SSJ User's Guide

Package `stochprocess`

Stochastic Processes

Version: June 18, 2014

This package provides tools to generate various stochastic processes.

Contents

Overview

This package provides classes to define stochastic processes $\{X(t),\, t \geq 0\}$, and to simulate their sample paths at a finite number of (discrete) observation times $t_0 \leq t_1 \leq \cdots \leq t_d$. The observation of the generated path is thus the vector $(X(t_0), X(t_1), \ldots, X(t_d))$.

The observation times $t_0, \ldots, t_d$ can be specified (or changed) after defining the process, with the method `setObservationTimes`. The random stream used to generate the sample path can also be changed, using `setStream`.

StochasticProcess

Abstract base class for a stochastic process $\{X(t) : t \geq 0\}$ sampled (or observed) at a finite number of time points, $0 = t_0 < t_1 < \cdots < t_d$. The observation times are usually all specified before generating a sample path. This can be done via `setObservationTimes`. The method `generatePath` generates $X(t_1), \ldots, X(t_d)$ and memorizes them in a vector, which can be recovered by `getPath`.

Alternatively, for some types of processes, the observations $X(t_j)$ can be generated sequentially, one at a time, by invoking `resetStartProcess` first, and then `nextObservation` repeatedly. For some types of processes, the observation times can be specified one by one as well, when generating the path. This may be convenient or even necessary if the observation times are random, for example.

WARNING: After having called the constructor for one of the subclass, one must always set the observation times of the process, by calling method `setObservationTimes` for example or otherwise.

```
package umontreal.iro.lecuyer.stochprocess;

public abstract class StochasticProcess
```

Methods

`public void setObservationTimes (double[] T, int d)`

Sets the observation times of the process to a copy of `T`, with $t_0 =$ `T[0]` and $t_d =$ `T[d]`. The size of `T` must be $d + 1$.

`public void setObservationTimes (double delta, int d)`

Sets equidistant observation times at $t_j = j\delta$, for $j = 0, \ldots, d$, and `delta` $= \delta$.

`public double[] getObservationTimes()`

Returns a reference to the array that contains the observation times $(t_0, \ldots, t_d)$. *Warning*: This method should only be used to *read* the observation times. Changing the values in the array directly may have unexpected consequences. The method `setObservationTimes` should be used to modify the observation times.

`public int getNbObservationTimes()`

Returns the number of observation times excluding the time t_0.

`public abstract double[] generatePath();`

Generates, returns, and saves the sample path $\{X(t_0), X(t_1), \ldots, X(t_d)\}$. It can then be accessed via `getPath`, `getSubpath`, or `getObservation`. The generation method depends on the process type.

`public double[] generatePath (RandomStream stream)`

Same as `generatePath()`, but first resets the stream to `stream`.

`public double[] getPath()`

Returns a *reference* to the last generated sample path $\{X(t_0), ..., X(t_d)\}$. *Warning*: The returned array and its size should not be modified, because this is the one that memorizes the observations (not a copy of it). To obtain a copy, use `getSubpath` instead.

`public void getSubpath (double[] subpath, int[] pathIndices)`

Returns in `subpath` the values of the process at a subset of the observation times, specified as the times t_j whose indices j are in the array `pathIndices`. The size of `pathIndices` should be at least as much as that of `subpath`.

`public double getObservation (int j)`

Returns $X(t_j)$ from the current sample path. *Warning*: If the observation $X(t_j)$ for the current path has not yet been generated, then the value returned is unpredictable.

`public void resetStartProcess()`

Resets the observation counter to its initial value $j = 0$, so that the current observation $X(t_j)$ becomes $X(t_0)$. This method should be invoked before generating observations sequentially one by one via `nextObservation`, for a new sample path.

`public boolean hasNextObservation()`

Returns `true` if $j < d$, where j is the number of observations of the current sample path generated since the last call to `resetStartProcess`. Otherwise returns `false`.

`public double nextObservation()`

Generates and returns the next observation $X(t_j)$ of the stochastic process. The processes are usually sampled *sequentially*, i.e. if the last observation generated was for time t_{j-1}, the next observation returned will be for time t_j. In some cases, subclasses extending this abstract class may use non-sequential sampling algorithms (such as bridge sampling). The order of generation of the t_j's is then specified by the subclass. All the processes generated using principal components analysis (PCA) do not have this method.

`public int getCurrentObservationIndex()`

Returns the value of the index j corresponding to the time t_j of the last generated observation.

`public double getCurrentObservation()`

Returns the value of the last generated observation $X(t_j)$.

`public double getX0()`

Returns the initial value $X(t_0)$ for this process.

`public void setX0 (double s0)`

Sets the initial value $X(t_0)$ for this process to `s0`, and reinitializes.

`public abstract void setStream (RandomStream stream);`

Resets the random stream of the underlying generator to `stream`.

`public abstract RandomStream getStream();`

Returns the random stream of the underlying generator.

`public int[] getArrayMappingCounterToIndex()`

Returns a reference to an array that maps an integer k to i_k, the index of the observation $S(t_{i_k})$ corresponding to the k-th observation to be generated for a sample path of this process. If this process is sampled sequentially, then this map is trivial (i.e. $i_k = k$). But it can be useful in a more general setting where the process is not sampled sequentially (for example, by a Brownian or gamma bridge) and one wants to know which observations of the current sample path were previously generated or will be generated next.

BrownianMotion

This class represents a *Brownian motion* process $\{X(t) : t \geq 0\}$, sampled at times $0 = t_0 < t_1 < \cdots < t_d$. This process obeys the stochastic differential equation

$$dX(t) = \mu dt + \sigma dB(t), \tag{1}$$

with initial condition $X(0) = x_0$, where μ and σ are the drift and volatility parameters, and $\{B(t),\, t \geq 0\}$ is a standard Brownian motion (with drift 0 and volatility 1). This process has stationary and independent increments over disjoint time intervals (it is a Lévy process) and the increment over an interval of length t is normally distributed with mean μt and variance $\sigma^2 t$.

In this class, this process is generated using the sequential (or random walk) technique: $X(0) = x_0$ and

$$X(t_j) - X(t_{j-1}) = \mu(t_j - t_{j-1}) + \sigma\sqrt{t_j - t_{j-1}}\, Z_j \tag{2}$$

where $Z_j \sim N(0, 1)$.

```
package umontreal.iro.lecuyer.stochprocess;

public class BrownianMotion extends StochasticProcess
```

Constructors

`public BrownianMotion (double x0, double mu, double sigma, RandomStream stream)`

Constructs a new `BrownianMotion` with parameters $\mu =$ `mu`, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates Z_j in (2) will be generated by inversion using `stream`.

`public BrownianMotion (double x0, double mu, double sigma, NormalGen gen)`

Constructs a new `BrownianMotion` with parameters $\mu =$ `mu`, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. Here, the normal variate generator `NormalGen` is specified directly instead of specifying the stream and using inversion. The normal generator `gen` can use another method than inversion.

Methods

`public double nextObservation (double nextTime)`

Generates and returns the next observation at time $t_{j+1} =$ `nextTime`. It uses the previous observation time t_j defined earlier (either by this method or by `setObservationTimes`), as well as the value of the previous observation $X(t_j)$. *Warning*: This method will reset the observations time t_{j+1} for this process to `nextTime`. The user must make sure that the t_{j+1} supplied is $\geq t_j$.

`public double nextObservation (double x, double dt)`

Generates an observation of the process in `dt` time units, assuming that the process has value x at the current time. Uses the process parameters specified in the constructor. Note that this method does not affect the sample path of the process stored internally (if any).

`public double[] generatePath (double[] uniform01)`

Same as generatePath(), but a vector of uniform random numbers must be provided to the method. These uniform random numbers are used to generate the path.

`public void setParams (double x0, double mu, double sigma)`

Resets the parameters $X(t_0) =$ `x0`, $\mu =$ `mu` and $\sigma =$ `sigma` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called too frequently.

`public void setStream (RandomStream stream)`

Resets the random stream of the normal generator to `stream`.

`public RandomStream getStream()`

Returns the random stream of the normal generator.

`public double getMu()`

Returns the value of μ.

`public double getSigma()`

Returns the value of σ.

`public NormalGen getGen()`

Returns the normal random variate generator used. The `RandomStream` used by that generator can be changed via `getGen().setStream(stream)`, for example.

BrownianMotionBridge

Represents a Brownian motion process $\{X(t) : t \geq 0\}$ sampled using the *bridge sampling* technique (see for example [7]). This technique generates first the value $X(t_d)$ at the last observation time, then the value at time $t_{d/2}$ (or the nearest integer), then the values at time $t_{d/4}$ and at time $t_{3d/4}$ (or the nearest integers), and so on. If the process has already been sampled at times $t_i < t_k$ but not in between, the next sampling point in that interval will be t_j where $j = \lfloor (i+k)/2 \rfloor$. For example, if the sampling times used are $\{t_0, t_1, t_2, t_3, t_4, t_5\}$, then the observations are generated in the following order: $X(t_5)$, $X(t_2)$, $X(t_1)$, $X(t_3)$, $X(t_4)$.

Warning: Both the `generatePath` and the `nextObservation` methods from `BrownianMotion` are modified to use the bridge method. [1] In the case of `nextObservation`, the user should understand that the observations returned are *not* ordered chronologically. However they will be once an entire path is generated and the observations are read from the internal array (referenced by the `getPath` method) that contains them.

The method `nextObservation(double nextTime)` differs from that of the class `BrownianMotion` in that `nextTime` represents the next observation time *of the Brownian bridge*. However, the t_i supplied must still be non-decreasing with i.

Note also that, if the path is not entirely generated before being read from this array, there will be "pollution" from the previous path generated, and the observations will not represent a sample path of this process.

```
package umontreal.iro.lecuyer.stochprocess;

public class BrownianMotionBridge extends BrownianMotion
```

Constructors

```
public BrownianMotionBridge (double x0, double mu, double sigma,
                             RandomStream stream)
```

Constructs a new `BrownianMotionBridge` with parameters $\mu =$ `mu`, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates will be generated by inversion using the `RandomStream stream`.

```
public BrownianMotionBridge (double x0, double mu, double sigma,
                             NormalGen gen)
```

Constructs a new `BrownianMotionBridge` with parameters $\mu =$ `mu`, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates will be generated by the `NormalGen gen`.

[1] From Pierre: We should probably remove the `nextObservation` methods from here.

BrownianMotionPCA

A Brownian motion process $\{X(t) : t \geq 0\}$ sampled using the *principal component* decomposition (PCA) [7, 8, 9].

```
package umontreal.iro.lecuyer.stochprocess;

public class BrownianMotionPCA extends BrownianMotion
```

Constructors

`public BrownianMotionPCA (double x0, double mu, double sigma, RandomStream stream)`

Constructs a new `BrownianMotionBridge` with parameters $\mu =$ `mu`, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates will be generated by inversion using `stream`.

`public BrownianMotionPCA (double x0, double mu, double sigma, NormalGen gen)`

Constructs a new `BrownianMotionBridge` with parameters $\mu =$ `mu`, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates will be generated by `gen`.

Methods

`public double[] getSortedEigenvalues()`

Returns the sorted eigenvalues obtained in the PCA decomposition.

GeometricBrownianMotion

Represents a *geometric Brownian motion* (GBM) process $\{S(t),\, t \geq 0\}$, which evolves according to the stochastic differential equation

$$dS(t) = \mu S(t)dt + \sigma S(t)dB(t), \tag{3}$$

where μ and σ are the drift and volatility parameters, and $\{B(t),\, t \geq 0\}$ is a standard Brownian motion (for which $B(t) \sim N(0,t)$). This process can also be written as the exponential of a Brownian motion:

$$S(t) = S(0)\exp\left[(\mu - \sigma^2/2)t + \sigma t B(t)\right] = S(0)\exp\left[X(t)\right], \tag{4}$$

where $X(t) = (\mu - \sigma^2/2)t + \sigma t B(t)$. The GBM process is simulated by simulating the BM process X and taking the exponential. This BM process is stored internally.

```
package umontreal.iro.lecuyer.stochprocess;

public class GeometricBrownianMotion extends StochasticProcess
```

Constructors

```
public GeometricBrownianMotion (double s0, double mu, double sigma,
                                RandomStream stream)
```

Same as `GeometricBrownianMotion (s0, mu, sigma, new BrownianMotion (0.0, 0.0, 1.0, stream))`.

```
public GeometricBrownianMotion (double s0, double mu, double sigma,
                                BrownianMotion bm)
```

Constructs a new `GeometricBrownianMotion` with parameters $\mu =$ `mu`, $\sigma =$ `sigma`, and $S(t_0) =$ `s0`, using `bm` as the underlying `BrownianMotion`. The parameters of `bm` are automatically reset to $\mu - \sigma^2/2$ and σ, regardless of the original parameters of `bm`. The observation times are the same as those of `bm`. The generation method depends on that of `bm` (sequential, bridge sampling, PCA, etc.).

Methods

```
public void resetStartProcess()
```

Same as in `StochasticProcess`, but also invokes `resetStartProcess` for the underlying `BrownianMotion` object.

```
public void setParams (double s0, double mu, double sigma)
```

Sets the parameters $S(t_0) =$ `s0`, $\mu =$ `mu` and $\sigma =$ `sigma` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called repeatedly.

`public void setStream (RandomStream stream)`

Resets the `RandomStream` for the underlying Brownian motion to `stream`.

`public RandomStream getStream()`

Returns the `RandomStream` for the underlying Brownian motion.

`public double getMu()`

Returns the value of μ.

`public double getSigma()`

Returns the value of σ.

`public NormalGen getGen()`

Returns the `NormalGen` used.

`public BrownianMotion getBrownianMotion()`

Returns a reference to the `BrownianMotion` object used to generate the process.

GeometricLevyProcess

Abstract class used as a parent class for the exponentiation of a Lévy process $X(t)$:

$$S(t) = S(0) \exp\left(X(t) + (r - \omega_{RN})t\right). \tag{5}$$

The interest is here denoted r and is refered to as `muGeom` in the class below. The risk neutral correction is given by ω_{RN} and takes into account risk aversion in the pricing of assets; its value depends on the specific Lévy process that is used.

`GeometricNormalInverseGaussianProcess` is implemented as a child of this class and so could `GeometricVarianceGammaProcess` and `GeometricBrownianMotion`.

```
package umontreal.iro.lecuyer.stochprocess;

public abstract class GeometricLevyProcess extends StochasticProcess
```

Methods

`public double[] generatePath()`

Generates a path.

`public double nextObservation()`

Returns the next observation. It will also work on a Lévy process which is sampled using the bridge order, but it will return the observations in the bridge order. If the underlying Lévy process is of the PCA type, this method is not usable.

`public void resetStartProcess()`

Resets the step counter of the geometric process and the underlying Lévy process to the start value.

`public void setObservationTimes(double[] time, int d)`

Sets the observation times on the geometric process and the underlying Lévy process.

`public double getOmega()`

Returns the risk neutral correction.

`public double getMuGeom()`

Returns the geometric drift parameter, which is usually the interest rate, r.

`public void setMuGeom (double muGeom)`

Sets the drift parameter (interest rate) of the geometric term.

`public StochasticProcess getLevyProcess()`

Returns the Lévy process.

`public void resetRiskNeutralCorrection (double omegaRN)`

Changes the value of ω_{RN}. There should usually be no need to redefine the risk neutral correction from the value set by the constructor. However it is sometimes not unique, e.g. in `GeometricNormalInverseGaussianProcess` [1].

`public RandomStream getStream()`

Returns the stream from the underlying Lévy process. If the underlying Lévy process has multiple streams, it returns what the `getStream()` method of that process was made to return.

`public void setStream (RandomStream stream)`

Resets the stream in the underlying Lévy process. If the underlying Lévy process has multiple streams, it sets the streams on this process in the same way as `setStream()` for that process.

InverseGaussianProcess

The inverse Gaussian process is a non-decreasing process where the increments are additive and are given by the inverse gaussian distribution, `InverseGaussianDist`. With parameters δ and γ, the time increments are given by `InverseGaussianDist`$(\delta dt/\gamma, \delta^2 dt^2)$.

[We here use the inverse gaussian distribution parametrized with IGDist(μ, λ), where $\mu = \delta/\gamma$ and $\lambda = \delta^2$. If we instead used the parametrization IGDist$^\star(\delta, \gamma)$, then the increment distribution of our process would have been written more simply as IGDist$^\star(\delta dt, \gamma)$.]

The increments are generated by using the inversion of the cumulative distribution function. It therefore uses only one `RandomStream`. Subclasses of this class use different generating methods and some need two `RandomStream`'s.

The initial value of this process is the initial observation time.

```
package umontreal.iro.lecuyer.stochprocess;

public class InverseGaussianProcess extends StochasticProcess
```

Constructors

```
public InverseGaussianProcess (double s0, double delta, double gamma,
                               RandomStream stream)
```

Constructs a new `InverseGaussianProcess`. The initial value $s0$ will be overridden by $t[0]$ when the observation times are set.

Methods

```
public double[] generatePath (double[] uniforms01)
```

Instead of using the internal stream to generate the path, uses an array of uniforms $U[0, 1)$. The array should be of the length of the number of periods in the observation times. This method is useful for `NormalInverseGaussianProcess`.

```
public double[] generatePath (double[] uniforms01, double[] uniforms01b)
```

This method does not work for this class, but will be useful for the subclasses that require two streams.

```
public void setParams (double delta, double gamma)
```

Sets the parameters.

```
public double getDelta()
```

Returns δ.

```
public double getGamma()
```

Returns γ.

`public double getAnalyticAverage (double time)`

Returns the analytic average which is $\delta t/\gamma$, with $t =$ `time`.

`public double getAnalyticVariance (double time)`

Returns the analytic variance which is $(\delta t)^2$, with $t =$ `time`.

`public int getNumberOfRandomStreams()`

Returns the number of random streams of this process. It is useful because some subclasses use different number of streams. It returns 1 for `InverseGaussianProcess`.

InverseGaussianProcessBridge

Samples the path by bridge sampling: first finding the process value at the final time and then the middle time, etc. The method `nextObservation()` returns the path value in that non-sequential order. This class uses two `RandomStream`'s to generate a path [17].

```
package umontreal.iro.lecuyer.stochprocess;

public class InverseGaussianProcessBridge extends InverseGaussianProcessMSH
```

Constructors

```
public InverseGaussianProcessBridge (double s0, double delta,
                                     double gamma, RandomStream stream,
                                     RandomStream otherStream)
```

Constructs a new `InverseGaussianProcessBridge`. The initial value `s0` will be overridden by $t[0]$ when the observation times are set.

Methods

```
public double[] generatePath()
```

Generates the path. The two inner `RandomStream`'s are sampled alternatively.

```
public double[] generatePath (double[] unifNorm, double[] unifOther)
```

Instead of using the internal streams to generate the path, it uses two arrays of uniforms $U[0, 1)$. The length of the arrays `unifNorm` and `unifOther` should be equal to the number of time steps, excluding t_0.

```
public double nextObservation()
```

Returns the next observation in the bridge order, not the sequential order.

```
public RandomStream getStream()
```

Only returns a stream if both inner streams are the same.

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

Sets the streams.

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

Sets both inner streams to the same `stream`.

InverseGaussianProcessMSH

Uses a faster generating method (MSH) [15] than the simple inversion of the distribution function used by `InverseGaussianProcess`. It is about 60 times faster. However it requires two `RandomStream`'s instead of only one for `InverseGaussianProcess`. The second stream is called `otherStream` below and it is used to randomly choose between two roots at each time step.

```
package umontreal.iro.lecuyer.stochprocess;

public class InverseGaussianProcessMSH extends InverseGaussianProcess
```

Constructors

```
public InverseGaussianProcessMSH (double s0, double delta, double gamma,
                                  RandomStream stream,
                                  RandomStream otherStream)
```

Constructs a new `InverseGaussianProcessMSH`. The initial value `s0` will be overridden by $t[0]$ when the observation times are set.

Methods

```
public double[] generatePath()
```

Generates the path. It is done by successively calling `nextObservation()`, therefore the two `RandomStream`s are sampled alternatively.

```
public double[] generatePath (double[] unifNorm, double[] unifOther)
```

Instead of using the internal streams to generate the path, uses two arrays of uniforms $U[0, 1)$. The length of the arrays should be equal to the number of periods in the observation times. This method is useful for `NormalInverseGaussianProcess`.

```
public double[] generatePath (double[] uniforms01)
```

Not implemented, requires two `RandomStream`'s.

```
public RandomStream getStream()
```

Only returns a stream if both inner `RandomStream`'s are the same.

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

Sets the streams.

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

Sets both inner streams to `stream`.

```
public void setOtherStream (RandomStream otherStream)
```

Sets the `otherStream`, which is the stream used to choose between the two roots in the MSH method.

`public RandomStream getOtherStream()`

Returns the `otherStream`, which is the stream used to choose between the two quadratic roots from the MSH method.

`public void setNormalGen (NormalGen normalGen)`

Sets the normal generator. It also sets one of the two inner streams to the stream of the normal generator.

`public NormalGen getNormalGen()`

Returns the normal generator.

InverseGaussianProcessPCA

Approximates a principal component analysis (PCA) decomposition of the `InverseGaussianProcess`
The PCA decomposition of a `BrownianMotionPCA` with a covariance matrix identical to the one of our `InverseGaussianProcess` is used to generate the path of our `InverseGaussianProcess` [10]. Such a path is a perfectly random path and it is hoped that it will provide reduction in the simulation variance when using quasi-Monte Carlo.

The method `nextObservation()` cannot be used with PCA decompositions since the whole path must be generated at once.

```
package umontreal.iro.lecuyer.stochprocess;

public class InverseGaussianProcessPCA extends InverseGaussianProcess
```

Constructors

`public InverseGaussianProcessPCA (double s0, double delta, double gamma, RandomStream stream)`

Constructs a new `InverseGaussianProcessPCA`. The initial value `s0` will be overridden by $t[0]$ when the observation times are set.

Methods

`public double[] generatePath (double[] uniforms01)`

Instead of using the internal stream to generate the path, uses an array of uniforms $U[0,1)$. The length of the array should be equal to the length of the number of periods in the observation times. This method is useful for `NormalInverseGaussianProcess`.

`public double nextObservation()`

Not implementable for PCA.

`public void setObservationTimes (double t[], int d)`

Sets the observation times of both the `InverseGaussianProcessPCA` and the inner `BrownianMotionPCA`.

`public void setBrownianMotionPCA (BrownianMotionPCA bmPCA)`

Sets the brownian motion PCA. The observation times will be overriden when the method `observationTimes()` is called on the `InverseGaussianProcessPCA`.

`public BrownianMotion getBrownianMotionPCA()`

Returns the `BrownianMotionPCA`.

NormalInverseGaussianProcess

This class represents a normal inverse gaussian process (NIG). It obeys the stochastic differential equation [4]

$$dX(t) = \mu dt + dB(h(t)), \tag{6}$$

where $\{B(t), t \geq 0\}$ is a `BrownianMotion` with drift β and variance 1, and $h(t)$ is an `InverseGaussianProcess` $IG(\nu/\gamma, \nu^2)$, with $\nu = \delta dt$ and $\gamma = \sqrt{\alpha^2 - \beta^2}$.

In this class, the process is generated using the sequential technique: $X(0) = x_0$ and

$$X(t_j) - X(t_{j-1}) = \mu dt + \beta Y_j + \sqrt{Y_j} Z_j, \tag{7}$$

where $Z_j \sim N(0,1)$, and $Y_j \sim IG(\nu/\gamma, \nu^2)$ with $\nu = \delta(t_j - t_{j-1})$.

There is one `RandomStream` used to generate the Z_j's and there are one or two streams used to generate the underlying `InverseGaussianProcess`, depending on which IG subclass is used.

In finance, a NIG process usually means that the log-return is given by a NIG process; `GeometricNormalInverseGaussianProcess` should be used in that case.

```
package umontreal.iro.lecuyer.stochprocess;

public class NormalInverseGaussianProcess extends StochasticProcess
```

Constructors

```
public NormalInverseGaussianProcess (double x0, double alpha,
                                     double beta, double mu,
                                     double delta,
                                     RandomStream streamBrownian,
                                     InverseGaussianProcess igP)
```

Given an `InverseGaussianProcess` `igP`, constructs a new `NormalInverseGaussianProcess`. The parameters and observation times of the IG process will be overriden by the parameters of the NIG process. If there are two `RandomStream`'s in the `InverseGaussianProcess`, this constructor assumes that both streams have been set to the same stream.

```
public NormalInverseGaussianProcess (double x0, double alpha,
                                     double beta, double mu,
                                     double delta,
                                     RandomStream streamBrownian,
                                     RandomStream streamIG1,
                                     RandomStream streamIG2,
                                     String igType)
```

Constructs a new `NormalInverseGaussianProcess`. The string argument corresponds to the type of underlying `InverseGaussianProcess`. The choices are SEQUENTIAL_SLOW, SEQUENTIAL_MSH, BRIDGE and PCA, which correspond respectively to `InverseGaussianProcess`, `InverseGaussianProcessMSH`, `InverseGaussianProcessBridge` and `InverseGaussianProcessPCA`. The third `RandomStream`, `streamIG2`, will not be used at all if the SEQUENTIAL_SLOW or PCA methods are chosen.

```
public NormalInverseGaussianProcess (double x0, double alpha,
                                     double beta, double mu,
                                     double delta,
                                     RandomStream streamAll,
                                     String igType)
```

Same as above, but all `RandomStream`'s are set to the same stream, `streamAll`.

Methods

`public double[] generatePath()`

Generates the path. This method samples each stream alternatively, which is useful for quasi-Monte Carlo, where all streams are in fact the same iterator on a `PointSet`.

`public double nextObservation()`

Returns the value of the process for the next time step. If the underlying `InverseGaussianProcess` is of type `InverseGaussianProcessPCA`, this method cannot be used. It will work with `InverseGaussianProcessBridge`, but the return order of the observations is the bridge order.

`public void setObservationTimes(double t[], int d)`

Sets the observation times on the NIG process as usual, but also sets the observation times of the underlying `InverseGaussianProcess`. It furthermore sets the starting *value* of the `InverseGaussianProcess` to `t[0]`.

```
public void setParams (double x0, double alpha, double beta,
                       double mu, double delta)
```

Sets the parameters. Also, computes $\gamma = \sqrt{\alpha^2 - \beta^2}$.

`public double getAlpha()`

Returns alpha.

`public double getBeta()`

Returns beta.

`public double getMu()`

Returns mu.

`public double getDelta()`

Returns delta.

`public double getGamma()`

Returns gamma.

`public double getAnalyticAverage (double time)`

Returns the analytic average, which is $\mu t + \delta t \beta / \gamma$.

`public double getAnalyticVariance (double time)`

Returns the analytic variance, which is $\delta t \alpha^2 / \gamma^3$.

`public RandomStream getStream()`

Only returns the stream if all streams are equal, including the stream(s) in the underlying `InverseGaussianProcess`.

`public void setStream (RandomStream stream)`

Sets all internal streams to `stream`, including the stream(s) of the underlying `InverseGaussianProcess`.

GeometricNormalInverseGaussianProcess

The geometric normal inverse gaussian (GNIG) process is the exponentiation of a `NormalInverseGaussianProcess`:

$$S(t) = S_0 \exp\left[(r - \omega_{RN})t + \mathrm{NIG}(t; \alpha, \beta, \mu, \delta)\right], \tag{8}$$

where r is the interest rate. It is a strictly positive process, which is useful in finance. There is also a neutral correction in the exponential, $\omega_{RN} = \mu + \delta\gamma - \delta\sqrt{\alpha^2 - (1+\beta)^2}$, which takes into account the market price of risk. The underlying NIG process must start at zero, $\mathrm{NIG}(t_0) = 0$ and the initial time should also be set to zero, $t_0 = 0$, both for the NIG and GNIG.

```
package umontreal.iro.lecuyer.stochprocess;

public class GeometricNormalInverseGaussianProcess extends
                                                    GeometricLevyProcess
```

Constructors

```
public GeometricNormalInverseGaussianProcess (
                                        double s0, double muGeom,
                                        double alpha, double beta,
                                        double mu, double delta,
                                        RandomStream streamBrownian,
                                        NormalInverseGaussianProcess nigP)
```

Constructs a new `GeometricNormalInverseGaussianProcess`. The parameters of the NIG process will be overwritten by the parameters given to the GNIG, with the initial value of the NIG set to 0. The observation times of the NIG will also be changed to those of the GNIG.

```
public GeometricNormalInverseGaussianProcess (
                                        double s0, double muGeom,
                                        double alpha, double beta,
                                        double mu, double delta,
                                        RandomStream streamBrownian,
                                        InverseGaussianProcess igP)
```

Constructs a new `GeometricNormalInverseGaussianProcess`. The process `igP` will be used internally by the underlying `NormalInverseGaussianProcess`.

```
public GeometricNormalInverseGaussianProcess (
                                        double s0, double muGeom,
                                        double alpha, double beta,
                                        double mu, double delta,
                                        RandomStream streamBrownian,
                                        RandomStream streamNIG1,
                                        RandomStream streamNIG2,
                                        String igType)
```

Constructs a new `GeometricNormalInverseGaussianProcess`. The drift of the geometric term, `muGeom`, is usually the interest rate r. `s0` is the initial value of the process and the other

four parameters are the parameters of the underlying `NormalInverseGaussianProcess` process.

```
public GeometricNormalInverseGaussianProcess (
                                        double s0, double muGeom,
                                        double alpha, double beta,
                                        double mu, double delta,
                                        RandomStream streamAll,
                                        String igType)
```

Constructs a new `GeometricNormalInverseGaussianProcess`. The String `igType` corresponds to the type of `InverseGaussianProcess` that will be used by the underlying `NormalInverseGaussianProcess`. All `RandomStream`'s used to generate the underlying `NormalInverseGaussianProcess` and its underlying `InverseGaussianProcess` are set to the same given `streamAll`.

OrnsteinUhlenbeckProcess

This class represents an *Ornstein-Uhlenbeck* process $\{X(t) : t \geq 0\}$, sampled at times $0 = t_0 < t_1 < \cdots < t_d$. This process obeys the stochastic differential equation

$$dX(t) = \alpha(b - X(t))dt + \sigma\, dB(t) \tag{9}$$

with initial condition $X(0) = x_0$, where α, b and σ are positive constants, and $\{B(t),\, t \geq 0\}$ is a standard Brownian motion (with drift 0 and volatility 1). This process is *mean-reverting* in the sense that it always tends to drift toward its general mean b. The process is generated using the sequential technique [7, p. 110]

$$X(t_j) = e^{-\alpha(t_j - t_{j-1})} X(t_{j-1}) + b\left(1 - e^{-\alpha(t_j - t_{j-1})}\right) + \sigma\sqrt{\frac{1 - e^{-2\alpha(t_j - t_{j-1})}}{2\alpha}}\, Z_j \tag{10}$$

where $Z_j \sim N(0, 1)$. The time intervals $t_j - t_{j-1}$ can be arbitrarily large.

```
package umontreal.iro.lecuyer.stochprocess;

public class OrnsteinUhlenbeckProcess extends StochasticProcess
```

Constructors

```
public OrnsteinUhlenbeckProcess (double x0, double alpha, double b,
                                 double sigma, RandomStream stream)
```

Constructs a new `OrnsteinUhlenbeckProcess` with parameters $\alpha =$ `alpha`, b, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates Z_j will be generated by inversion using the stream `stream`.

```
public OrnsteinUhlenbeckProcess (double x0, double alpha, double b,
                                 double sigma, NormalGen gen)
```

Here, the normal variate generator is specified directly instead of specifying the stream. The normal generator `gen` can use another method than inversion.

Methods

```
public double nextObservation (double nextTime)
```

Generates and returns the next observation at time $t_{j+1} =$ `nextTime`, using the previous observation time t_j defined earlier (either by this method or by `setObservationTimes`), as well as the value of the previous observation $X(t_j)$. *Warning*: This method will reset the observations time t_{j+1} for this process to `nextTime`. The user must make sure that the t_{j+1} supplied is $\geq t_j$.

```
public double nextObservation (double x, double dt)
```

Generates an observation of the process in `dt` time units, assuming that the process has value x at the current time. Uses the process parameters specified in the constructor. Note that this method does not affect the sample path of the process stored internally (if any).

`public void setParams (double x0, double alpha, double b, double sigma)`

Resets the parameters $X(t_0) =$ `x0`, $\alpha =$ `alpha`, $b =$ `b` and $\sigma =$ `sigma` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called too frequently.

`public void setStream (RandomStream stream)`

Resets the random stream of the normal generator to `stream`.

`public RandomStream getStream ()`

Returns the random stream of the normal generator.

`public double getAlpha()`

Returns the value of α.

`public double getB()`

Returns the value of b.

`public double getSigma()`

Returns the value of σ.

`public NormalGen getGen()`

Returns the normal random variate generator used. The `RandomStream` used for that generator can be changed via `getGen().setStream(stream)`, for example.

OrnsteinUhlenbeckProcessEuler

This class represents an *Ornstein-Uhlenbeck* process as in `OrnsteinUhlenbeckProcess`, but the process is generated using the simple Euler scheme

$$X(t_j) - X(t_{j-1}) = \alpha(b - X(t_{j-1}))(t_j - t_{j-1}) + \sigma\sqrt{t_j - t_{j-1}}\, Z_j \tag{11}$$

where $Z_j \sim N(0,1)$. This is a good approximation only for small time intervals $t_j - t_{j-1}$.

```
package umontreal.iro.lecuyer.stochprocess;

public class OrnsteinUhlenbeckProcessEuler extends OrnsteinUhlenbeckProcess
```

Constructors

```
public OrnsteinUhlenbeckProcessEuler (double x0, double alpha, double b,
                                      double sigma, RandomStream stream)
```

Constructor with parameters $\alpha =$ `alpha`, b, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates Z_j will be generated by inversion using the stream `stream`.

```
public OrnsteinUhlenbeckProcessEuler (double x0, double alpha, double b,
                                      double sigma, NormalGen gen)
```

Here, the normal variate generator is specified directly instead of specifying the stream. The normal generator `gen` can use another method than inversion.

CIRProcess

This class represents a *CIR* (Cox, Ingersoll, Ross) process [5] $\{X(t) : t \geq 0\}$, sampled at times $0 = t_0 < t_1 < \cdots < t_d$. This process obeys the stochastic differential equation

$$dX(t) = \alpha(b - X(t))dt + \sigma\sqrt{X(t)}\, dB(t) \tag{12}$$

with initial condition $X(0) = x_0$, where α, b and σ are positive constants, and $\{B(t),\, t \geq 0\}$ is a standard Brownian motion (with drift 0 and volatility 1). This process is *mean-reverting* in the sense that it always tends to drift toward its general mean b. The process is generated using the sequential technique [7, p. 122]

$$X(t_j) = \frac{\sigma^2\left(1 - e^{-\alpha(t_j - t_{j-1})}\right)}{4\alpha} \chi'^2_\nu \left(\frac{4\alpha e^{-\alpha(t_j - t_{j-1})} X(t_{j-1})}{\sigma^2\left(1 - e^{-\alpha(t_j - t_{j-1})}\right)} \right), \tag{13}$$

where $\nu = 4b\alpha/\sigma^2$, and $\chi'^2_\nu(\lambda)$ is a noncentral chi-square random variable with ν degrees of freedom and noncentrality parameter λ.

```
package umontreal.iro.lecuyer.stochprocess;

public class CIRProcess extends StochasticProcess
```

Constructors

`public CIRProcess (double x0, double alpha, double b, double sigma, RandomStream stream)`

Constructs a new `CIRProcess` with parameters $\alpha =$ `alpha`, b, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The noncentral chi-square variates $\chi'^2_\nu(\lambda)$ will be generated by inversion using the stream `stream`.

`public CIRProcess (double x0, double alpha, double b, double sigma, ChiSquareNoncentralGen gen)`

The noncentral chi-square variate generator `gen` is specified directly instead of specifying the stream. `gen` can use a method other than inversion.

Methods

`public double nextObservation (double nextTime)`

Generates and returns the next observation at time $t_{j+1} =$ `nextTime`, using the previous observation time t_j defined earlier (either by this method or by `setObservationTimes`), as well as the value of the previous observation $X(t_j)$. *Warning*: This method will reset the observations time t_{j+1} for this process to `nextTime`. The user must make sure that the t_{j+1} supplied is $\geq t_j$.

`public double nextObservation (double x, double dt)`

Generates an observation of the process in `dt` time units, assuming that the process has value x at the current time. Uses the process parameters specified in the constructor. Note that this method does not affect the sample path of the process stored internally (if any).

`public void setParams (double x0, double alpha, double b, double sigma)`

Resets the parameters $X(t_0) =$ `x0`, $\alpha =$ `alpha`, $b =$ `b` and $\sigma =$ `sigma` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called too frequently.

`public void setStream (RandomStream stream)`

Resets the random stream of the noncentral chi-square generator to `stream`.

`public RandomStream getStream()`

Returns the random stream of the noncentral chi-square generator.

`public double getAlpha()`

Returns the value of α.

`public double getB()`

Returns the value of b.

`public double getSigma()`

Returns the value of σ.

`public ChiSquareNoncentralGen getGen()`

Returns the noncentral chi-square random variate generator used. The `RandomStream` used for that generator can be changed via `getGen().setStream(stream)`, for example.

CIRProcessEuler

This class represents a *CIR* process as in `CIRProcess`, but the process is generated using the simple Euler scheme

$$X(t_j) - X(t_{j-1}) = \alpha(b - X(t_{j-1}))(t_j - t_{j-1}) + \sigma\sqrt{(t_j - t_{j-1})X(t_{j-1})}\, Z_j \tag{14}$$

where $Z_j \sim N(0,1)$. This is a good approximation only for small time intervals $t_j - t_{j-1}$.

```
package umontreal.iro.lecuyer.stochprocess;

public class CIRProcessEuler extends StochasticProcess
```

Constructors

`public CIRProcessEuler (double x0, double alpha, double b, double sigma, RandomStream stream)`

Constructs a new `CIRProcessEuler` with parameters $\alpha =$ `alpha`, b, $\sigma =$ `sigma` and initial value $X(t_0) =$ `x0`. The normal variates Z_j will be generated by inversion using the stream `stream`.

`public CIRProcessEuler (double x0, double alpha, double b, double sigma, NormalGen gen)`

The normal variate generator `gen` is specified directly instead of specifying the stream. `gen` can use another method than inversion.

Methods

`public double nextObservation (double nextTime)`

Generates and returns the next observation at time $t_{j+1} =$ `nextTime`, using the previous observation time t_j defined earlier (either by this method or by `setObservationTimes`), as well as the value of the previous observation $X(t_j)$. *Warning*: This method will reset the observations time t_{j+1} for this process to `nextTime`. The user must make sure that the t_{j+1} supplied is $\geq t_j$.

`public double nextObservation (double x, double dt)`

Generates an observation of the process in `dt` time units, assuming that the process has value x at the current time. Uses the process parameters specified in the constructor. Note that this method does not affect the sample path of the process stored internally (if any).

`public void setParams (double x0, double alpha, double b, double sigma)`

Resets the parameters $X(t_0) =$ `x0`, $\alpha =$ `alpha`, $b =$ `b` and $\sigma =$ `sigma` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called too frequently.

`public void setStream (RandomStream stream)`

Resets the random stream of the normal generator to `stream`.

`public RandomStream getStream()`

Returns the random stream of the normal generator.

`public double getAlpha()`

Returns the value of α.

`public double getB()`

Returns the value of b.

`public double getSigma()`

Returns the value of σ.

`public NormalGen getGen()`

Returns the normal random variate generator used. The `RandomStream` used for that generator can be changed via `getGen().setStream(stream)`, for example.

GammaProcess

This class represents a *gamma* process [12, page 82] $\{S(t) = G(t; \mu, \nu) : t \geq 0\}$ with mean parameter μ and variance parameter ν. It is a continuous-time process with stationary, independent gamma increments such that for any $\Delta t > 0$,

$$S(t + \Delta t) = S(t) + X, \tag{15}$$

where X is a random variate from the gamma distribution $\texttt{Gamma}(\mu^2 \Delta t/\nu, \mu/\nu)$.

In this class, the gamma process is sampled sequentially using equation (15).

```
package umontreal.iro.lecuyer.stochprocess;

public class GammaProcess extends StochasticProcess
```

Constructors

`public GammaProcess (double s0, double mu, double nu, RandomStream stream)`

Constructs a new `GammaProcess` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. The gamma variates X in (15) are generated by inversion using `stream`.

`public GammaProcess (double s0, double mu, double nu, GammaGen Ggen)`

Constructs a new `GammaProcess` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. The gamma variates X in (15) are supplied by the gamma random variate generator `Ggen`. Note that the parameters of the `GammaGen` object `Ggen` are not important since the implementation forces the generator to use the correct parameters (as defined above).

Methods

`public double nextObservation (double nextT)`

Generates and returns the next observation at time $t_{j+1} =$ `nextTime`, using the previous observation time t_j defined earlier (either by this method or by `setObservationTimes`), as well as the value of the previous observation $X(t_j)$. *Warning*: This method will reset the observations time t_{j+1} for this process to `nextT`. The user must make sure that the t_{j+1} supplied is $\geq t_j$.

`public double[] generatePath()`

Generates, returns and saves the path $\{X(t_0), X(t_1), \ldots, X(t_d)\}$. The gamma variates X in (15) are generated using the `RandomStream stream` or the `RandomStream` included in the `GammaGen Ggen`.

`public double[] generatePath (double[] uniform01)`

Generates, returns and saves the path $\{X(t_0), X(t_1), \ldots, X(t_d)\}$. This method does not use the `RandomStream stream` nor the `GammaGen Ggen`. It uses the vector of uniform random

numbers $U(0,1)$ provided by the user and generates the path by inversion. The vector `uniform01` must be of dimension d.

`public void setParams (double s0, double mu, double nu)`

Sets the parameters $S(t_0) =$ `s0`, $\mu =$ `mu` and $\nu =$ `nu` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called repeatedly.

`public double getMu()`

Returns the value of the parameter μ.

`public double getNu()`

Returns the value of the parameter ν.

`public void setStream (RandomStream stream)`

Resets the `RandomStream` of the `GammaGen` to `stream`.

`public RandomStream getStream()`

Returns the `RandomStream stream`.

GammaProcessBridge

This class represents a gamma process $\{S(t) = G(t; \mu, \nu) : t \geq 0\}$ with mean parameter μ and variance parameter ν, sampled using the *gamma bridge* method (see for example [16, 3]). This is analogous to the bridge sampling used in `BrownianMotionBridge`.

Note that gamma bridge sampling requires not only gamma variates, but also *beta* variates. The latter generally take a longer time to generate than the former. The class `GammaSymmetricalBridgeProcess` provides a faster implementation when the number of observation times is a power of two.

The warning from class `BrownianMotionBridge` applies verbatim to this class.

```
package umontreal.iro.lecuyer.stochprocess;

public class GammaProcessBridge extends GammaProcess
```

Constructors

```
public GammaProcessBridge (double s0, double mu, double nu,
                           RandomStream stream)
```

Constructs a new `GammaProcessBridge` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. Uses `stream` to generate the gamma and beta variates by inversion.

```
public GammaProcessBridge (double s0, double mu, double nu,
                           GammaGen Ggen, BetaGen Bgen)
```

Constructs a new `GammaProcessBridge`. Uses the random variate generators `Ggen` and `Bgen` to generate the gamma and beta variates, respectively. Note that both generator uses the same `RandomStream`. Furthermore, the parameters of the `GammaGen` and `BetaGen` objects are not important since the implementation forces the generators to use the correct parameters. (as defined in [16, page 7]).

Methods

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

Resets the `RandomStream` of the `GammaGen` and the `BetaGen` to `stream`.

GammaProcessSymmetricalBridge

This class differs from `GammaProcessBridge` only in that it requires the number of interval of the path to be a power of 2 and of equal size. It is then possible to generate the bridge process using a special implementation of the beta random variate generator (using the *symmetrical* beta distribution) that is much faster (HOW MUCH? QUANTIFY!) than the general case. Note that when the method `setObservationTimes` is called, the equality of the size of the time steps is verified. To allow for differences due to floating point errors, time steps are considered to be equal if their relative difference is less than 10^{-15}.

```
package umontreal.iro.lecuyer.stochprocess;

public class GammaProcessSymmetricalBridge extends GammaProcessBridge
```

Constructors

```
public GammaProcessSymmetricalBridge (double s0, double mu, double nu,
                                      RandomStream stream)
```

Constructs a new `GammaProcessSymmetricalBridge` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. The random variables are created using the `RandomStream stream`. Note that the same `RandomStream stream` is used for the `GammaGen` and for the `BetaSymmetricalGen` inluded in this class.

```
public GammaProcessSymmetricalBridge (double s0, double mu, double nu,
                                      GammaGen Ggen,
                                      BetaSymmetricalGen BSgen)
```

Constructs a new `GammaProcessSymmetricalBridge` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. Note that the `RandomStream` included in the `BetaSymmetricalGen` is sets to the one included in the `GammaGen` to avoid confusion. This `RandomStream` is then used to generate all the random variables.

GammaProcessPCA

Represents a *gamma* process sampled using the principal component analysis (PCA). To simulate the gamma process at times $t_0 < t_1 < \cdots < t_d$ by PCA sampling, a Brownian motion $\{W(t), t \geq 0\}$ with mean 0 and variance parameter ν is first generated at times $t_0 < t_1 < \cdots < t_d$ by PCA sampling (see class `BrownianMotionPCA`). The independent increments $W(t_j) - W(t_{j-1})$ of this process are then transformed into independent $U(0,1)$ random variates V_j via

$$V_j = \Phi\left(\sqrt{\tau_j - \tau_{j-1}}[W(\tau_j) - W(\tau_{j-1})]\right), \quad j = 1, \ldots, s$$

Finally, the increments of the Gamma process are computed as $Y(t_j) - Y(t_{j-1}) = G^{-1}(V_j)$, where G is the gamma distribution function.

```
package umontreal.iro.lecuyer.stochprocess;

public class GammaProcessPCA extends GammaProcess
```

Constructors

`public GammaProcessPCA (double s0, double mu, double nu, RandomStream stream)`

Constructs a new `GammaProcessPCA` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. The random variables are created using `stream`. Note that the same `RandomStream` is used for the `GammaProcessPCA` and for the `BrownianMotionPCA` included in this class. Both the `GammaProcessPCA` and the `BrownianMotionPCA` are generated by inversion.

`public GammaProcessPCA (double s0, double mu, double nu, GammaGen Ggen)`

Constructs a new `GammaProcessPCA` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. All the random variables, i.e. the gamma ones and the normal ones, are created using the `RandomStream` included in the `GammaGen Ggen`. Note that the parameters of the `GammaGen` object are not important since the implementation forces the generator to use the correct parameters (as defined above).

Methods

`public double nextObservation()`

This method is not implemented in this class since the path cannot be generated sequentially.

`public double nextObservation (double nextT)`

This method is not implemented in this class since the path cannot be generated sequentially.

`public BrownianMotionPCA getBMPCA()`

Returns the `BrownianMotionPCA` that is included in the `GammaProcessPCA` object.

`public void setObservationTimes (double[] t, int d)`

Sets the observation times of the `GammaProcessPCA` and the `BrownianMotionPCA`.

`public void setParams (double s0, double mu, double nu)`

Sets the parameters `s0`, μ and ν to new values, and sets the variance parameters of the `BrownianMotionPCA` to ν.

`public void setStream (RandomStream stream)`

Resets the `RandomStream` of the gamma generator and the `RandomStream` of the inner `BrownianMotionPCA` to `stream`.

GammaProcessPCABridge

Same as `GammaProcessPCA`, but the generated uniforms correspond to a bridge transformation of the `BrownianMotionPCA` instead of a sequential transformation.

```
package umontreal.iro.lecuyer.stochprocess;

public class GammaProcessPCABridge extends GammaProcessPCA
```

Constructors

```
public GammaProcessPCABridge (double s0, double mu, double nu,
                              RandomStream stream)
```

Constructs a new `GammaProcessPCABridge` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. The same stream is used to generate the gamma and beta random numbers. All these numbers are generated by inversion in the following order: the first uniform random number generated is used for the gamma and the other $d - 1$ for the beta's.

Methods

```
public BrownianMotionPCA getBMPCA()
```

Returns the inner `BrownianMotionPCA`.

GammaProcessPCASymmetricalBridge

Same as `GammaProcessPCABridge`, but uses the fast inversion method for the symmetrical beta distribution, proposed by L'Ecuyer and Simard [11], to accelerate the generation of the beta random variables. This class works only in the case where the number of intervals is a power of 2 and all these intervals are of equal size.

```
package umontreal.iro.lecuyer.stochprocess;

public class GammaProcessPCASymmetricalBridge extends GammaProcessPCABridge
```

Constructors

```
public GammaProcessPCASymmetricalBridge (double s0, double mu, double nu,
                                         RandomStream stream)
```

Constructs a new `GammaProcessPCASymmetricalBridge` with parameters $\mu =$ `mu`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. The `RandomStream stream` is used in the `GammaGen` and in the `BetaSymmetricalGen`. These two generators use inversion to generate random numbers. The first uniform random number generated by `stream` is used for the gamma, and the other $d - 1$ for the beta's.

VarianceGammaProcess

This class represents a *variance gamma* (VG) process $\{S(t) = X(t;\theta,\sigma,\nu) : t \geq 0\}$. This process is obtained as a subordinate of the Brownian motion process $B(t;\theta,\sigma)$ using the operational time $G(t;1,\nu)$ (see [6, 2]):

$$X(t;\theta,\sigma,\nu) := B(G(t;1,\nu),\theta,\sigma). \tag{16}$$

See also [12, 13, 14] for applications to modelling asset returns and option pricing.

The process is sampled as follows: when `generatePath()` is called, the method `generatePath()` of the inner `GammaProcess` is called; its path is then used to set the observation times of the `BrownianMotion`. Finally, the method `generatePath()` of the `BrownianMotion` is called. *Warning*: If one wants to reduced the variance as much as possible in a QMC simulation, this way of proceeding is not optimal. Use the method `generatePath(uniform01)` instead.

If one calls the `nextObservation` method, the operational time is generated first, followed by the corresponding brownian motion increment, which is then returned.

Note that if one wishes to use *bridge* sampling with the `nextObservation` method, both the gamma process G and the Brownian motion process B should use bridge sampling so that their observations are synchronized.

```
package umontreal.iro.lecuyer.stochprocess;

public class VarianceGammaProcess extends StochasticProcess
```

Constructors

```
public VarianceGammaProcess (double s0, double theta, double sigma,
                             double nu, RandomStream stream)
```

Constructs a new `VarianceGammaProcess` with parameters $\theta =$ `theta`, $\sigma =$ `sigma`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. `stream` is used to generate both the `BrownianMotion` B and the `GammaProcess` G in (16).

```
public VarianceGammaProcess (double s0, BrownianMotion BM,
                             GammaProcess Gamma)
```

Constructs a new `VarianceGammaProcess`. The parameters θ and σ are set to the parameters μ and σ, respectively, of the `BrownianMotion BM` and the parameter ν is set to the parameter ν of the `GammaProcess Gamma`. The parameters μ and $x0$ of the `GammaProcess` are overwritten to equal 1 and 0 respectively. The initial value of the process is $S(t_0) =$ `s0`.

Methods

`public double nextObservation()`

Generates the observation for the next time. It also works with bridge sampling; however *both* `BrownianMotionBridge` and `GammaProcessBridge` must be used in the constructor in that case. Furthermore, for bridge sampling, the order of the observations is that of the bridge, not sequential order.

`public double[] generatePath()`

Generates and returns the path. To do so, it first generates the complete path of the inner `GammaProcess` and sets the observation times of the inner `BrownianMotion` to this path. This method is not optimal to reduce the variance in QMC simulations; use `generatePath(double[] uniform01)` for that.

`public double[] generatePath (double[] uniform01)`

Similar to the usual `generatePath()`, but here the uniform random numbers used for the simulation must be provided to the method. This allows to properly use the uniform random variates in QMC simulations. This method divides the table of uniform random numbers `uniform01` in two smaller tables, the first one, containing the odd indices of `uniform01` which are used to generate the path of the inner `GammaProcess`, and the even indices (in the second table) are used to generate the path of the inner `BrownianMotion`. This way of proceeding reduces the variance as much as possible for QMC simulations.

`public void resetStartProcess()`

Resets the observation index and counter to 0 and applies the `resetStartProcess` method to the `BrownianMotion` and the `GammaProcess` objects used to generate this process.

`public void setParams (double s0, double theta, double sigma, double nu)`

Sets the parameters $S(t_0) =$ `s0`, $\theta =$ `theta`, $\sigma =$ `sigma` and $\nu =$ `nu` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called repeatedly.

`public double getTheta()`

Returns the value of the parameter θ.

`public double getSigma()`

Returns the value of the parameter σ.

`public double getNu()`

Returns the value of the parameter ν.

`public void setObservationTimes (double t[], int d)`

Sets the observation times on the `VarianceGammaProcess` as usual, but also sets the observation times of the underlying `GammaProcess`. It furthermore sets the starting *value* of the `GammaProcess` to `t[0]`.

`public void setStream (RandomStream stream)`

Resets the `RandomStream`'s. Warning: this method sets both the `RandomStream` of the `BrownianMotion` and of the `GammaProcess` to the same `RandomStream`.

`public RandomStream getStream()`

Returns the random stream of the `BrownianMotion` process, which should be the same as for the `GammaProcess`.

`public BrownianMotion getBrownianMotion()`

Returns a reference to the inner `BrownianMotion`.

`public GammaProcess getGammaProcess()`

Returns a reference to the inner `GammaProcess`.

VarianceGammaProcessDiff

This class represents a *variance gamma* (VG) process $\{S(t) = X(t;\theta,\sigma,\nu) : t \geq 0\}$. This process is generated using *difference of gamma sampling* (see [3, 2]), which uses the representation of the VG process as the difference of two independent `GammaProcess`'es (see [12]):

$$X(t;\theta,\sigma,\nu) := X(0) + \Gamma^+(t;\mu_p,\nu_p) - \Gamma^-(t;\mu_n,\nu_n) \tag{17}$$

where $X(0)$ is a constant corresponding to the initial value of the process and

$$\begin{aligned} \mu_p &= (\sqrt{\theta^2 + 2\sigma^2/\nu} + \theta)/2 \\ \mu_n &= (\sqrt{\theta^2 + 2\sigma^2/\nu} - \theta)/2 \\ \nu_p &= \nu\mu_p^2 \\ \nu_n &= \nu\mu_n^2 \end{aligned} \tag{18}$$

```
package umontreal.iro.lecuyer.stochprocess;

public class VarianceGammaProcessDiff extends VarianceGammaProcess
```

Constructors

```
public VarianceGammaProcessDiff (double s0, double theta, double sigma,
                                 double nu, RandomStream stream)
```

Constructs a new `VarianceGammaProcessDiff` with parameters $\theta =$ `theta`, $\sigma =$ `sigma`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. `stream` is used by two instances of `GammaProcess`, Γ^+ and Γ^-, respectively. The other parameters are as in the class `VarianceGammaProcess`. The `GammaProcess` objects for Γ^+ and Γ^- are constructed using the parameters from (18) and their initial values $\Gamma^+(t_0)$ and $\Gamma^-(t_0)$ are set to 0.

```
public VarianceGammaProcessDiff (double s0, double theta, double sigma,
                                 double nu, GammaProcess gpos,
                                 GammaProcess gneg)
```

The parameters of the `GammaProcess` objects for Γ^+ and Γ^- are set to those of (18) and their initial values $\Gamma^+(t_0)$ and $\Gamma^-(t_0)$ are set to t_0. The `RandomStream` of the Γ^- process is overwritten with the `RandomStream` of the Γ^+ process.

Methods

```
public double[] generatePath()
```

Generates, returns and saves the path. To do so, the path of Γ^+ is first generated and then the path of Γ^-. This is not the optimal way of proceeding in order to reduce the variance in QMC simulations; for that, use `generatePath(double[] uniform01)` instead.

`public double[] generatePath (double[] uniform01)`

Similar to the usual `generatePath()`, but here the uniform random numbers used for the simulation must be provided to the method. This allows to properly use the uniform random variates in QMC simulations. This method divides the table of uniform random numbers `uniform01` in two smaller tables, the first one containing the odd indices of `uniform01` are used to generate the path of Γ^+ and the even indices are used to generate the path of Γ^-. This way of proceeding further reduces the variance for QMC simulations.

`public void resetStartProcess()`

Sets the observation times on the `VarianceGammaProcessDiff` as usual, but also applies the `resetStartProcess` method to the two `GammaProcess` objects used to generate this process.

`public GammaProcess getGpos()`

Returns a reference to the `GammaProcess` object `gpos` used to generate the Γ^+ component of the process.

`public GammaProcess getGneg()`

Returns a reference to the `GammaProcess` object `gneg` used to generate the Γ^- component of the process.

`public void setObservationTimes (double t[], int d)`

Sets the observation times on the `VarianceGammaProcesDiff` as usual, but also sets the observation times of the underlying `GammaProcess`'es.

`public RandomStream getStream()`

Returns the `RandomStream` of the Γ^+ process.

`public void setStream (RandomStream stream)`

Sets the `RandomStream` of the two `GammaProcess`'es to `stream`.

VarianceGammaProcessDiffPCA

Same as `VarianceGammaProcessDiff`, but the two inner `GammaProcess`'es are of PCA type. Also, `generatePath(double[] uniforms01)` distributes the uniform random variates to the `GammaProcessPCA`'s according to their eigenvalues, i.e. the `GammaProcessPCA` with the higher eigenvalue gets the next uniform random number. If one should decide to create a `VarianceGammaProcessDiffPCA` by giving two `GammaProcessPCA`'s to an objet of the class `VarianceGammaProcessDiff`, the uniform random numbers would not be given this way to the `GammaProcessPCA`'s; this might give less variance reduction when used with QMC.

```
package umontreal.iro.lecuyer.stochprocess;

public class VarianceGammaProcessDiffPCA extends VarianceGammaProcessDiff
```

Constructors

```
public VarianceGammaProcessDiffPCA (double s0, double theta,
                                    double sigma, double nu,
                                    RandomStream stream)
```

Constructs a new `VarianceGammaProcessDiffPCA` with parameters θ = `theta`, σ = `sigma`, ν = `nu` and initial value $S(t_0)$ = `s0`. There is only one `RandomStream` here which is used for the two inner `GammaProcessPCA`'s. The other parameters are set as in `VarianceGammaProcessDiff`.

```
public VarianceGammaProcessDiffPCA (double s0, double theta,
                                    double sigma, double nu,
                                    GammaProcessPCA gpos,
                                    GammaProcessPCA gneg)
```

Constructs a new `VarianceGammaProcessDiffPCA` with parameters θ = `theta`, σ = `sigma`, ν = `nu` and initial value $S(t_0)$ = `s0`. As in `VarianceGammaProcessDiff`, the `RandomStream` of `gneg` is replaced by the one of `gpos` to avoid any confusion.

Methods

```
public double nextObservation()
```

This method is not implemented is this class since the path cannot be generated sequentially.

VarianceGammaProcessDiffPCABridge

Same as `VarianceGammaProcessDiff`, but the two inner `GammaProcess`'es are of the type PCABridge. Also, `generatePath(double[] uniform01)` distributes the lowest coordinates uniforms to the inner `GammaProcessPCABridge` according to their eigenvalues.

```
package umontreal.iro.lecuyer.stochprocess;

public class VarianceGammaProcessDiffPCABridge extends
                                              VarianceGammaProcessDiffPCA
```

Constructors

```
public VarianceGammaProcessDiffPCABridge (double s0, double theta,
                                          double sigma, double nu,
                                          RandomStream stream)
```

Constructs a new `VarianceGammaProcessDiffPCABridge` with parameters $\theta =$ `theta`, $\sigma =$ `sigma`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. There is only one `RandomStream` here which is used for the two inner `GammaProcessPCABridge`'s. The other parameters are set as in `VarianceGammaProcessDiff`.

VarianceGammaProcessDiffPCASymmetricalBridge

Same as `VarianceGammaProcessDiff`, but the two inner `GammaProcess`'es are of the PCASymmetricalBridge type. Also, `generatePath(double[] uniform01)` distributes the lowest coordinates uniforms to the inner GammaProcessPCA according to their eigenvalues.

```
package umontreal.iro.lecuyer.stochprocess;

public class VarianceGammaProcessDiffPCASymmetricalBridge extends
             VarianceGammaProcessDiffPCA
```

Constructors

```
public VarianceGammaProcessDiffPCASymmetricalBridge (
                                                 double s0, double theta,
                                                 double sigma, double nu,
                                                 RandomStream stream)
```

Constructs a new `VarianceGammaProcessDiffPCASymmetricalBridge` with parameters $\theta =$ `theta`, $\sigma =$ `sigma`, $\nu =$ `nu` and initial value $S(t_0) =$ `s0`. There is only one `RandomStream` here which is used for the two inner `GammaProcessPCASymmetricalBridge`'s. The other parameters are set as in `VarianceGammaProcessDiff`.

GeometricVarianceGammaProcess

This class represents a *geometric variance gamma* process $S(t)$ (see [12, page 86]). This stochastic process is defined by the equation

$$S(t) = S(0)\,\exp(\mu t + X(t;\sigma,\nu,\theta) + \omega t), \tag{19}$$

where X is a variance gamma process and

$$\omega = (1/\nu)\,\ln(1 - \theta\nu - \sigma^2\nu/2). \tag{20}$$

```
package umontreal.iro.lecuyer.stochprocess;

public class GeometricVarianceGammaProcess extends StochasticProcess
```

Constructors

```
public GeometricVarianceGammaProcess (double s0, double theta,
                                      double sigma, double nu,
                                      double mu, RandomStream stream)
```

Constructs a new `GeometricVarianceGammaProcess` with parameters $\theta =$ `theta`, $\sigma =$ `sigma`, $\nu =$ `nu`, $\mu =$ `mu` and initial value $S(t_0) =$ `s0`. The `stream` is used to generate the `VarianceGammaProcess` object used to implement X in (19).

```
public GeometricVarianceGammaProcess (double s0, double mu,
                                      VarianceGammaProcess vargamma)
```

Constructs a new `GeometricVarianceGammaProcess`. The parameters θ, σ, ν are set to the parameters of the `VarianceGammaProcess vargamma`. The parameter μ is set to `mu` and the initial values $S(t_0) =$ `s0`.

Methods

```
public void resetStartProcess()
```

Resets the `GeometricaVarianceGammaProcess`, but also applies the `resetStartProcess` method to the `VarianceGammaProcess` object used to generate this process.

```
public void setParams (double s0, double theta, double sigma, double nu,
                       double mu)
```

Sets the parameters $S(t_0) =$ `s0`, $\theta =$ `theta`, $\sigma =$ `sigma`, $\nu =$ `nu` and $\mu =$ `mu` of the process. *Warning*: This method will recompute some quantities stored internally, which may be slow if called repeatedly.

```
public double getTheta()
```

Returns the value of the parameter θ.

`public double getMu()`

Returns the value of the parameter μ.

`public double getNu()`

Returns the value of the parameter ν.

`public double getSigma()`

Returns the value of the parameter σ.

`public double getOmega()`

Returns the value of the quantity ω defined in (20).

`public VarianceGammaProcess getVarianceGammaProcess()`

Returns a reference to the variance gamma process X defined in the constructor.

References

[1] H. Albrecher and M. Predota. On Asian option pricing for NIG Lévy processes. *Journal of computational and applied mathematics*, 172:153–168, 2004.

[2] T. Avramidis and P. L'Ecuyer. Efficient Monte Carlo and quasi-Monte Carlo option pricing under the variance-gamma model. *Management Science*, 52(12):1930–1944, 2006.

[3] T. Avramidis, P. L'Ecuyer, and P.-A. Tremblay. Efficient simulation of gamma and variance-gamma processes. In *Proceedings of the 2003 Winter Simulation Conference*, pages 319–326, Piscataway, New Jersey, 2003. IEEE Press.

[4] O. E. Barndorff-Nielsen. Processes of normal inverse gaussian type. *Finance and Stochastics*, 2:41–68, 1998.

[5] J. C. Cox, J. E. Ingersoll, and S. A. Ross. A theory of the term structure of interest rates. *Econometrica*, 53:385–407, 1985.

[6] W. Feller. *An Introduction to Probability Theory and Its Applications, Vol. 2.* Wiley, New York, NY, first edition, 1966.

[7] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* Springer-Verlag, New York, 2004.

[8] J. Imai and K. S. Tan. A general dimension reduction technique for derivative pricing. *Journal of Computational Finance*, 10(2):129–155, 2006.

[9] P. L'Ecuyer. Quasi-Monte Carlo methods in finance. In R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A Peters, editors, *Proceedings of the 2004 Winter Simulation Conference*, Piscataway, New Jersey, 2004. IEEE Press.

[10] P. L'Ecuyer, J. S. Parent-Chartier, and M. Dion. Simulation of a Lévy process by PCA sampling to reduce the effective dimention. In *Proceedings of the 2008 Winter Simulation Conference*, pages 436–443, Piscataway, NJ, 2008. IEEE Press.

[11] P. L'Ecuyer and R. Simard. Inverting the symmetrical beta distribution. *ACM Transactions on Mathematical Software*, 32(4):509–520, 2006.

[12] D. B. Madan, P. P. Carr, and E. C. Chang. The variance gamma process and option pricing. *European Finance Review*, 2:79–105, 1998.

[13] D. B. Madan and F. Milne. Option pricing with V.G. martingale components. *Mathematical Finance*, 1:39–55, 1991.

[14] D. B. Madan and E. Seneta. The variance gamma (V.G.) model for share market returns. *Journal of Business*, 63:511–524, 1990.

[15] 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.

[16] C. Ribeiro and N. Webber. Valuing path-dependent options in the variance-gamma model by Monte Carlo with a gamma bridge. manuscript, December 2002.

[17] 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.