SSJ User's Guide

Package `rng`

Random Number Generators

Version: December 17, 2014

Contents

Overview

This package offers the basic facilities for generating uniform random numbers. It provides an interface called `RandomStream` and some implementations of that interface. The interface specifies that each stream of random numbers is partitioned into multiple substreams and that methods are available to jump between the substreams, as discussed in [9, 8, 11]. For an example of how to use these streams properly, see `InventoryCRN` in the set of example programs.

Each implementation uses a specific backbone uniform random number generator (RNG), whose period length is typically partitioned into very long non-overlapping segments to provide the streams and substreams. A stream can generate uniform variates (real numbers) over the interval (0,1), uniform integers over a given range of values $\{i, \ldots, j\}$, and arrays of these.

The generators provided here have various speeds and period lengths. `MRG32k3a` is the one that has been most extensively tested, but it is not among the fastest. The `LFSR113`, `GenF2w32`, `MT19937`, and the `WELL` generators produce sequences of bits that obey a linear recurrence, so they eventually fail statistical tests that measure the linear complexity of these bits sequences. But this can affect only very special types of applications.

For each generator, the following tables give the approximate period length (period), the CPU time (in seconds) to generate 10^9 $U(0,1)$ random numbers (gen. time), and the CPU time to jump ahead 10^6 times to the next substream (jump time). The following timings are on a 2100 MHz 32-bit AMD Athlon XP 2800+ computer running Linux, with the JDK 1.4.2.

RNG	period	gen. time	jump time
LFSR113	2^{113}	51	0.08
WELL512	2^{512}	55	372
WELL1024	2^{1024}	55	1450
MT19937	2^{19937}	56	60
WELL607	2^{607}	61	523
GenF2w32	2^{800}	62	937
MRG31k3p	2^{185}	66	1.8
MRG32k3a	2^{191}	109	2.3
F2NL607	2^{637}	125	523
RandRijndael	2^{130}	260	0.9

The following timings are on a 2400 MHz 64-bit AMD Athlon 64 Processor 4000+ computer running Linux, with the JDK 1.5.0.

RNG	period	gen. time	jump time
LFSR113	2^{113}	31	0.08
WELL607	2^{607}	33	329
WELL512	2^{512}	33	234
WELL1024	2^{1024}	34	917
LFSR258	2^{258}	35	0.18
MT19937	2^{19937}	36	46
GenF2w32	2^{800}	43	556
MRG31k3p	2^{185}	51	0.89
F2NL607	2^{637}	65	329
MRG32k3a	2^{191}	70	1.1
RandRijndael	2^{130}	127	0.6

Other tools included in this package permit one to manage and synchronize several streams simultaneously (`RandomStreamManager`), to create random stream factories for a given type of stream (`BasicRandomStreamFactory`), and to apply automatic transformations to the output of a given stream (`AntitheticStream` and `BakerTransformedStream`).

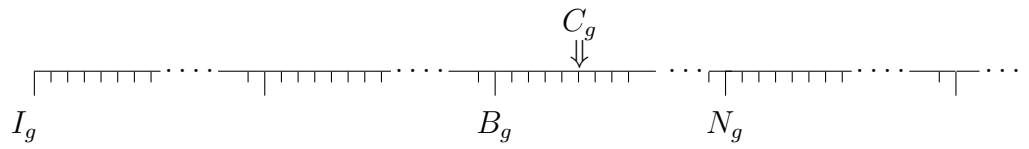
For further details about uniform RNGs, we refer the reader to [1, 6, 7].

RandomStream

This interface defines the basic structures to handle multiple streams of uniform (pseudo)-random numbers and convenient tools to move around within and across these streams. The actual random number generators (RNGs) are provided in classes that implement this `RandomStream` interface. Each stream of random numbers is an object of the class that implements this interface, and can be viewed as a virtual random number generator.

For each type of base RNG (i.e., each implementation of the `RandomStream` interface), the full period of the generator is cut into adjacent *streams* (or segments) of length Z, and each of these streams is partitioned into V *substreams* of length W, where $Z = VW$. The values of V and W depend on the specific RNG, but are usually larger than 2^{50}. Thus, the distance Z between the starting points of two successive streams provided by an RNG usually exceeds 2^{100}. The initial seed of the RNG is the starting point of the first stream. It has a default value for each type of RNG, but this initial value can be changed by calling `setPackageSeed` for the corresponding class. Each time a new `RandomStream` is created, its starting point (initial seed) is computed automatically, Z steps ahead of the starting point of the previously created stream of the same type, and its current state is set equal to this starting point.

For each stream, one can advance by one step and generate one value, or go ahead to the beginning of the next substream within this stream, or go back to the beginning of the current substream, or to the beginning of the stream, or jump ahead or back by an arbitrary number of steps. Denote by C_g the current state of a stream g, I_g its initial state, B_g the state at the beginning of the current substream, and N_g the state at the beginning of the next substream. The following diagram shows an example of a stream whose state is at the 6th value of the third substream, i.e., $2W + 5$ steps ahead of its initial state I_g and 5 steps ahead of its state B_g. The form of the state of a stream depends on its type. For example, the state of a stream of class `MRG32k3a` is a vector of six 32-bit integers represented internally as floating-point numbers (in `double`).



The methods for manipulating the streams and generating random numbers are implemented differently for each type of RNG. The methods whose formal parameter types do not depend on the RNG type are specified in the interface `RandomStream`. The others (e.g., for setting the seeds) are given only in the classes that implement the specific RNG types.

See [2, 9, 11] for examples of situations where the multiple streams offered here are useful.

Methods for generating random variates from non-uniform distributions are provided in the `randvar` package.

```
package umontreal.iro.lecuyer.rng;

public interface RandomStream
```

Methods

`public void resetStartStream();`

Reinitializes the stream to its initial state I_g: C_g and B_g are set to I_g.

`public void resetStartSubstream();`

Reinitializes the stream to the beginning of its current substream: C_g is set to B_g.

`public void resetNextSubstream();`

Reinitializes the stream to the beginning of its next substream: N_g is computed, and C_g and B_g are set to N_g.

`public String toString();`

Returns a string containing the current state of this stream.

`public double nextDouble();`

Returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, using this stream, after advancing its state by one step. The generators programmed in SSJ never return the values 0 or 1.

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

Generates `n` (pseudo)random numbers from the uniform distribution and stores them into the array `u` starting at index `start`.

`public int nextInt (int i, int j);`

Returns a (pseudo)random number from the discrete uniform distribution over the integers $\{i, i+1, \ldots, j\}$, using this stream. (Calls `nextDouble` once.)

`public void nextArrayOfInt (int i, int j, int[] u, int start, int n);`

Generates `n` (pseudo)random numbers from the discrete uniform distribution over the integers $\{i, i+1, \ldots, j\}$, using this stream and stores the result in the array `u` starting at index `start`. (Calls `nextInt` `n` times.)

CloneableRandomStream

`CloneableRandomStream` extends `RandomStream` and `Cloneable`. All classes that implements this interface are able to produce cloned objects.

The cloned object is entirely independent of the older odject. Moreover the cloned object has all the same properties as the older one. All his seeds are duplicated, and therefore both generators will produce the same random number sequence.

```
package umontreal.iro.lecuyer.rng;

public interface CloneableRandomStream extends RandomStream, Cloneable
```

Methods

```
public CloneableRandomStream clone();
```

Clones the current object and returns its copy.

RandomStreamBase

This class provides a convenient foundation on which RNGs can be built. It implements all the methods which do not depend directly on the generator itself, but only on its output, which is to be defined by implementing the `abstract` method `nextValue`. In the present class, all methods returning random numbers directly or indirectly (`nextDouble`, `nextArrayOfDouble`, `nextInt` and `nextArrayOfInt`) call `nextValue`. Thus, to define a subclass that implements a RNG, it suffices to implement `nextValue`, in addition to the `reset...` and `toString` methods. Of course, the other methods may also be overridden for improved efficiency.

If the `nextValue` already generates numbers with a precision of 53-bits or higher, then `nextDouble` can be overridden to improve the performance. The mechanism for increasing the precision assumes that `nextValue` returns at least 29 bits of precision, in which case the higher precision numbers will have roughly 52 bits of precision. This mechanism was designed primarily for RNGs that return numbers with around 30 to 32 bits of precision.

`RandomStreamBase` and its subclasses are implementing the `Serializable` interface. Each class has a serial number wich represent the class version. For instance `70510` means that the last change was the `10th May 2007`.

```
package umontreal.iro.lecuyer.rng;

public abstract class RandomStreamBase implements CloneableRandomStream,
                                                   Serializable

   public abstract void resetStartStream();
   public abstract void resetStartSubstream();
   public abstract void resetNextSubstream();
   public abstract String toString();

   public void increasedPrecision (boolean incp)
```

After calling this method with `incp = true`, each call to the RNG (direct or indirect) for this stream will return a uniform random number with more bits of precision than what is returned by `nextValue`, and will advance the state of the stream by 2 steps instead of 1 (i.e., `nextValue` will be called twice for each random number).

More precisely, if `s` is a stream of a subclass of `RandomStreamBase`, when the precision has been increased, the instruction "`u = s.nextDouble()`", is equivalent to "`u = (s.nextValue() + s.nextValue()*fact) % 1.0`" where the constant `fact` is equal to 2^{-24}. This also applies when calling `nextDouble` indirectly (e.g., via `nextInt`, etc.). By default, or if this method is called again with `incp = false`, each call to `nextDouble` for this stream advances the state by 1 step and returns the same number as `nextValue`.

```
   protected abstract double nextValue();
```

This method should return the next random number (between 0 and 1) from the current stream. If the stream is set to the high precision mode (`increasedPrecision(true)` was called), then each call to `nextDouble` will call `nextValue` twice, otherwise it will call it only once.

`public double nextDouble()`

Returns a uniform random number between 0 and 1 from the stream. Its behavior depends on the last call to `increasedPrecision`. The generators programmed in SSJ never return the values 0 or 1.

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

Calls `nextDouble` `n` times to fill the array `u`.

`public int nextInt (int i, int j)`

Calls `nextDouble` once to create one integer between `i` and `j`. This method always uses the highest order bits of the random number. It should be overridden if a faster implementation exists for the specific generator.

`public void nextArrayOfInt (int i, int j, int[] u, int start, int n)`

Calls `nextInt` `n` times to fill the array `u`. This method should be overridden if a faster implementation exists for the specific generator.

```
@Deprecated
public String formatState()
```

Use the `toString` method.

`public RandomStreamBase clone()`

Clones the current generator and return its copy.

RandomPermutation

Provides methods to randomly shuffle arrays or lists using a random stream.

```
package umontreal.iro.lecuyer.rng;

public class RandomPermutation
```

`public static void init (byte[] array, int n)`

Initializes `array` with the first n positive integers in natural order as `array`$[i-1] = i$, for $i = 1, ..., n$. The size of `array` must be at least n.

`public static void init (short[] array, int n)`

Similar to `init(byte[], int)`.

`public static void init (int[] array, int n)`

Similar to `init(byte[], int)`.

`public static void init (long[] array, int n)`

Similar to `init(byte[], int)`.

`public static void init (float[] array, int n)`

Similar to `init(byte[], int)`.

`public static void init (double[] array, int n)`

Similar to `init(byte[], int)`.

`public static void shuffle (List<?> list, RandomStream stream)`

Same as `java.util.Collections.shuffle(List<?>, Random)`, but uses a `RandomStream` instead of `java.util.Random`.

`public static void shuffle (Object[] array, RandomStream stream)`

Randomly permutes `array` using `stream`. This method permutes the whole array.

`public static void shuffle (byte[] array, RandomStream stream)`

Randomly permutes `array` using `stream`. This method permutes the whole array.

`public static void shuffle (short[] array, RandomStream stream)`

Similar to `shuffle(byte[], RandomStream)`.

`public static void shuffle (int[] array, RandomStream stream)`

Similar to `shuffle(byte[], RandomStream)`.

`public static void shuffle (long[] array, RandomStream stream)`

Similar to `shuffle(byte[], RandomStream)`.

`public static void shuffle (char[] array, RandomStream stream)`

Similar to `shuffle(byte[], RandomStream)`.

`public static void shuffle (boolean[] array, RandomStream stream)`

Similar to `shuffle(byte[], RandomStream)`.

`public static void shuffle (float[] array, RandomStream stream)`

Similar to `shuffle(byte[], RandomStream)`.

`public static void shuffle (double[] array, RandomStream stream)`

Similar to `shuffle(byte[], RandomStream)`.

`public static void shuffle (List<?> list, int k, RandomStream stream)`

Partially permutes `list` as follows using `stream`: draws the first k new elements of `list` randomly among the n old elements of `list`, assuming that $k \leq n =$ `list.size()`. In other words, k elements are selected at random without replacement from the n `list` entries and are placed in the first k positions, in random order.

```
public static void shuffle (Object[] array, int n, int k,
                            RandomStream stream)
```

Partially permutes `array` as follows using `stream`: draws the new k elements, `array[0]` to `array[k-1]`, randomly among the old n elements, `array[0]` to `array[n-1]`, assuming that $k \leq n \leq$ `array.length`. In other words, k elements are selected at random without replacement from the first n array elements and are placed in the first k positions, in random order.

```
public static void shuffle (byte[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

```
public static void shuffle (short[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

```
public static void shuffle (int[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

```
public static void shuffle (long[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

```
public static void shuffle (char[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

```
public static void shuffle (boolean[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

```
public static void shuffle (float[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

```
public static void shuffle (double[] array, int n, int k,
                            RandomStream stream)
```

Similar to `shuffle(Object[], n, k, RandomStream)`.

RandomStreamManager

Manages a list of random streams for more convenient synchronization. All streams in the list can be reset simultaneously by a single call to the appropriate method of this stream manager, instead of calling explicitly the reset method for each individual stream.

After a random stream manager is constructed, any existing `RandomStream` object can be registered to this stream manager (i.e., added to the list) and eventually unregistered (removed from the list).

```
package umontreal.iro.lecuyer.rng;

public class RandomStreamManager
```

`public RandomStream add (RandomStream stream)`

Adds the given `stream` to the internal list of this random stream manager and returns the added stream.

`public boolean remove (RandomStream stream)`

Removes the given stream from the internal list of this random stream manager. Returns `true` if the stream was properly removed, `false` otherwise.

`public void clear()`

Removes all the streams from the internal list of this random stream manager.

`public List getStreams()`

Returns an unmodifiable list containing all the random streams in this random stream manager. The returned list, constructed by `unmodifiableList`, can be assumed to contain non-`null` `RandomStream` instances.

`public void resetStartStream()`

Forwards to the `resetStartStream` methods of all streams in the list.

`public void resetStartSubstream()`

Forwards to the `resetStartSubstream` methods of all streams in the list.

`public void resetNextSubstream()`

Forwards to the `resetNextSubstream` methods of all streams in the list.

RandomStreamFactory

Represents a random stream factory capable of constructing instances of a given type of random stream by invoking the `newInstance` method each time a new random stream is needed, instead of invoking directly the specific constructor of the desired type. Hence, if several random streams of a given type (class) must be constructed at different places in a large simulation program, and if we decide to change the type of stream in the future, there is no need to change the code at those different places. With the random stream factory, the class-specific code for constructing these streams appears at a single place, where the factory is constructed.

The class `BasicRandomStreamFactory` provides an implementation of this interface.

```
package umontreal.iro.lecuyer.rng;

public interface RandomStreamFactory

   public RandomStream newInstance();
```

Constructs and returns a new random stream. If the instantiation of the random stream fails, this method throws a `RandomStreamInstantiationException`.

BasicRandomStreamFactory

Represents a basic random stream factory that can constructs new instances of a given `RandomStream` implementation via the `newInstance` method. The class name of the implementation to be used must be passed to the constructor as a `String`, which must be the name of a nullary constructor of a `RandomStream` object (i.e., a constructor that has no parameters). The streams are constructed by the factory by reflection from this `String`.

```
package umontreal.iro.lecuyer.rng;

public class BasicRandomStreamFactory implements RandomStreamFactory
```

`public BasicRandomStreamFactory (Class rsClass)`

Constructs a new basic random stream factory with random stream class `rsClass`. The supplied class object must represent an implementation of `RandomStream` and must provide a nullary constructor. For example, to construct a factory producing `MRG32k3a` random streams, this constructor must be called with `MRG33k3a.class`.

`public Class getRandomStreamClass()`

Returns the random stream class associated with this object.

`public void setRandomStreamClass (Class rsClass)`

Sets the associated random stream class to `rsClass`. The supplied class object must represent an implementation of `RandomStream` and must provide a nullary constructor.

RandomStreamInstantiationException

This exception is thrown when a random stream factory cannot instantiate a stream on a call to its `newInstance` method.

```
package umontreal.iro.lecuyer.rng;

public class RandomStreamInstantiationException extends RuntimeException
```

`public RandomStreamInstantiationException (RandomStreamFactory factory)`

Constructs a new random stream instantiation exception with no message, no cause, and thrown by the given `factory`.

`public RandomStreamInstantiationException (RandomStreamFactory factory, String message)`

Constructs a new random stream instantiation exception with the given `message`, no cause, and concerning `factory`.

`public RandomStreamInstantiationException (RandomStreamFactory factory, Throwable cause)`

Constructs a new random stream instantiation exception with no message, the given `cause`, and concerning `factory`.

`public RandomStreamInstantiationException (RandomStreamFactory factory, String message, Throwable cause)`

Constructs a new random stream instantiation exception with the given `message`, the supplied `cause`, and concerning `factory`.

`public RandomStreamFactory getRandomStreamFactory()`

Returns the random stream factory concerned by this exception.

`public String toString()`

Returns a short description of the exception. If `getRandomStreamFactory` returns `null`, this calls `super.toString`. Otherwise, the result is the concatenation of:

a) the name of the actual class of the exception;
b) the string `":  For random stream factory "`;
c) the result of `getRandomStreamFactory.toString()`;
d) if `getMessage` is non-`null`, `", "` followed by the result of `getMessage`.

RandomStreamWithCache

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

Such caching allows for a better performance with common random numbers, when generating uniforms is time-consuming. It can also help with restoring the simulation to a certain state without setting stream-specific seeds. However, using such caching may lead to memory problems if a large quantity of random numbers are needed.

```
package umontreal.iro.lecuyer.rng;

public class RandomStreamWithCache implements RandomStream
```

Constructors

`public RandomStreamWithCache (RandomStream stream)`

Constructs a new cached random stream with internal stream `stream`.

`public RandomStreamWithCache (RandomStream stream, int initialCapacity)`

Constructs a new cached random stream with internal stream `stream`. The `initialCapacity` parameter is used to set the initial capacity of the internal array which can grow as needed; it does not limit the total size of the cache.

Methods

`public boolean isCaching()`

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

`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 RandomStream getCachedStream()`

Returns a reference to the random stream whose values are cached.

`public void setCachedStream (RandomStream stream)`

Sets the random stream whose values are cached to `stream`. If the stream is changed, the `clearCache` method is called to clear the cache.

`public void clearCache()`

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

`public void initCache()`

Resets this random stream to recover values from the cache. Subsequent calls to `nextDouble` will return the cached uniforms until all the values are returned. When the array of cached values is exhausted, the internal random stream 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 random stream.

`public int getCacheIndex()`

Return the index of the next cached value that will be returned by the stream. 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 stream.

`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.

AntitheticStream

This container class allows the user to force any `RandomStream` to return antithetic variates. That is, `nextDouble` returns $1 - u$ instead of u and the corresponding change is made in `nextInt`. Any instance of this class behaves exactly like a `RandomStream`, except that it depends on another random number generator stream, called the *base stream*, to generate its numbers. Any call to one of the `next...` methods of this class will modify the state of the base stream.

```
package umontreal.iro.lecuyer.rng;

public class AntitheticStream implements RandomStream
```

Constructors

`public AntitheticStream (RandomStream stream)`

Constructs a new antithetic stream, using the random numbers from the base stream `stream`.

Methods

`public String toString()`

Returns a string starting with `"Antithetic of "` and finishing with the result of the call to the `toString` method of the generator.

`public double nextDouble()`

Returns `1.0 - s.nextDouble()` where `s` is the base stream.

`public int nextInt (int i, int j)`

Returns `j - i - s.nextInt(i, j)` where `s` is the base stream.

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

Calls `nextArrayOfDouble (u, start, n)` for the base stream, then replaces each `u[i]` by `1.0 - u[i]`.

`public void nextArrayOfInt (int i, int j, int[] u, int start, int n)`

Calls `nextArrayOfInt (i, j, u, start, n)` for the base stream, then replaces each `u[i]` by `j - i - u[i]`.

BakerTransformedStream

This container class permits one to apply the baker's transformation to the output of any `RandomStream`. It transforms each $u \in [0,1]$ into $2u$ if $u \leq 1/2$ and $2(1-u)$ if $u > 1/2$. The `nextDouble` method will return the result of this transformation and the other `next...` methods are affected accordingly. Any instance of this class contains a `RandomStream` called its *base stream*, used to generate its numbers and to which the transformation is applied. Any call to one of the `next...` methods of this class will modify the state of the base stream.

The baker transformation is often applied when the `RandomStream` is actually an iterator over a point set used for quasi-Monte Carlo integration (see the `hups` package).

```
package umontreal.iro.lecuyer.rng;

public class BakerTransformedStream implements RandomStream
```

Constructors

`public BakerTransformedStream (RandomStream stream)`

Constructs a new baker transformed stream, using the random numbers from the base stream `stream`.

Methods

`public String toString()`

Returns a string starting with `"Baker transformation of "` and finishing with the result of the call to the `toString` method of the generator.

`public double nextDouble()`

Returns the baker transformation of `s.nextDouble()` where `s` is the base stream.

`public int nextInt (int i, int j)`

Generates a random integer in $\{i, ..., j\}$ via `nextDouble` (in which the baker transformation is applied).

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

Calls `nextArrayOfDouble (u, start, n)` for the base stream, then applies the baker transformation.

`public void nextArrayOfInt (int i, int j, int[] u, int start, int n)`

Fills up the array by calling `nextInt (i, j)`.

TruncatedRandomStream

Represents a container random stream generating numbers in an interval (a, b) instead of in $(0, 1)$, where $0 \leq a < b \leq 1$, by using the contained stream. If `nextDouble` returns u for the contained stream, it will return $v = a + (b - a)u$, which is uniform over (a, b), for the truncated stream. The method `nextInt` returns the integer that corresponds to v (by inversion); this integer is no longer uniformly distributed in general.

```
package umontreal.iro.lecuyer.rng;

public class TruncatedRandomStream implements RandomStream
```

Constructor

```
public TruncatedRandomStream (RandomStream stream, double a, double b)
```

RandMrg

USE `MRG32k3a` *INSTEAD of this class.* This class implements the interface `RandomStream` directly, with a few additional tools. It uses the same backbone (or main) generator as `MRG32k3a`, but it is an older implementation that does not extend `RandomStreamBase`, and it is about 10% slower.

```
package umontreal.iro.lecuyer.rng;

@Deprecated
public class RandMrg implements CloneableRandomStream, Serializable
```

Constructors

`public RandMrg()`

Constructs a new stream, initializes its seed I_g, sets B_g and C_g equal to I_g, and sets its antithetic switch to `false`. The seed I_g is equal to the initial seed of the package given by `setPackageSeed` if this is the first stream created, otherwise it is Z steps ahead of that of the stream most recently created in this class.

`public RandMrg (String name)`

Constructs a new stream with an identifier `name` (can be used when printing the stream state, in error messages, etc.).

Methods

`public static void setPackageSeed (long seed[])`

Sets the initial seed for the class `RandMrg` to the six integers in the vector `seed[0..5]`. This will be the seed (initial state) of the first stream. If this method is not called, the default initial seed is $(12345, 12345, 12345, 12345, 12345, 12345)$. If it is called, the first 3 values of the seed must all be less than $m_1 = 4294967087$, and not all 0; and the last 3 values must all be less than $m_2 = 4294944443$, and not all 0.

`public void increasedPrecis (boolean incp)`

After calling this method with `incp = true`, each call to the generator (direct or indirect) for this stream will return a uniform random number with (roughly) 53 bits of resolution instead of 32 bits, and will advance the state of the stream by 2 steps instead of 1. More precisely, if `s` is a stream of the class `RandMrg`, in the non-antithetic case, the instruction "`u = s.nextDouble()`", when the resolution has been increased, is equivalent to "`u = (s.nextDouble() + s.nextDouble()*fact) % 1.0`" where the constant `fact` is equal to 2^{-24}. This also applies when calling `nextDouble` indirectly (e.g., via `nextInt`, etc.).

By default, or if this method is called again with `incp = false`, each call to `nextDouble` for this stream advances the state by 1 step and returns a number with 32 bits of resolution.

`public void advanceState (int e, int c)`

Advances the state of this stream by k values, without modifying the states of other streams (as in `setSeed`), nor the values of B_g and I_g associated with this stream. If $e > 0$, then $k = 2^e + c$; if $e < 0$, then $k = -2^{-e} + c$; and if $e = 0$, then $k = c$. Note: c is allowed to take negative values. This method should be used only in very exceptional cases; proper use of the `reset...` methods and of the stream constructor cover most reasonable situations.

`public void setSeed (long seed[])`

Sets the initial seed I_g of this stream to the vector `seed[0..5]`. This vector must satisfy the same conditions as in `setPackageSeed`. The stream is then reset to this initial seed. The states and seeds of the other streams are not modified. As a result, after calling this method, the initial seeds of the streams are no longer spaced Z values apart. For this reason, this method should be used only in very exceptional situations; proper use of `reset...` and of the stream constructor is preferable.

`public double[] getState()`

Returns the current state C_g of this stream. This is a vector of 6 integers represented in floating-point format. This method is convenient if we want to save the state for subsequent use.

`public String toStringFull()`

Returns a string containing the name of this stream and the values of all its internal variables.

`public double nextDouble()`

Returns a (pseudo)random number from the uniform distribution over the interval $(0, 1)$, using this stream, after advancing its state by one step. Normally, the returned number has 32 bits of resolution, in the sense that it is always a multiple of $1/(2^{32} - 208)$. However, if the precision has been increased by calling `increasedPrecis` for this stream, the resolution is higher and the stream state advances by two steps.

`public RandMrg clone()`

Clones the current generator and return its copy.

MRG32k3a

Extends the abstract class `RandomStreamBase` by using as a backbone (or main) generator the combined multiple recursive generator (CMRG) `MRG32k3a` proposed by L'Ecuyer [4], implemented in 64-bit floating-point arithmetic. This backbone generator has a period length of $\rho \approx 2^{191}$. The values of V, W, and Z are 2^{51}, 2^{76}, and 2^{127}, respectively. (See `RandomStream` for their definition.) The seed of the RNG, and the state of a stream at any given step, are six-dimensional vectors of 32-bit integers, stored in `double`. The default initial seed of the RNG is $(12345, 12345, 12345, 12345, 12345, 12345)$.

```
package umontreal.iro.lecuyer.rng;

public class MRG32k3a extends RandomStreamBase
```

Constructors

`public MRG32k3a()`

Constructs a new stream, initializes its seed I_g, sets B_g and C_g equal to I_g, and sets its antithetic switch to `false`. The seed I_g is equal to the initial seed of the package given by `setPackageSeed` if this is the first stream created, otherwise it is Z steps ahead of that of the stream most recently created in this class.

`public MRG32k3a (String name)`

Constructs a new stream with an identifier `name` (used when printing the stream state).

Methods

`public static void setPackageSeed (long seed[])`

Sets the initial seed for the class `MRG32k3a` to the six integers in the vector `seed[0..5]`. This will be the seed (initial state) of the first stream. If this method is not called, the default initial seed is $(12345, 12345, 12345, 12345, 12345, 12345)$. If it is called, the first 3 values of the seed must all be less than $m_1 = 4294967087$, and not all 0; and the last 3 values must all be less than $m_2 = 4294944443$, and not all 0.

`public void setSeed (long seed[])`

Sets the initial seed I_g of this stream to the vector `seed[0..5]`. This vector must satisfy the same conditions as in `setPackageSeed`. The stream is then reset to this initial seed. The states and seeds of the other streams are not modified. As a result, after calling this method, the initial seeds of the streams are no longer spaced Z values apart. For this reason, *this method should be used only in very exceptional situations* (I have never used it myself!); proper use of `reset...` and of the stream constructor is preferable.

`public long[] getState()`

Returns the current state C_g of this stream. This is a vector of 6 integers. This method is convenient if we want to save the state for subsequent use.

`public String toString()`

Returns a string containing the name and the current state C_g of this stream.

`public String toStringFull()`

Returns a string containing the name of this stream and the values of all its internal variables.

`public MRG32k3a clone()`

Clones the current generator and return its copy.

MRG32k3aL

The same generator as `MRG32k3a`, except here it is implemented with type `long` instead of `double`. (See `MRG32k3a` for more information.)

```
package umontreal.iro.lecuyer.rng;

public class MRG32k3aL extends RandomStreamBase
```

Constructors

```
public MRG32k3aL()

public MRG32k3aL (String name)
```

Methods

See the description of the same methods in class `MRG32k3a`.

```
public static void setPackageSeed (long seed[])

public void setSeed (long seed[])

public long[] getState()

public String toString()

public String toStringFull()

public MRG32k3aL clone()
```

MRG31k3p

Extends the abstract class `RandomStreamBase`, thus implementing the `RandomStream` interface indirectly. The backbone generator is the combined multiple recursive generator (CMRG) `MRG31k3p` proposed by L'Ecuyer and Touzin [12], implemented in 32-bit integer arithmetic. This RNG has a period length of $\rho \approx 2^{185}$. The values of V, W and Z are 2^{62}, 2^{72} and 2^{134} respectively. (See `RandomStream` for their definition.) The seed and the state of a stream at any given step are six-dimensional vectors of 32-bit integers. The default initial seed is $(12345, 12345, 12345, 12345, 12345, 12345)$. The method `nextValue` provides 31 bits of precision.

The difference between the RNG of class `MRG32k3a` and this one is that this one has all its coefficients of the form $a = \pm 2^q \pm 2^r$. This permits a faster implementation than for arbitrary coefficients. On a 32-bit computer, `MRG31k3p` is about twice as fast as `MRG32k3a`. On the other hand, the latter does a little better in the spectral test and has been more extensively tested.

```
package umontreal.iro.lecuyer.rng;

public class MRG31k3p extends RandomStreamBase
```

Constructors

`public MRG31k3p()`

Constructs a new stream, initialized at its beginning. Its seed is $Z = 2^{134}$ steps away from the previous seed.

`public MRG31k3p (String name)`

Constructs a new stream with the identifier `name` (used when formatting the stream state).

Methods

`public static void setPackageSeed (int seed[])`

Sets the initial seed for the class `MRG31k3p` to the six integers of the vector `seed[0..5]`. This will be the initial state (or seed) of the next created stream. By default, if this method is not called, the first stream is created with the seed $(12345, 12345, 12345, 12345, 12345, 12345)$. If it is called, the first 3 values of the seed must all be less than $m_1 = 2147483647$, and not all 0; and the last 3 values must all be less than $m_2 = 2147462579$, and not all 0.

`public void setSeed (int seed[])`

Use of this method is strongly discouraged. Initializes the stream at the beginning of a stream with the initial seed `seed[0..5]`. This vector must satisfy the same conditions as in `setPackageSeed`. This method only affects the specified stream, all the others are not modified, so the beginning of the streams are no longer spaced Z values apart. For this reason, this method should be used only in very exceptional situations; proper use of `reset...` and of the stream constructor is preferable.

`public int[] getState()`

Returns the current state C_g of this stream. This is a vector of 6 integers represented. This method is convenient if we want to save the state for subsequent use.

`public MRG31k3p clone()`

Clones the current generator and return its copy.

LFSR113

Extends `RandomStreamBase` using a composite linear feedback shift register (LFSR) (or Tausworthe) RNG as defined in [3, 17]. This generator is the `LFSR113` proposed by [5]. It has four 32-bit components combined by a bitwise xor. Its period length is $\rho \approx 2^{113}$. The values of V, W and Z are 2^{35}, 2^{55} and 2^{90} respectively (see `RandomStream` for their definition). The seed of the RNG, and the state of a stream at any given step, are four-dimensional vectors of 32-bit integers. The default initial seed of the RNG is [1] (987654321, 987654321, 987654321, 987654321). The `nextValue` method returns numbers with 32 bits of precision.

```
package umontreal.iro.lecuyer.rng;

public class LFSR113 extends RandomStreamBase
```

Constructors

`public LFSR113()`

Constructs a new stream.

`public LFSR113 (String name)`

Constructs a new stream with the identifier `name`.

Methods

`public static void setPackageSeed (int[] seed)`

Sets the initial seed for the class `LFSR113` to the four integers of the vector `seed[0..3]`. This will be the initial state of the next created stream. The default seed for the first stream is [2] (987654321, 987654321, 987654321, 987654321). The first, second, third and fourth integers of `seed` must be either negative, or greater than or equal to 2, 8, 16 and 128 respectively.

`public void setSeed (int[] seed)`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial seed `seed[0..3]`. The seed must satisfy the same conditions as in `setPackageSeed`. This method only affects the specified stream; the others are not modified, so the beginning of the streams will not be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public int[] getState()`

Returns the current state of the stream, represented as an array of four integers.

`public LFSR113 clone()`

Clones the current generator and return its copy.

---

[1] In previous versions, it was (12345, 12345, 12345, 12345).
[2] In previous versions, it was (12345, 12345, 12345, 12345).

LFSR258

Extends `RandomStreamBase` using a 64-bit composite linear feedback shift register (LFSR) (or Tausworthe) RNG as defined in [3, 17]. This generator is the `LFSR258` proposed in [5]. It has five components combined by a bitwise xor. Its period length is $\rho \approx 2^{258}$. The values of V, W and Z are 2^{100}, 2^{100} and 2^{200} respectively (see `RandomStream` for their definition). The seed of the RNG, and the state of a stream at any given step, are five-dimensional vectors of 64-bit integers. The default initial seed [3] of the RNG is (123456789123456789, 123456789123456789, 123456789123456789, 123456789123456789, 123456789123456789). The `nextValue` method returns numbers with 53 bits of precision. This generator is fast for 64-bit machines.

```
package umontreal.iro.lecuyer.rng;

public class LFSR258 extends RandomStreamBase
```

Constructors

`public LFSR258()`

Constructs a new stream.

`public LFSR258 (String name)`

Constructs a new stream with the identifier `name`.

Methods

`public static void setPackageSeed (long seed[])`

Sets the initial seed for the class `LFSR258` to the five integers of array `seed[0..4]`. This will be the initial state of the next created stream. The default seed [4] for the first stream is (123456789123456789, 123456789123456789, 123456789123456789, 123456789123456789, 123456789123456789). The first, second, third, fourth and fifth integers of `seed` must be either negative, or greater than or equal to 2, 512, 4096, 131072 and 8388608 respectively.

`public void setSeed (long seed[])`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial seed `seed[0..4]`. The seed must satisfy the same conditions as in `setPackageSeed`. This method only affects the specified stream; the others are not modified, so the beginning of the streams will not be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public long[] getState()`

Returns the current state of the stream, represented as an array of five integers.

---

[3] In previous versions, it was (1234567890, 1234567890, 1234567890, 1234567890, 1234567890).
[4] In previous versions, it was (1234567890, 1234567890, 1234567890, 1234567890, 1234567890).

`public LFSR258 clone()`

Clones the current generator and return its copy.

WELL512

This class implements the `RandomStream` interface via inheritance from `RandomStreamBase`. The backbone generator is a Well Equidistributed Long period Linear Random Number Generator (WELL), proposed by F. Panneton in [16, 14], and which has a state size of 512 bits and a period length of $\rho \approx 2^{512}$. The values of V, W and Z are 2^{150}, 2^{200} and 2^{350} respectively (see `RandomStream` for their definition). The seed of the RNG, and the state of a stream at any given step, is a 16-dimensional vector of 32-bit integers.

```
package umontreal.iro.lecuyer.rng;

public class WELL512 extends RandomStreamBase
```

Constructors

`public WELL512()`

Constructs a new stream.

`public WELL512 (String name)`

Constructs a new stream with the identifier `name` (used in the `toString` method).

Methods

`public static void setPackageSeed (int seed[])`

Sets the initial seed of the class `WELL512` to the 16 integers of the vector `seed[0..15]`. This will be the initial seed of the class of the next created stream. At least one of the integers must be non-zero.

`public void setSeed (int seed[])`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial seed `seed[0..15]`. The seed must satisfy the same conditions as in `setPackageSeed`. This method only affects the specified stream; the others are not modified. Hence after calling this method, the beginning of the streams will no longer be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public int[] getState()`

Returns the current state of the stream, represented as an array of 16 integers.

`public WELL512 clone()`

Clones the current generator and return its copy.

WELL607

This class implements the `RandomStream` interface via inheritance from `RandomStreamBase`. The backbone generator is a Well Equidistributed Long period Linear Random Number Generator (WELL), proposed by F. Panneton in [16, 14]. The implemented generator is the `WELL607`, which has a state size of 607 bits and a period length of $\rho \approx 2^{607}$. The values of V, W and Z are 2^{150}, 2^{250} and 2^{400} respectively (see `RandomStream` for their definition). The seed of the RNG, and the state of a stream at any given step, is a 19-dimensional vector of 32-bit integers. The output of `nextValue` has 32 bits of precision.

```
package umontreal.iro.lecuyer.rng;

public class WELL607 extends WELL607base
```

Constructors

`public WELL607()`

Constructs a new stream.

`public WELL607 (String name)`

Constructs a new stream with the identifier `name` (used in the `toString` method).

Methods

`public static void setPackageSeed (int seed[])`

Sets the initial seed of the class `WELL607` to the 19 integers of the vector `seed[0..18]`. This will be the initial seed of the next created stream. At least one of the integers must not be zero and if this integer is the last one, it must not be equal to `0x80000000`.

`public void setSeed (int seed[])`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial seed `seed[0..18]`. The seed must satisfy the same conditions as in `setPackageSeed`. This method only affects the specified stream; the others are not modified. Hence after calling this method, the beginning of the streams will no longer be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public int[] getState()`

Returns the current state of the stream, represented as an array of 19 integers.

`public WELL607 clone()`

Clones the current generator and return its copy.

WELL1024

Implements the `RandomStream` interface via inheritance from `RandomStreamBase`. The backbone generator is a Well Equidistributed Long period Linear Random Number Generator (WELL), proposed by F. Panneton in [16, 14], and which has a state size of 1024 bits and a period length of $\rho \approx 2^{1024}$. The values of V, W and Z are 2^{300}, 2^{400} and 2^{700} respectively (see `RandomStream` for their definition). The seed of the RNG, and the state of a stream at any given step, is a 16-dimensional vector of 32-bit integers. The output of `nextValue` has 32 bits of precision.

```
package umontreal.iro.lecuyer.rng;

public class WELL1024 extends RandomStreamBase
```

Constructors

`public WELL1024()`

Constructs a new stream.

`public WELL1024 (String name)`

Constructs a new stream with the identifier `name` (used in the `toString` method).

Methods

`public static void setPackageSeed (int seed[])`

Sets the initial seed of this class to the 32 integers of array `seed[0..31]`. This will be the initial seed of the class and of the next created stream. At least one of the integers must be non-zero.

`public void setSeed (int seed[])`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial seed `seed[0..31]`. The seed must satisfy the same conditions as in `setPackageSeed`. This method only affects the specified stream; the others are not modified. Hence after calling this method, the beginning of the streams will no longer be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public int[] getState()`

Returns the current state of the stream, represented as an array of 32 integers.

`public WELL1024 clone()`

Clones the current generator and return its copy.

GenF2w32

Implements the `RandomStream` interface via inheritance from `RandomStreamBase`. The backbone generator is a Linear Congruential Generator (LCG) in the finite field $\mathbb{F}_{2^w}$ instead of $\mathbb{F}_2$. The implemented generator is the `GenF2w2_32` proposed by Panneton [15, 14]. Its state is 25 32-bit words and it has a period length of $2^{800} - 1$. The values of V, W and Z are 2^{200}, 2^{300} and 2^{500} respectively (see `RandomStream` for their definition). The seed of the RNG, and the state of a stream at any given step, is a 25-dimensional vector of 32-bits integers. Its `nextValue` method returns numbers with 32 bits of precision.

```
package umontreal.iro.lecuyer.rng;

public class GenF2w32 extends RandomStreamBase
```

Constructors

`public GenF2w32()`

Constructs a new stream.

`public GenF2w32 (String name)`

Constructs a new stream with the identifier `name` (used in the `toString` method).

Methods

`public static void setPackageSeed (int seed[])`

Sets the initial seed of the class `GenF2w2r32` to the 25 integers of the vector `seed[0..24]`. This will be the initial seed of the class for the next created stream. At least one of the integers must be non-zero.

`public void setSeed (int seed[])`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial seed `seed[0..24]`. The seed must satisfy the same conditions as in `setPackageSeed`. This method only affects the specified stream; the others are not modified. Hence after calling this method, the beginning of the streams will no longer be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public int[] getState()`

Returns the current state of the stream, represented as an array of 25 integers.

`public GenF2w32 clone()`

Clones the current generator and return its copy.

MT19937

Implements the `RandomStream` interface via inheritance from `RandomStreamBase`. The backbone generator is the MT19937 Mersenne Twister, proposed by Matsumoto and Nishimura [13], which has a state size of 19937 bits and a period length of $\rho \approx 2^{19937}$. Each instance uses another `CloneableRandomStream` to fill its initial state. With this design, the initial states of successive streams are not spaced by an equal number of steps, and there is no guarantee that different streams do not overlap, but damaging overlap is unlikely because of the huge size of the state space. The seed of the RNG, and the state of a stream at any given step, is a 624-dimensional vector of 32-bit integers. The output of `nextValue` has 32 bits of precision.

```
package umontreal.iro.lecuyer.rng;

public class MT19937 extends RandomStreamBase
```

Constructors

`public MT19937 (CloneableRandomStream rng)`

Constructs a new stream, using `rng` to fill its initial state.

`public MT19937 (CloneableRandomStream rng, String name)`

Constructs a new stream with the identifier `name` (used in the `toString` method).

Methods

`public MT19937 clone()`

Clones the current generator and return its copy.

F2NL607

Implements the `RandomStream` interface by using as a backbone generator the combination of the WELL607 proposed in [14, 16] (and implemented in `WELL607`) with a nonlinear generator. This nonlinear generator is made up of a small number of components (say n) combined via addition modulo 1. Each component contains an array already filled with a "random" permutation of $\{0, ..., s-1\}$ where `s` is the size of the array. These numbers and the lengths of the components can be changed by the user. Each call to the generator uses the next number in each array (or the first one if we are at the end of the array). By default, there are 3 components of lengths 1019, 1021, and 1031, respectively. The non-linear generator is combined with the WELL using a bitwise XOR operation. This ensures that the new generator has at least as much equidistribution as the WELL607, as shown in [10].

The combined generator has a period length of $\rho \approx 2^{637}$. The values of V, W and Z are 2^{250}, 2^{150}, and 2^{400}, respectively (see `RandomStream` for their definition). The seed of the RNG has two part: the linear part is a 19-dimensional vector of 32-bit integers, while the nonlinear part is made up of a n-dimensional vector of indices, representing the position of the generator in each array of the nonlinear components.

```
package umontreal.iro.lecuyer.rng;

public class F2NL607
```

Constructors

`public F2NL607()`

Constructs a new stream, initializing it at its beginning. Also makes sure that the seed of the next constructed stream is Z steps away. Sets its antithetic switch to `false` and sets the stream to normal precision mode (offers 32 bits of precision).

`public F2NL607 (String name)`

Constructs a new stream with the identifier `name` (used in the `toString` method).

Methods

`public static void setPackageLinearSeed (int seed[])`

Sets the initial seed of the linear part of the class `F2NL607` to the 19 integers of the vector `seed[0..18]`. This will be the initial seed (or seed) of the next created stream. At least one of the integers must be non-zero and if this integer is the last one, it must not be equal to `0x7FFFFFFF`.

`public void setLinearSeed (int seed[])`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial linear seed `seed[0..18]`. The seed must satisfy the same conditions

as in `setPackageSeed`. The non-linear seed is not modified; thus the non-linear part of the random number generator is reset to the beginning of the old stream. This method only affects the specified stream; the others are not modified. Hence after calling this method, the beginning of the streams will no longer be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public int[] getLinearState()`

Returns the current state of the linear part of the stream, represented as an array of 19 integers.

`public static void setPackageNonLinearSeed (int seed[])`

Sets the non-linear part of the initial seed of the class `F2NL607` to the n integers of the vector `seed[0..n-1]`, where n is the number of components of the non-linear part. The default is $n = 3$. Each of the integers must be between 0 and the length of the corresponding component minus one. By default, the lengths are $(1019, 1021, 1031)$.

`public void setNonLinearSeed (int seed[])`

This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial non-linear seed `seed[0..n-1]`, where n is the number of components of the non-linear part of the generator. The linear seed is not modified so the linear part of the random number generator is reset to the beginning of the old stream. This method only affects the specified stream; the others are not modified. Hence after calling this method, the beginning of the streams will no longer be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public int[] getNonLinearState()`

Returns the current state of the non-linear part of the stream, represented as an array of n integers, where n is the number of components in the non-linear generator.

`public static int[][] getNonLinearData()`

Return the data of all the components of the non-linear part of the random number generator. This data is explained in the introduction.

`public static void setNonLinearData (int[][] data)`

Selects new data for the components of the non-linear generator. The number of arrays in `data` will decide the number of components. Each of the arrays will be assigned to one of the components. The period of the resulting non-linear generator will be equal to the lowest common multiple of the lengths of the arrays. It is thus recommended to choose only prime length for the best results.

NOTE : This method cannot be called if at least one instance of `F2NL607` has been constructed. In that case, it will throw an `IllegalStateException`.

```
public static void setScrambleData (RandomStream rand, int steps,
                                    int[] size)
```

Selects new data for the components of the non-linear generator. The number of arrays in `data` will decide the number of components. Each of the arrays will be assigned to one of

the components. The period of the resulting non-linear generator will be equal to the lowest common multiple of the lengths of the arrays. It is thus recommended to choose only prime length for the best results.

NOTE : This method cannot be called if at least one instance of `F2NL607` has been constructed. In that case, it will throw an `IllegalStateException`.

`public F2NL607 clone()`

Clones the current generator and return its copy.

RandRijndael

Implements a RNG using the Rijndael block cipher algorithm (AES) with key and block lengths of 128 bits. A block of 128 bits is encrypted by the Rijndael algorithm to generate 128 pseudo-random bits. Those bits are split into four words of 32 bits which are returned successively by the method `nextValue`. The unencrypted block is the state of the generator. It is incremented by 1 at every four calls to `nextValue`. Thus, the period is 2^{130} and jumping ahead is easy. The values of V, W and Z are 2^{40}, 2^{42} and 2^{82}, respectively (see `RandomStream` for their definition). Seeds/states must be given as 16-dimensional vectors of bytes (8-bit integers). The default initial seed is a vector filled with zeros.

The Rijndael implementation used here is that of the *Cryptix Development Team*, which can be found on the Rijndael creators' page `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/`.

```
package umontreal.iro.lecuyer.rng;

public class RandRijndael extends RandomStreamBase
```

Constructors

`public RandRijndael()`
Constructs a new stream.

`public RandRijndael (String name)`
Constructs a new stream with the identifier `name` (used in the `toString` method).

Methods

`public static void setPackageSeed (byte seed[])`
Sets the initial seed for the class `RandRijndael` to the 16 bytes of the vector `seed[0..15]`. This will be the initial state (or seed) of the next created stream. The default seed for the first stream is $(0, 0, \ldots, 0, 0)$.

`public void setSeed (byte seed[])`
This method is discouraged for normal use. Initializes the stream at the beginning of a stream with the initial seed `seed[0..15]`. This method only affects the specified stream; the others are not modified, so the beginning of the streams will not be spaced Z values apart. For this reason, this method should only be used in very exceptional cases; proper use of the `reset...` methods and of the stream constructor is preferable.

`public byte[] getState()`
Returns the current state of the stream, represented as an array of four integers. It should be noted that each state of this generator returns 4 successive values. The particular value of these 4 which will be returned next is not given by this method.

`public RandRijndael clone()`
Clones the current generator and return its copy.

References

[1] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1998.

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

[3] P. L'Ecuyer. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–213, 1996.

[4] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[5] P. L'Ecuyer. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, 68(225):261–269, 1999.

[6] P. L'Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*, pages 95–105, Piscataway, NJ, 2001. IEEE Press.

[7] P. L'Ecuyer. Random number generation. In J. E. Gentle, W. Haerdle, and Y. Mori, editors, *Handbook of Computational Statistics*, pages 35–70. Springer-Verlag, Berlin, 2004. Chapter II.2.

[8] P. L'Ecuyer and T. H. Andres. A random number generator based on the combination of four LCGs. *Mathematics and Computers in Simulation*, 44:99–107, 1997.

[9] P. L'Ecuyer and S. Côté. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[10] P. L'Ecuyer and J. Granger-Piché. Combined generators with components from different families. *Mathematics and Computers in Simulation*, 62:395–404, 2003.

[11] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

[12] P. L'Ecuyer and R. Touzin. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In *Proceedings of the 2000 Winter Simulation Conference*, pages 683–689, Piscataway, NJ, 2000. IEEE Press.

[13] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[14] F. Panneton. *Construction d'ensembles de points basée sur des récurrences linéaires dans un corps fini de caractéristique 2 pour la simulation Monte Carlo et l'intégration quasi-Monte Carlo*. PhD thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, Canada, August 2004.

[15] F. Panneton and P. L'Ecuyer. Random number generators based on linear recurrences in F_{2^w}. In H. Niederreiter, editor, *Monte Carlo and Quasi-Monte Carlo Methods 2002*, pages 367–378, Berlin, 2004. Springer-Verlag.

[16] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16, 2006.

[17] S. Tezuka and P. L'Ecuyer. Efficient and portable combined Tausworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, 1(2):99–112, 1991.