a`, $b =$ `b`, and $c =$ `c`, using stream `s`.

```
public double getA()
```

Returns the parameter a.

```
public double getB()
```

Returns the parameter b.

```
public double getC()
```

Returns the parameter c.

```
public void setParams (double a, double b, double c)
```

Sets the parameters a, b and c for this object.

RayleighGen

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

$$f(x) = \begin{cases} \frac{(x-a)}{\beta^2}\, e^{-(x-a)^2/2\beta^2} & \text{for } x \ge a \\ 0 & \text{for } x < a, \end{cases}$$

where $\beta > 0$.

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

public class RayleighGen extends RandomVariateGen
```

Constructors

`public RayleighGen (RandomStream s, double a, double beta)`

Creates a Rayleigh random variate generator with parameters $a =$ `a` and $\beta =$ `beta`, using stream `s`.

`public RayleighGen (RandomStream s, double beta)`

Creates a Rayleigh random variate generator with parameters $a = 0$ and $\beta =$ `beta`, using stream `s`.

`public RayleighGen (RandomStream s, RayleighDist dist)`

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

Methods

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

Uses inversion to generate a new variate from the Rayleigh distribution with parameters $a =$ `a` and $\beta =$ `beta`, using stream `s`.

`public double getA()`

Returns the parameter a.

`public double getSigma()`

Returns the parameter β.

`public void setParams (double a, double beta)`

Sets the parameters $a =$ `a` and $\beta =$ `beta` for this object.

StudentGen

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

$$f(x) = \frac{\Gamma\left((n+1)/2\right)}{\Gamma(n/2)\sqrt{\pi n}} \left[1 + \frac{x^2}{n}\right]^{-(n+1)/2} \qquad \text{for } -\infty < x < \infty, \tag{39}$$

where $\Gamma(x)$ is the gamma function defined in (21).

The `nextDouble` method simply calls `inverseF` on the distribution.

The following table gives the CPU time needed to generate 10^7 Student random variates using the different implementations available in SSJ. The second test (Q) was made with the inverse in `StudentDistQuick`, while the first test was made with the inverse in `StudentDist`. These tests were made on a machine with processor AMD Athlon 4000, running Red Hat Linux, with clock speed at 2400 MHz.

Generator	time in seconds
`StudentGen`	22.4
`StudentGen(Q)`	6.5
`StudentPolarGen`	1.4

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

public class StudentGen extends RandomVariateGen
```

Constructors

`public StudentGen (RandomStream s, int n)`

Creates a Student random variate generator with n degrees of freedom, using stream `s`.

`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`.

`public int getN()`

Returns the value of n for this object.

StudentPolarGen

This class implements *Student* random variate generators using the *polar* method of [4]. The code is adapted from UNURAN (see [29]).

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.

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

public class StudentPolarGen extends StudentGen
```

Constructors

`public StudentPolarGen (RandomStream s, int n)`

Creates a Student random variate generator with n degrees of freedom, using stream `s`.

`public StudentPolarGen (RandomStream s, StudentDist dist)`

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

StudentNoncentralGen

This class implements random variate generators for the *noncentral Student-t* distribution with $n > 0$ degrees of freedom and noncentrality parameter δ. If X is distributed according to a normal distribution with mean δ and variance 1, and Y (statistically independent of X) is distributed according to a chi-square distribution with n degrees of freedom, then

$$T' = \frac{X}{\sqrt{Y/n}}$$

has a noncentral t-distribution with n degrees of freedom and noncentrality parameter δ.

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

```
public class StudentNoncentralGen extends RandomVariateGen
```

Constructors

```
public StudentNoncentralGen (NormalGen ngen, ChiSquareGen cgen)
```

Creates a *noncentral-t* random variate generator using normal generator `ngen` and chi-square generator `cgen`.

Methods

```
public void setNormalGen (NormalGen ngen)
```

Sets the normal generator to `ngen`.

```
public void setChiSquareGen (ChiSquareGen cgen)
```

Sets the chi-square generator to `cgen`.

TriangularGen

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

$$f(x) = \begin{cases} \frac{2(x-a)}{(b-a)(m-a)} & \text{for } a \le x \le m, \\ \frac{2(b-x)}{(b-a)(b-m)} & \text{for } m \le x \le b, \\ 0 & \text{elsewhere,} \end{cases} \tag{40}$$

where $a \le m \le b$ (see, e.g., [28]).

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, double a, double b, double m)
```

Creates a triangular random variate generator over the interval (`a`, `b`), with parameter `m`, using stream `s`.

```
public TriangularGen (RandomStream s, double m)
```

Creates a triangular random variate generator over the interval $(0, 1)$, with parameter `m`, using stream `s`.

```
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 $a =$ `a`, $b =$ `b` and $m =$ `m` and stream `s`, using inversion.

```
public double getA()
```

Returns the value of a for this object.

```
public double getB()
```

Returns the value of b for this object.

```
public double getM()
```

Returns the value of m for this object.

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

$$f(x) = 1/(b - a) \qquad \text{for } a \le x \le b. \tag{41}$$

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, double a, double b)`

Creates a uniform random variate generator over the interval (`a`, `b`), using stream `s`.

`public UniformGen (RandomStream s)`

Creates a uniform random variate generator over the interval $(0, 1)$, using stream `s`.

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

`public double getA()`

Returns the value of a for this object.

`public double getB()`

Returns the value of b for this object.

WeibullGen

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

$$f(x) = \alpha\lambda^{\alpha}(x-\delta)^{\alpha-1}\exp[-(\lambda(x-\delta))^{\alpha}] \qquad \text{for } x > \delta, \tag{42}$$

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

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, double alpha, double lambda,
                                   double delta)
```

Creates a Weibull random variate generator with parameters $\alpha =$ `alpha`, $\lambda =$ `lambda` and $\delta =$ `delta`, using stream `s`.

```
public WeibullGen (RandomStream s, double alpha)
```

Creates a Weibull random variate generator with parameters $\alpha =$ `alpha`, $\lambda = 1$ and $\delta = 0$, using stream `s`.

```
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 $\alpha =$ `alpha`, $\lambda =$ `lambda`, and $\delta =$ `delta`, using stream `s`.

```
public double getAlpha()
```

Returns the parameter α.

```
public double getLambda()
```

Returns the parameter λ.

```
public double getDelta()
```

Returns the parameter δ.

UnuranContinuous

This class permits one to create continuous univariate generators 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 generator 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 bionomial 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] Z. W. Birnbaum and S. C. Saunders. A new family of life distributions. *Journal of Applied Probability*, 6:319–327, 1969.

[8] 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.

[9] 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.

[10] R. C. H. Cheng. The generation of gamma variables with non-integral shape parameter. *Applied Statistics*, 26:71–75, 1977.

[11] R. C. H. Cheng. Generating beta variates with nonintegral shape parameters. *Communications of the ACM*, 21:317–322, 1978.

[12] G. Derflinger, W. Hörmann, and J. Leydold. Random variate generation by numerical inversion when only the density is known. *ACM Transactions on Modeling and Computer Simulation*, 20(4):Article 18, 2010.

[13] L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY, 1986.

[14] L. Devroye. Random variate generation in one line of code. In *Proceedings of the 1996 Winter Simulation Conference*, pages 265–271. IEEE Press, 1996.

[15] L. Devroye. Nonuniform random variate generation. In S. G. Henderson and B. L. Nelson, editors, *Simulation*, Handbooks in Operations Research and Management Science, pages 83–121. Elsevier, Amsterdam, The Netherlands, 2006. Chapter 4.

[16] L. Devroye and L. Györfi. *Nonparametric Density Estimation: The L_1 View*. John Wiley, New York, NY, 1985.

[17] J. E. Gentle. *Random Number Generation and Monte Carlo Methods*. Springer, New York, NY, 1998.

[18] W. Höermann and G. Derflinger. The ACR method for generating normal random variables. *OR Spektrum*, 12:181–185, 1990.

[19] 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, Pistacaway, NJ, Dec 2000. IEEE Press.

[20] W. Hörmann, J. Leydold, and G. Derflinger. *Automatic Nonuniform Random Variate Generation*. Springer-Verlag, Berlin, 2004.

[21] N. L. Johnson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, volume 1. Wiley, 2nd edition, 1994.

[22] N. L. Johnson, S. Kotz, and N. Balakrishnan. *Continuous Univariate Distributions*, volume 2. Wiley, 2nd edition, 1995.

[23] V. Kachitvichyanukul and B. Schmeiser. Computer generation of hypergeometric random variates. *J. Statist. Comput. Simul.*, 22:127–145, 1985.

[24] A. Kalemanova, B. Schmid, and R. Werner. The normal inverse gaussian distribution for synthetic CDO pricing. *Journal of derivatives*, 14(3):80–93, 2007.

[25] A. W. Kemp. Efficient generation of logarithmically distributed pseudo-random variables. *Applied Statistics*, 30:249–253, 1981.

[26] A. J. Kinderman and J. G. Ramage. Computer generation of normal random variables. *Journal of the American Statistical Association*, 71:893–898, 1976.

[27] K. Krishnamoorthy. *Handbook of statistical distributions with applications*. Chapman & Hall/CRC Press, Boca Raton, FL, 2006.

[28] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, NY, third edition, 2000.

[29] J. Leydold and W. Hörmann. *UNU.RAN—A Library for Universal Non-Uniform Random Number Generators*, 2002. Available at `http://statistik.wu-wien.ac.at/unuran`.

[30] G. Marsaglia. Improving the polar method for generating a pair of random variables. Technical report, Boeing Scientific Research Laboratory, Seattle, Washington, 1962.

[31] J. R. Michael, W. R. Schuchany, and R. W. Haas. Generating random variates using transformations with multiple roots. *The American Statistician*, 30:88–90, 1976.

[32] H. Sakasegawa. Stratified rejection and squeeze method for generating beta random numbers. *Annals of the Institute of Mathematical Statistics*, 35B:291–302, 1983.

[33] B. Silverman. *Density Estimation for Statistics and Data Analysis.* Chapman and Hall, London, 1986.

[34] E. Stadlober and H. Zechner. Generating beta variates via patchwork rejection. *Computing*, 50:1–18, 1993.

[35] G. Tirler, P. Dalgaard, W. Hörmann, and J. Leydold. An error in the Kinderman-Ramage method and how to fix it. *Computational Statistics and Data Analysis*, 47(3):433–440, 2004.

[36] G. Ulrich. Computer generation of distributions on the m-sphere. *Applied Statistics*, 33:158–163, 1984.

[37] N. Webber and C. Ribeiro. A Monte Carlo method for the normal inverse gaussian option valuation model using an inverse gaussian bridge. Technical Report 5, Society for Computational Economics, 2003.