SSJ User's Guide

Package `hups`

Tools for Quasi-Monte Carlo

Version: February 26, 2009

This package provides classes implementing highly uniform point sets (HUPS) and tools for their randomization. These point sets can be used for quasi-Monte Carlo integration. Randomized quasi-Monte Carlo point sets can in fact replace streams of uniform random numbers in a simulation, for the purpose of reducing the variance of the estimator.

Contents

Overview

Monte Carlo and quasi-Monte Carlo

This package provides classes implementing *highly uniform point sets* (HUPS) over the s-dimensional unit hypercube $[0,1)^s$, and tools for their randomization. The terminology *low-discrepancy sequence* (LDS) is often used for infinite sequences of points such that the *discrepancy* between the distribution of the first n points of the sequence and the uniform distribution converges to zero at a certain rate when $n \to \infty$ [26]. HUPS and LDS are used for quasi-Monte Carlo integration, as we now briefly explain. See, e.g., [8, 9, 12, 21, 18, 28, 26, 34, 38] for further details.

Suppose we want to estimate the integral of a function f defined over the s-dimensional unit hypercube,

$$\mu = \int_{[0,1)^s} f(\boldsymbol{u}) d\boldsymbol{u}. \tag{1}$$

Practically any mathematical expectation that can be estimated by simulation can be written in this way, usually for a very complicated f and sometimes for $s = \infty$. Indeed, the source of randomness of stochastic simulations is usually a *stream* of real numbers $\boldsymbol{u} = (u_0, u_1, u_2, \ldots)$ whose purpose is to imitate i.i.d. $U(0,1)$ random variables. These real numbers are transformed in complicated ways to produce the estimator. Thus, the dimension s of the integral (1) represents the number of calls to the uniform random number generator if that number is deterministic. If it is random and unbounded, we take $s = \infty$. In the latter case, however, we can assume that the *actual* number of calls is finite with probability one (otherwise the simulation may never end).

We consider an estimator of μ of the form

$$Q_n = \frac{1}{n} \sum_{i=0}^{n-1} f(\boldsymbol{u}_i), \tag{2}$$

which is the average of f over the *point set* $P_n = \{\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{n-1}\} \subset [0,1)^s$.

With the *Monte Carlo* (MC) method, the $\boldsymbol{u}_i$'s are i.i.d. random vectors uniformly distributed over $[0,1)^s$. Then, Q_n is an unbiased estimator of μ with variance σ^2/n, where

$$\sigma^2 = \int_{[0,1)^s} f^2(\boldsymbol{u}) d\boldsymbol{u} - \mu^2, \tag{3}$$

and it obeys a central-limit theorem if $\sigma^2 < \infty$.

Quasi-Monte Carlo (QMC) methods use point sets P_n that are *more evenly distributed* over the unit hypercube than typical random points. We call them *highly uniform point sets* (HUPS). The aim is to reduce the size of the integration error $Q_n - \mu$. Two important classes of methods for constructing such point sets are *digital nets* and *integration lattices* [26, 34, 21, 9]. Both are implemented in this package, in various flavors.

Elementary constructions

To give an idea of how HUPS and LDS can be constructed, we start with a simple one-dimensional example. If $s = 1$ and n is fixed, very simple highly uniform constructions are the point sets $P_n = \{0, 1/n, \ldots, (n-1)/n\}$ and the shifted version $P'_n = \{1/(2n), 3/(2n), \ldots, (2n-1)/(2n)\}$.

In $s > 1$ dimensions, the simplest extensions would be as follows. Let $n = d^s$ for some integer d and define P_n as the Cartesian product of s copies of the one-dimensional sets P_d; that is, $P_n = \{(u_0, \ldots, u_{s-1}) : u_j \in \{0, 1/d, \ldots, (d-1)/d\}$ for each $j\}$, and similarly for P'_n. The point sets thus obtained are regular rectangular grids. Unfortunately, this approach breaks down rapidly when s gets large, because n must increase exponentially fast with s for fixed d. Another important drawback is that when P_n is projected over lower-dimensional subspaces, several points are projected onto each other and become redundant [21].

A better idea is to construct a point set P_n in s dimensions such that each one-dimensional projection of P_n is the set of values $\{0, 1/n, \ldots, (n-1)/n\}$. Of course, these values should not be visited in the same order for all coordinates, because otherwise all the points would lie on the diagonal line going from $(0, \ldots, 0)$ to $(1, \ldots, 1)$. In other words, for each coordinate j, $0 \le j < s$, we must define a different *permutation* of the integers $\{0, \ldots, n-1\}$ and visit the values $\{0, 1/n, \ldots, (n-1)/n\}$ in the order determined by that permutation. The trick is to select those permutations in a way that P_n itself is highly uniform over $[0,1)^s$ in a well-defined sense (to be determined). This is what most construction methods attempt to achieve. Before looking at concrete ways of defining such permutations, we introduce a related issue: what to do if n is not fixed.

For $s = 1$, a simple way of filling up the unit interval $[0,1)$ uniformly is via the low-discrepancy sequence 0, 1/2, 1/4, 3/4, 1/8, 5/8, 3/8, 7/8, 1/16, 9/16, ..., called the *van der Corput sequence* in base 2. More generally, select an integer $b \ge 2$, called the *base*. The *radical inverse* function in base b, $\psi_b : \mathbb{N} \to [0,1)$, is defined as follows. If i is a k-digit integer in base b with digital b-ary expansion

$$i = a_0 + a_1 b + \ldots + a_{k-1} b^{k-1},$$

then

$$\psi_b(i) = a_0 b^{-1} + a_1 b^{-2} + \cdots + a_{k-1} b^{-k}.$$

For a given b, $\psi_b(0), \psi_b(1), \psi_b(2), \ldots$ is called the *van der Corput sequence in base b*. This sequence fills up the unit interval $[0,1)$ quite uniformly. For example, for $b = 2$ we obtain the sequence mentioned above and for $b = 3$ we obtain 0, 1/3, 2/3, 1/9, 4/9, 7/9, 2/9, 5/9, 8/9, 1/27, 10/27, 19/27, ... Moreover, for two relatively prime bases b_1 and b_2, the two sequences have no value in common except 0.

For $s > 1$, one could either take different (relatively prime) bases for the different coordinates, or take the same basis b but permute the successive values using a different permutation for each coordinate. These permutations are usually selected in a way that for every integer k, the first b^k values that are enumerated remain the same (they are the values of $\psi_b(i)$ for $i = 0, \ldots, b^k - 1$), but they are enumerated in a different order. Several digital net constructions (to be defined later) fit this framework.

If we decide to take different bases, the most natural choice is to take the jth smallest prime, b_j, as a base for coordinate $j-1$; that is, base 2 for coordinate 0, base 3 for coordinate 1, base 5 for coordinate 2, and so on. The infinite sequence thus defined, where point i is

$$\boldsymbol{u}_i = (\psi_{b_1}(i), \psi_{b_2}(i), \ldots, \psi_{b_s}(i)) \tag{4}$$

for $i \geq 0$, was proposed in [10] and is called the *Halton sequence*. One drawback of this sequence is that for large s, the base b_s becomes quite large.

In the case where n is fixed, we can always take i/n as the first coordinate of point i. In particular, the *Hammersley point set* with n points in s dimensions contains the points

$$\boldsymbol{u}_i = (i/n, \psi_{b_1}(i), \psi_{b_2}(i), \ldots, \psi_{b_{s-1}}(i)), \tag{5}$$

for $i = 0, \ldots, n-1$ [11]. Historically, Halton sequences were defined as extensions of Hammersley point sets.

Digital nets

Digital nets and sequences are an important class of HUPS and LDS constructions. Most concrete implementations, e.g., those proposed by Sobol', Faure, Niederreiter, and Niederreiter and Xing, are *linear* digital nets and sequences, defined as follows (see also [26, 38, 21]).

Let $b \geq 2$ be an arbitrary integer (usually a prime number), called the *base*. A net that contains $n = b^k$ points in s dimensions is defined via s *generator matrices* $\mathbf{C}_0, \ldots, \mathbf{C}_{s-1}$, which are (in theory) $\infty \times k$ matrices whose elements are in $\mathbb{Z}_b = \{0, \ldots, b-1\}$. The matrix $\mathbf{C}_j$ is used for coordinate j of all the points, for $j \geq 0$. To define the ith point $\boldsymbol{u}_i$, for $i = 0, \ldots, b^k - 1$, write the digital expansion of i in base b and multiply the vector of its digits by $\mathbf{C}_j$ to obtain the digits of the expansion of $u_{i,j}$, the jth coordinate of $\boldsymbol{u}_i$. That is,

$$i = \sum_{\ell=0}^{k-1} a_{i,\ell} b^{\ell}, \tag{6}$$

$$\begin{pmatrix} u_{i,j,1} \\ u_{i,j,2} \\ \vdots \end{pmatrix} = \mathbf{C}_j \begin{pmatrix} a_{i,0} \\ a_{i,1} \\ \vdots \\ a_{i,k-1} \end{pmatrix}, \tag{7}$$

$$u_{i,j} = \sum_{\ell=1}^{\infty} u_{i,j,\ell} b^{-\ell}, \tag{8}$$

$$\boldsymbol{u}_i = (u_{i,0}, \ldots, u_{i,s-1}). \tag{9}$$

In practice, the expansion in (8) is truncated to the first r digits for some positive integer r, so each matrix $\mathbf{C}_j$ is actually truncated to a $r \times k$ matrix. Typically r is equal to k, or is slightly larger, or is selected so that b^r is near 2^{31}.

Usually, the first k lines of each $\mathbf{C}_j$ form a nonsingular $k \times k$ matrix. Then, the n output values for coordinate j, $u_{0,j}, \ldots, u_{n-1,j}$, when truncated to their first k fractional digits in

base b, are a permutation of the numbers $0, 1/n, \ldots, (n-1)/n$. Different coordinates simply use different permutations, implemented via the matrices $\mathbf{C}_j$.

When the first k lines of $\mathbf{C}_j$ form the identity and the other lines are zero, the first n output values are the first n elements of the van der Corput sequence in base b. If we reverse the order of the columns of that matrix $\mathbf{C}_j$ (i.e., column c will contain a one in line $k-c+1$ and zeros elsewhere, for $0 \le c < k$), we obtain the output values $0, 1/n, \ldots, (n-1)/n$ in that order. With a slight abuse of language, we shall call this first matrix (with the identity followed by lines of zeros) the *identity* and the second one (with the columns in reverse order) the *reflected identity*. It is customary to take $\mathbf{C}_0$ as the identity for digital sequences, and often for digital nets as well. But for digital nets (where n is fixed in advance), one can take $\mathbf{C}_0$ as the reflected identity instead, then $\mathbf{C}_1$ as the identity, and so on. That is, the matrix $\mathbf{C}_j$ for the digital net is taken as the matrix $\mathbf{C}_{j-1}$ of the digital sequence. Our package often gives the choice.

For digital sequences, the matrices $\mathbf{C}_j$ actually have an infinite number of columns, although only the first k columns are needed to generate the first b^k points. So in practice, we never need to store more than a finite number of columns at a time. Whenever we find that we need more than b^k points for the current value of k, we can simply increase k and add the corresponding columns to the matrices $\mathbf{C}_j$.

The classes `DigitalNet` and `DigitalSequence` implement generic digital nets and sequences. Specific instances are constructed in subclasses of these two classes.

Lattice Rules

An *integration lattice* is a discrete (but infinite) subset of $\mathbb{R}^s$ of the form

$$L_s = \left\{ \mathbf{v} = \sum_{j=1}^{s} h_j \mathbf{v}_j \text{ such that each } h_j \in \mathbb{Z} \right\},$$

where $\mathbf{v}_1, \ldots, \mathbf{v}_s \in \mathbb{R}^s$ are linearly independent over $\mathbb{R}$ and $\mathbb{Z}^s \subseteq L_s$. This last condition means that L_s must contain all integer vectors, and this implies that L_s is periodic with period 1 along each of the s coordinates. The approximation of μ by Q_n with the point set $P_n = L_s \cap [0,1)^s$ is called a *lattice rule* [16, 34]. The value of n is the number of points of the lattice that are in the unit hypercube $[0,1)^s$.

Let $\mathbf{V}$ be the matrix whose rows are the basis vectors $\mathbf{v}_1, \cdots, \mathbf{v}_s$ and $\mathbf{V}^{-1}$ its inverse. One has $\mathbb{Z}^s \subseteq L_s$ if and only if all entries of $\mathbf{V}^{-1}$ are integer. When this holds, $n = \det(\mathbf{V}^{-1})$ and all entries of $\mathbf{V}$ are multiples of $1/n$.

The *rank* of the lattice is the smallest r such that one can find a basis of the form $\mathbf{v}_1, \ldots, \mathbf{v}_r, \mathbf{e}_{r+1}, \cdots, \mathbf{e}_s$, where $\mathbf{e}_j$ is the jth unit vector in s dimensions. In particular, a lattice rule of *rank 1* has a basis of the form $\mathbf{v}_1 = (a_1, \ldots, a_s)/n$ and $\mathbf{v}_j = \mathbf{e}_j$ for $j > 1$, where $a_j \in \mathbb{Z}_n$ for each j. It is a *Korobov* rule if $\mathbf{v}_1$ has the special form $\mathbf{v}_1 = (1,\ a,\ a^2 \bmod n,\ \ldots,\ a^{s-1} \bmod n)/n$ for some $a \in \mathbb{Z}_n$. The point set P_n of a Korobov lattice rule can also be written as $P_n = \{(x_1, \ldots, x_s)/n$ such that $x_1 \in \mathbb{Z}_n$ and

$x_j = ax_{j-1} \bmod n$ for all $j > 1\}$. This is the set of all vectors of successive values produced by a linear congruential generator (LCG) with modulus n and multiplier a, from all possible initial states, including 0. In this case, the points are easy to enumerate by using the recurrence.

Cycle-based point sets

Certain types of point sets are defined pretty much like random number generators: choose a finite state space $\mathcal{S}$, a transition function $f : \mathcal{S} \to \mathcal{S}$, an output function $g : \mathcal{S} \to [0, 1)$, and define

$$P_n = \{\boldsymbol{u} = (u_0, u_1, \ldots) : s_0 \in \mathcal{S},\ s_j = f(s_{j-1}),\ \text{and } u_j = g(s_j) \text{ for all } j\}. \tag{10}$$

This is the set of all vectors of successive output values produced by the recurrence defined by f and the output function g, from all possible initial states. The value of n is the cardinality of $\mathcal{S}$ and the dimension s is infinite. We could also have $n = \infty$ (an infinite sequence) if $\mathcal{S}$ is infinite but denumerable and ordered (so we know in which order to enumerate the points).

Let us assume that n is finite and that for each $s_0 \in \mathcal{S}$, the recurrence $s_j = f(s_{j-1})$ is *purely periodic*, i.e., there is always an integer j such that $s_j = s_0$. The smallest such j, called the *period length*, depends in general on s_0. Thus, the state space $\mathcal{S}$ is partitioned into a finite number of *cycles*. The successive coordinates of any point $\boldsymbol{u} \in P_n$ are periodic with period length equal to the length of the cycle that contains s_0 (and the following s_j's).

One way of implementing such a point set while avoiding to recompute f and g each time a coordinate is needed is to store explicitly all the cycles of the recurrence, in the form of a *list of cycles*. We can store either the successive u_j's directly, or the successive s_j's, over each cycle. The class `CycleBasedPointSet` provides the framework for doing that.

For example, a Korobov lattice point set is defined via the recurrence $x_j = ax_{j-1} \bmod n$ and output function $u_j = x_j/n$. If n is prime and a is a primitive element modulo n, then there are two cycles: one of period 1 that contains only 0, and the other of period $n - 1$. For more general n and a, there will be more cycles. The class `LCGPointSet` constructs this type of point set and stores explicitly the successive values of u_j over the different cycles.

There are cases where n is a power of two, say $n = 2^k$, and where the state s_j is represented as a k-bit string. In that context, it is often more convenient to store the successive s_j's instead of the successive u_j's, over the set of cycles (e.g., if a random digital shift in base 2 is to be applied to randomize the points, it can be performed by applying a bitwise xor directly to s_j). When generating the coordinates, the s_j's can be interpreted as 2^k-bit integers and multiplied by 2^{-k} to produce the output. This is supported by the class `CycleBasedPointSetBase2`. Special instances of this class are usually based on linear recurrences modulo 2 and they include the Korobov-type *polynomial lattice rules* [17, 19, 22, 24, 31].

Randomized quasi-Monte Carlo

In their original versions, these HUPS are deterministic, and the corresponding QMC methods give a *deterministic* integration error that is difficult to estimate. In *randomized* QMC methods, P_n is randomized, preferably in a way that it retains its high uniformity over $[0,1)^s$ when taken as a set, while each of its points has the uniform distribution over $[0,1)^s$ when taken individually. Then, Q_n becomes an unbiased estimator of μ, hopefully with smaller variance than the standard MC estimator. To estimate the variance and compute a confidence interval on μ, one can apply m independent randomizations to the same P_n, and compute $\bar{X}_m$ and $S^2_{m,x}$, the sample mean and sample variance of the m corresponding (independent) copies of Q_n. Then, $E[\bar{X}_m] = \mu$ and $E[S^2_{m,x}] = \mathrm{Var}[Q_n] = m\mathrm{Var}[\bar{X}_m]$ [20].

Two examples of such randomizations are the *random shift modulo 1*, proposed in [4] and implemented in class `RandShiftedPointSet`, and the *random digital shift in base b*, described and implemented in class `DigitalNet`. These randomizations are also incorporated directly in certain types of point sets such as `CycleBasedPointSet`, `CycleBasedPointSetBase2`, etc.

In the random shift modulo 1, we generate a *single* point $\boldsymbol{u}$ uniformly over $[0,1)^s$ and add it to each point of P_n, coordinate-wise, modulo 1. Since all points of P_n are shifted by the same amount, the set retains most of its structure and uniformity.

For the random digital shift in base b, we generate again a single $\boldsymbol{u} = (u_1, \ldots, u_s)$ uniformly over $[0,1)^s$, write the digital expansion in base b of each of its coordinates, say $u_j = \sum_{\ell=1}^{\infty} d_{j,\ell} b^{-\ell}$, then add $d_{j,\ell}$ modulo b to the ℓth digit of the digital expansion in base b of the jth coordinate of each point $\boldsymbol{u}_i \in P_n$. For $b = 2$, the digit-wise addition modulo b becomes a bitwise exclusive-or, which is fast to perform on a computer.

An interesting property of the digital shift in base b is that if the hypercube $[0,1)^s$ is partitioned into $b^{q_1+\cdots+q_s}$ rectangular boxes of the same size by partitioning the jth axis into b^{q_j} equal parts for each j, for some integers $q_j \geq 0$ (such a partition is called a $\mathbf{q}$-*equidissection in base b* of the unit hypercube, where $\mathbf{q} = (q_1, \ldots, q_s)$), then the number of boxes that contain m points, for each integer m, is unchanged by the randomization. In particular, if each box contains the same number of points of P_n before the randomization, then it also does after the randomization. In this case, we say that P_n is $\mathbf{q}$-*equidistributed in base b*. Several other randomization methods exist and most are adapted to special types of point sets; see, e.g., `DigitalNet` and [29].

Point set implementations and enumeration tools

Let $\boldsymbol{u}_i = (u_{i,0}, u_{i,1}, \ldots, u_{i,s-1})$ be the elements of the point set P_n, for $i = 0, \ldots, n-1$. Both the number of points n and the dimension s can be finite or infinite. The point set can be viewed as a two-dimensional array whose element (i,j) contains $u_{i,j}$, the coordinate j of point i. In the implementations of typical point sets, the values $u_{i,j}$ are not stored explicitly in a two-dimensional array, but pertinent information is organized so that the points and their coordinates can be generated efficiently. The base class for point sets is the abstract class `PointSet`.

To enumerate the successive points or the successive coordinates of a given point, we use *point set iterators*, which resemble the iterators defined in Java *collections*, except that they loop over bi-dimensional sets. Their general behavior is defined in the interface `PointSetIterator`. Several independent iterators can coexist at any given time for the same point set. Each one maintains a current point index and a current coordinate index, which are incremented by one when the iterator advances to the next point or the next coordinate. Both are initialized to 0. Each subclass of `PointSet` has its own implementation of `PointSetIterator` and has a method `iterator` that creates and returns a new point set iterator of the correct type.

An important feature of the `PointSetIterator` interface is that it extends the `RandomStream` interface. This means that any point set iterator can be used in place of a random stream that is supposed to generate i.i.d. $U(0,1)$ random variables, anywhere in a simulation program. It then becomes very easy to replace the (pseudo)random numbers by the coordinates $u_{i,j}$ of a randomized HUPS without changing the internal code of the simulation program.

Transformed point sets and containers

HUPS are often transformed either deterministically or randomly. Deterministic transformations can be applied to improve the uniformity, or to eliminate some points or coordinates (i.e., selecting subsets), or to concatenate point sets (padding), or to take an antithetic version of a point set, etc. Random transformations are used for randomized QMC. They are also useful when the *average* of a uniformity measure of interest over the outcomes of a certain type of randomization is much better than the worst case and may be better than the uniformity measure of the original point set. When a point set is transformed, we often want to keep the original as well, and we may want to apply different types of transformations or different independent randomizations to the same point set.

This can be achieved via *container* point sets, which are defined in terms of another point set to which they keep a reference and apply certain transformations. `ContainerPointSet` is the base class for such containers. One example is `RandShiftedPointSet`, which applies a random shift modulo 1 to the point set that it contains. Of course, the contained point set can be a container itself and this can be done recursively, but too many levels of recursiveness may impair the performance (speed).

Examples

To be done...

PointSet

This abstract class defines the basic methods for accessing and manipulating point sets. A point set can be represented as a two-dimensional array, whose element (i, j) contains $u_{i,j}$, the *coordinate* j of point i. Each coordinate $u_{i,j}$ is assumed to be in the unit interval $[0, 1]$. Whether the values 0 and 1 can occur may depend on the actual implementation of the point set.

All points have the same number of coordinates (their *dimension*) and this number can be queried by `getDimension`. The number of points is queried by `getNumPoints`. The points and coordinates are both numbered starting from 0 and their number can actually be infinite.

The `iterator` method provides a *point set iterator* which permits one to enumerate the points and their coordinates. Several iterators over the same point set can coexist at any given time. These iterators are instances of a hidden inner-class that implements the `PointSetIterator` interface. The default implementation of iterator provided here relies on the method `getCoordinate` to access the coordinates directly. However, this approach is rarely efficient. Specialized implementations that dramatically improve the performance are provided in subclasses of `PointSet`. The `PointSetIterator` interface actually extends the `RandomStream` interface, so that the iterator can also be seen as a `RandomStream` and used wherever such a stream is required for generating uniform random numbers. This permits one to easily replace pseudorandom numbers by the coordinates of a selected set of highly-uniform points, i.e., to replace Monte Carlo by quasi-Monte Carlo in a simulation program.

This abstract class has only one abstract method: `getCoordinate`. Providing an implementation for this method is already sufficient for the subclass to work. However, in practically all cases, efficiency can be dramatically improved by overwriting `iterator` to provide a custom iterator that does not necessarily rely on `getCoordinate`. In fact, direct use of `getCoordinate` to access the coordinates is discouraged. One should access the coordinates only via the iterators.

```
package umontreal.iro.lecuyer.hups;

public abstract class PointSet
```

Methods

`public int getDimension()`

Returns the dimension (number of available coordinates) of the point set. If the dimension is actually infinite, `Integer.MAX_VALUE` is returned.

`public int getNumPoints()`

Returns the number of points. If this number is actually infinite, `Integer.MAX_VALUE` is returned.

`public abstract double getCoordinate (int i, int j);`

Returns $u_{i,j}$, the coordinate j of the point i.

`public PointSetIterator iterator()`

Constructs and returns a point set iterator. The default implementation returns an iterator that uses the method `getCoordinate (i,j)` to iterate over the points and coordinates, but subclasses can reimplement it for better efficiency.

`public void setStream (RandomStream stream)`

Sets the random stream used to generate random shifts to `stream`.

`public RandomStream getStream()`

Returns the random stream used to generate random shifts.

`public void randomize (PointSetRandomization rand)`

Randomizes the point set using the given `rand`.

`public void addRandomShift (int d1, int d2, RandomStream stream)`

This method does nothing for this generic class. In some subclasses, it adds a random shift to all the points of the point set, using stream `stream` to generate the random numbers, for coordinates `d1` to `d2-1`.

`public void addRandomShift (RandomStream stream)`

Similar to `addRandomShift (0, d2, stream)`, with `d2` the dimension of the current random shift. This method does nothing for this generic class.

`public void clearRandomShift()`

Erases the current random shift, if any.

`public void randomize (int d1, int d2, RandomStream stream)`

By default, this method simply calls `addRandomShift(d1, d2, stream)`.

`public void randomize (RandomStream stream)`

By default, this method simply calls `addRandomShift(stream)`.

`public void unrandomize()`

By default, this method simply calls `clearRandomShift()`.

`public String toString()`

Formats a string that contains the information about the point set.

`public String formatPoints()`

Same as invoking `formatPoints` with n and d equal to the number of points and the dimension, respectively.

`public String formatPoints (int n, int d)`

Formats a string that displays the same information as returned by `toString`, together with the first d coordinates of the first n points. If the number of points is less than n or the dimension is less than d, the default implementation throws an exception. Of course, this method should be used only for small values of n and d.

PointSetIterator

Objects of classes that implement this interface are *iterators* that permit one to enumerate (or observe) the successive points of a point set and the successive coordinates of these points. Each `PointSetIterator` is associated with a given point set and maintains a *current point* index i and a *current coordinate* index j, which are both initialized to zero.

Successive coordinates can be accessed one or many at a time by the methods `nextCoordinate` and `nextCoordinates`, respectively. The current coordinate index j can be set explicitely by `setCurCoordIndex` and `resetCurCoordIndex`. Similar methods are available for resetting and accessing the current point. The method `nextPoint` permits one to enumerate the successive points in natural order.

This class also implements the `RandomStream` interface. This permits one to replace random numbers by the coordinates of (randomized) quasi-Monte Carlo points without changing the code that calls the generators in a simulation program. That is, the same simulation program can be used for both Monte Carlo and quasi-Monte Carlo simulations. The method `nextDouble` does exactly the same as `nextCoordinate`, it returns the current coordinate of the current point and advances the current coordinate by one. The substreams correspond to the points, so `resetStartSubstream` resets the current point coordinate to zero, `resetNextSubstream` resets the iterator to the next point, and `resetStartStream` resets the iterator to the first point of the point set.

There can be several iterators over the same point set. These iterators are independent from each other. Classes that implement this interface must maintain enough information so that each iterator is unaffected by other iterator's operations. However, the iterator does not need to be independent of the underlying point set. If the point set is modified (e.g., randomized), the iterator may continue to work as usual.

Point set iterators are implemented as inner classes because this gives a direct access to the private members (or variables) of the class. This is important for efficiency. They are quite similar to the iterators in Java *collections*.

```
package umontreal.iro.lecuyer.hups;

public interface PointSetIterator extends RandomStream

   public void setCurCoordIndex (int j);
      Sets the current coordinate index to j, so that the next calls to nextCoordinate or
      nextCoordinates will return the values u_{i,j}, u_{i,j+1}, ..., where i is the index of the current
      point.

   public void resetCurCoordIndex();
      Equivalent to setCurCoordIndex (0).

   public int getCurCoordIndex();
      Returns the index j of the current coordinate. This may be useful, e.g., for testing if enough
      coordinates are still available.
```

`public boolean hasNextCoordinate();`

Returns `true` if the current point has another coordinate. This can be useful for testing if coordinates are still available.

`public double nextCoordinate();`

Returns the current coordinate $u_{i,j}$ and advances to the next one. If no current coordinate is available (either because the current point index has reached the number of points or because the current coordinate index has reached the number of dimensions), it throws a `NoSuchElementException`.

`public void nextCoordinates (double[] p, int d);`

Returns the next `d` coordinates of the current point in `p` and advances the current coordinate index by `d`. If the remaining number of coordinates is too small, a `NoSuchElementException` is thrown, as in `nextCoordinate`.

`public void setCurPointIndex (int i);`

Resets the current point index to i and the current coordinate index to zero. If `i` is larger or equal to the number of points, an exception will *not* be raised here, but only later if we ask for a new coordinate or point.

`public void resetCurPointIndex();`

Equivalent to `setCurPointIndex (0)`.

`public int resetToNextPoint();`

Increases the current point index by 1 and returns its new value. If there is no more point, an exception will be raised only if we ask for a new coordinate or point later on.

`public int getCurPointIndex();`

Returns the index i of the current point. This can be useful, e.g., for caching point sets.

`public boolean hasNextPoint();`

Returns `true` if there is a next point. This can be useful for testing if points are still available.

`public int nextPoint (double[] p, int d);`

Returns the *first* `d` coordinates of the *current* point in `p`, advances to the next point, and returns the index of the *new* current point. Even if the current coordinate index is 0, the point returned starts from coordinate 0. After obtaining the last point via this method, the current point index (returned by the method) is equal to the number of points, so it is no longer a valid point index. An exception will then be raised if we attempt to generate additional points or coordinates.

Specialized implementations of this method often allow for increased efficiency, e.g., for cycle-based point sets where the cycles (but not the points) are stored explicitly or for digital nets by allowing non-incremental point enumerations via Gray-code counters.

PointSetRandomization

This interface is used to randomize a `PointSet`. One can implement method `randomize` in any way. This method must use an internal `RandomStream`. This stream can be set in the constructor, but the methods `getStream` and `setStream` must be implemented.

The method `randomize` must be implemented using combinations of the randomization methods from the point set such as `addRandomShift`, `leftMatrixScramble`, `stripedMatrixScramble`, ...

If more than one `PointSetRandomization` is applied to the same point set, the randomizations will concatenate if they are of different types, but only the last of each type will remain.

```
package umontreal.iro.lecuyer.hups;

public interface PointSetRandomization
```

Methods

`public void randomize (PointSet p);`

This method must randomize `p`.

`public void setStream (RandomStream stream);`

Sets the internal `RandomStream` to `stream`.

`public RandomStream getStream();`

Returns the internal `RandomStream`.

EmptyRandomization

This class implements an empty `PointSetRandomization`. The method `randomize` does nothing. The internal stream is never used. This class can be used in methods where a randomization is needed but you don't want one.

```
package umontreal.iro.lecuyer.hups;

public class EmptyRandomization implements PointSetRandomization
```

Methods

`public void randomize (PointSet p)`

This method does nothing.

`public void setStream (RandomStream stream)`

Sets the internal `RandomStream` to `stream`.

`public RandomStream getStream()`

Returns the internal `RandomStream`.

RandomShift

This class implements a `PointSetRandomization`. The `RandomStream` is stored internally. The method `randomize` simply calls `addRandomShift(stream)`.

This class can be used as a base class to implement a specific randomization by overriding method `randomize`.

```
package umontreal.iro.lecuyer.hups;

public class RandomShift implements PointSetRandomization
```

Constructor

`public RandomShift (RandomStream stream)`

Sets the internal `RandomStream` to `stream`.

Methods

`public void randomize (PointSet p)`

This method calls `addRandomShift (stream)`.

`public void setStream (RandomStream stream)`

Sets the internal `RandomStream` to `stream`.

`public RandomStream getStream()`

Returns the internal `RandomStream`.

LMScrambleShift

This class implements a `PointSetRandomization` that performs a left matrix scrambling and adds a random digital shift. Point set must be a `DigitalNet` or an `IllegalArgumentException` is thrown.

```
package umontreal.iro.lecuyer.hups;

public class LMScrambleShift extends RandomShift
```

Constructor

`public LMScrambleShift (RandomStream stream)`

Sets internal variable `stream` to the given `stream`.

Methods

`public void randomize (PointSet p)`

This method calls `leftMatrixScramble`, then `addRandomShift`. If `p` is not a `DigitalNet`, an `IllegalArgumentException` is thrown.

SMScrambleShift

This class implements a `PointSetRandomization` that performs a striped matrix scrambling and adds a random digital shift. Point set must be a `DigitalNet` or an `IllegalArgumentException` is thrown.

```
package umontreal.iro.lecuyer.hups;

public class SMScrambleShift extends RandomShift
```

Constructor

`public SMScrambleShift (RandomStream stream)`

Sets internal variable `stream` to the given `stream`.

Methods

`public void randomize (PointSet p)`

This method calls `stripedMatrixScramble`, then `addRandomShift`. If `p` is not a `DigitalNet`, an `IllegalArgumentException` is thrown.

ContainerPointSet

This acts as a generic base class for all *container classes* that contain a point set and apply some kind of transformation to the coordinates to define a new point set. One example of such transformation is the *antithetic* map, applied by the container class `AntitheticPointSet`, where each output coordinate $u_{i,j}$ is transformed into $1-u_{i,j}$. Another example is `RandShiftedPointSet`.

The class implements a specialized type of iterator for container point sets. This type of iterator contains itself an iterator for the containee and uses it to access the points and coordinates internally, instead of maintaining itself indices for the current point and current coordinate.

```
package umontreal.iro.lecuyer.hups;

public abstract class ContainerPointSet extends PointSet
```

`protected void init (PointSet P0)`

Initializes the container point set which will contain point set `P0`. This method must be called by the constructor of any class inheriting from `ContainerPointSet`.

`public int getDimension()`

Returns the dimension of the contained point set.

`public int getNumPoints()`

Returns the number of points of the contained point set.

`public void randomize (PointSetRandomization rand)`

Randomizes the contained point set using `rand`.

`public void addRandomShift (int d1, int d2, RandomStream stream)`

Calls `addRandomShift(d1, d2, stream)` of the contained point set.

`public void addRandomShift (RandomStream stream)`

Calls `addRandomShift(stream)` of the contained point set.

`public void clearRandomShift()`

Calls `clearRandomShift()` of the contained point set.

CachedPointSet

This container class caches a point set by precomputing and storing its points locally in an array. This can be used to speed up computations when using a small low-dimensional point set more than once.

```
package umontreal.iro.lecuyer.hups;

public class CachedPointSet extends PointSet
```

Constructor

`public CachedPointSet (PointSet P, int n, int dim)`

Creates a new `PointSet` object that contains an array storing the first `dim` coordinates of the first `n` points of `P`. The original point set `P` itself is not modified.

`public CachedPointSet (PointSet P)`

Creates a new `PointSet` object that contains an array storing the points of `P`. The number of points and their dimension are the same as in the original point set. Both must be finite.

`public void randomize (PointSetRandomization rand)`

Randomizes the underlying point set using `rand` and recaches the points.

SubsetOfPointSet

This container class permits one to select a subset of a point set. This is done by selecting a range or providing an array of either point or coordinate indices. A typical application of a range selection is to make the number of points or the dimension finite. It is also possible to provide, for example, a random permutation in the selection of components.

Selecting a new subset of points or coordinates overwrites the previous selection. The specification of a subset with respect to the points is independent from selecting a subset with respect to the coordinates. The number of points and the dimension are adapted to the current selection and all indices still start from 0, i.e., the subset works just like an ordinary point set.

When the points or coordinates ranges are changed, existing iterators become invalid. They should be reconstructed or reset to avoid inconsistencies.

```
package umontreal.iro.lecuyer.hups;

public class SubsetOfPointSet extends PointSet
```

Constructor

`public SubsetOfPointSet (PointSet P)`

Constructs a new `PointSet` object, initially identical to `P`, and from which a subset of the points and/or a subset of the coordinates is to be extracted.

Methods

`public void selectPointsRange (int from, int to)`

Selects the points numbered from "`from`" to "`to - 1`" from the original point set.

`public void selectPoints (int[] pointIndices, int numPoints)`

Selects the `numPoints` points whose numbers are provided in the array `pointIndices`.

`public void selectCoordinatesRange (int from, int to)`

Selects the coordinates from "`from`" to "`to - 1`" from the original point set.

`public void selectCoordinates (int[] coordIndices, int numCoord)`

Selects the `numCoord` coordinates whose numbers are provided in the array `coordIndices`.

2

[2] From Pierre: Code à reviser.

PaddedPointSet

This container class realizes *padded point sets*, constructed by taking some coordinates from a point set P_1, other coordinates from a point set P_2, and so on. This can be used to implement *latin supercube sampling* [28], for example. After calling the constructor to create the structure, component point sets can be padded to it by calling `padPointSet` or `padPointSetPermute`.

Only sets with the same number of points can be padded. Point sets with too many points or coordinates can be trimmed down by using the class `SubsetOfPointSet` before they are padded. Infinite-dimensional point sets are allowed, but once one is padded, no additional point set can be padded.

The points of each padded set can be permuted randomly, independently across the padded sets. If such a random permutation is desired, the point set should be padded via `padPointSetPermute`. When calling `randomize`, random permutations are generated for all point sets that have been padded by `padPointSetPermute`.

```
package umontreal.iro.lecuyer.hups;

public class PaddedPointSet extends PointSet
```

Constructor

`public PaddedPointSet (int maxPointSets)`

Constructs a structure for padding at most `maxPointSets` point sets. This structure is initially empty and will eventually contain the different point sets that are padded.

Methods

`public void padPointSet (PointSet P)`

Pads the point set `P` to the present structure.

`public void padPointSetPermute (PointSet P)`

Pads the point set `P`, which is assumed to be *finite*. A random permutation will be generated (when calling `randomize`) and used to access the coordinates taken from the points of `P` (i.e., these points are randomly permuted).

AntitheticPointSet

This container class provides antithetic points. That is, $1 - u_{i,j}$ is returned in place of coordinate $u_{i,j}$. To generate regular and antithetic variates with a point set `p`, e.g., for variance reduction, one can define an `AntitheticPointSet` object `pa` that contains `p`, and then generate the regular variates with `p` and the antithetic variates with `pa`.

```
package umontreal.iro.lecuyer.hups;

public class AntitheticPointSet extends ContainerPointSet
```

Constructor

`public AntitheticPointSet (PointSet P)`

Constructs an antithetic point set from the given point set `P`.

RandShiftedPointSet

This container class embodies a point set to which a random shift modulo 1 is applied (i.e., a single uniform random point is added to all points, modulo 1, to randomize the inner point set). When calling `addRandomShift`, a new random shift will be generated. This shift is represented by a vector of d uniforms over $(0, 1)$, where d is the current dimension of the shift.

```
package umontreal.iro.lecuyer.hups;

public class RandShiftedPointSet extends ContainerPointSet
```

Constructor

`public RandShiftedPointSet (PointSet P, int dimShift, RandomStream stream)`

Constructs a structure to contain a randomly shifted version of `P`. The random shifts will be generated in up to `dimShift` dimensions, using stream `stream`.

Methods

`public int getShiftDimension()`

Returns the number of dimensions of the current random shift.

`public void addRandomShift (int d1, int d2, RandomStream stream)`

Changes the stream used for the random shifts to `stream`, then refreshes the shift for coordinates `d1` to `d2-1`.

`public void addRandomShift (RandomStream stream)`

Changes the stream used for the random shifts to `stream`, then refreshes all coordinates of the random shift, up to its current dimension.

`public void addRandomShift (int d1, int d2)`

Refreshes the random shift (generates new uniform values for the random shift coordinates) for coordinates `d1` to `d2-1`.

`public void addRandomShift()`

Refreshes all coordinates of the random shift, up to its current dimension.

BakerTransformedPointSet

This container class embodies a point set to which a *Baker transformation* is applied (see, e.g., [13]). It transforms each coordinate u into $2u$ if $u \leq 1/2$ and $2(1-u)$ if $u > 1/2$.

```
package umontreal.iro.lecuyer.hups;

public class BakerTransformedPointSet extends ContainerPointSet
```

Constructor

`public BakerTransformedPointSet (PointSet P)`

Constructs a Baker-transformed point set from the given point set `P`.

CycleBasedPointSet

This abstract class provides the basic structures for storing and manipulating a *highly uniform point set* defined by a set of cycles. The s-dimensional points are all the vectors of s successive values found in any of the cycles, from any starting point. Since this is defined for any positive integer s, the points effectively have an infinite number of dimensions. The number of points, n, is the sum of lengths of all the cycles. The cycles of the point set are simply stored as a list of arrays, where each array contains the successive values for a given cycle. By default, the values are stored in `double`.

This structure is convenient for implementing recurrence-based point sets, where the point set in s dimensions is defined as the set of all vectors of s successive values of a periodic recurrence, from all its possible initial states.

```
package umontreal.iro.lecuyer.hups;

public abstract class CycleBasedPointSet extends PointSet
```

`public void addRandomShift (int d1, int d2, RandomStream stream)`

Adds a random shift to all the points of the point set, using stream `stream` to generate the random numbers, for coordinates `d1` to `d2 - 1`. This applies an addition modulo 1 of a single random point to all the points.

`protected void addCycle (AbstractList c)`

Adds the cycle `c` to the list of all cycles. This method is used by subclass constructors to fill up the list of cycles.

LCGPointSet

Implements a recurrence-based point set defined via a linear congruential recurrence of the form $x_i = ax_{i-1} \bmod n$ and $u_i = x_i/n$. The implementation is done by storing the values of u_i over the set of all cycles of the recurrence.

```
package umontreal.iro.lecuyer.hups;

public class LCGPointSet extends CycleBasedPointSet
```

Constructors

`public LCGPointSet (int n, int a)`

Constructs and stores the set of cycles for an LCG with modulus n and multiplier a. If the LCG has full period length $n - 1$, there are two cycles, the first one containing only 0 and the second one of period length $n - 1$.

`public LCGPointSet (int b, int e, int c, int a)`

Constructs and stores the set of cycles for an LCG with modulus $n = b^e + c$ and multiplier a.

`public int geta ()`

Returns the value of the multiplier a.

CycleBasedPointSetBase2

Similar to `CycleBasedPointSet`, except that the successive values in the cycles are stored as integers in the range $\{0, \ldots, 2^k - 1\}$, where $1 \leq k \leq 31$. The output values $u_{i,j}$ are obtained by dividing these integer values by 2^k. Point sets where the successive coordinates of each point are obtained via linear recurrences modulo 2 (e.g., linear feedback shift registers or Korobov-type polynomial lattice rules) are naturally expressed in this form. Storing the integers $2^k u_{i,j}$ instead of the $u_{i,j}$ themselves makes it easier to apply randomizations such as digital random shifts in base 2, which are applied to the bits *before* transforming the value to a real number $u_{i,j}$. When a random digital shift is performed, it applies a bitwise exclusive-or of all the points with a single random point.

```
package umontreal.iro.lecuyer.hups;

public abstract class CycleBasedPointSetBase2 extends CycleBasedPointSet
```

`public void addRandomShift (int d1, int d2, RandomStream stream)`

Adds a random digital shift in base 2 to all the points of the point set, using stream `stream` to generate the random numbers, for coordinates `d1` to `d2 - 1`. This applies a bitwise exclusive-or of all the points with a single random point.

F2wStructure

This class implements methods and fields needed by the classes `F2wNetLFSR`, `F2wNetPolyLCG`, `F2wCycleBasedLFSR` and `F2wCycleBasedPolyLCG`. It also stores the parameters of these point sets which will contain 2^{rw} points (see the meaning of r and w below). The parameters can be stored as a polynomial $P(z)$ over $\mathbb{F}_{2^w}[z]$

$$P(z) = z^r + \sum_{i=1}^{r} b_i z^{r-i}$$

where $b_i \in \mathbb{F}_{2^w}$ for $i = 1, \ldots, r$. Let ζ be the root of an irreducible polynomial $Q(z) \in \mathbb{F}_2[z]$. It is well known that ζ is a generator of the finite field $\mathbb{F}_{2^w}$. The elements of $\mathbb{F}_{2^w}$ are represented using the polynomial ordered basis $(1, \zeta, \ldots, \zeta^{w-1})$.

In this class, only the non-zero coefficients of $P(z)$ are stored. It is stored as

$$P(z) = z^{\mathtt{r}} + \sum_{i=0}^{\mathtt{nbcoeff}} \mathtt{coeff}[i] z^{\mathtt{nocoeff}[i]}$$

where the coefficients in `coeff[]` represent the non-zero coefficients b_i of $P(z)$ using the polynomial basis. The finite field $\mathbb{F}_{2^w}$ used is defined by the polynomial

$$Q(z) = z^w + \sum_{i=1}^{w} a_i z^{w-i}$$

where $a_i \in \mathbb{F}_2$, for $i = 1, \ldots, w$. Polynomial Q is stored as the bit vector $\mathtt{modQ} = (a_w, \ldots, a_1)$.

The class also stores the parameter `step` that is used by the classes `F2wNetLFSR`, `F2wNetPolyLCG`, `F2wCycleBasedLFSR` and `F2wCycleBasedPolyLCG`. This parameter is such that the implementation of the recurrence will output a value at every `step` iterations.

```
package umontreal.iro.lecuyer.hups;

public class F2wStructure
```

Constructors

```
F2wStructure (int w, int r, int modQ, int step, int nbcoeff,
              int coeff[], int nocoeff[])
```

Constructs a `F2wStructure` object that contains the parameters of a polynomial in $\mathbb{F}_{2^w}[z]$, as well as a stepping parameter.

```
F2wStructure (String filename, int no)
```

Constructs a polynomial in $\mathbb{F}_{2^w}[z]$ after reading its parameters from file `filename`; the parameters of this polynomial are stored at line number `no` of `filename`. The files are kept

in different directories depending on the criteria used in the searches for the parameters defining the polynomials. The different criteria for the searches and the theory behind it are described in [32, 30]. The existing files and the number of polynomials they contain are given in the following tables. The first table below contains files in subdirectory `LFSR_equid_max`. The name of each file indicates the value of r and w for the polynomials. For example, file `f2wR2_W5.dat` in directory `LFSR_equid_max` contains the parameters of 2358 polynomials with $r = 2$ and $w = 5$. For example, to use the 5-*th* polynomial of file `f2wR2_W5.dat`, one may call `F2wStructure("f2wR2_W5.dat", 5)`. The files of parameters have been stored at the address `http://www.iro.umontreal.ca/~simardr/ssj-1/dataF2w/`.

Methods

`int getLog2N ()`

This method returns the product rw.

`int multiply (int a, int b)`

Method that multiplies two elements in $\mathbb{F}_{2^w}$.

`public static void print (String filename)`

Prints the content of file `filename`. See the constructor above for the conditions on `filename`.

`public String toString ()`

This method returns a string containing the polynomial $P(z)$ and the stepping parameter.

Directory `LFSR_equid_max`

Filename	Num of poly.
f2wR2_W5.dat	2358
f2wR2_W6.dat	1618
f2wR2_W7.dat	507
f2wR2_W8.dat	26
f2wR2_W9.dat	3
f2wR3_W4.dat	369
f2wR3_W5.dat	26
f2wR3_W6.dat	1
f2wR4_W3.dat	117
f2wR4_W4.dat	1
f2wR5_W2.dat	165
f2wR5_W3.dat	1
f2wR6_W2.dat	36
f2wR6_W3.dat	1
f2wR7_W2.dat	10
f2wR8_W2.dat	11
f2wR9_W2.dat	1

Directory `LFSR_equid_sum`

Filename	Num of poly.
f2wR2_W5.dat	2276
f2wR2_W6.dat	1121
f2wR2_W7.dat	474
f2wR2_W8.dat	37
f2wR2_W9.dat	6
f2wR3_W4.dat	381
f2wR3_W5.dat	65
f2wR3_W6.dat	7
f2wR4_W3.dat	154
f2wR4_W4.dat	2
f2wR5_W2.dat	688
f2wR5_W3.dat	5
f2wR6_W2.dat	70
f2wR6_W3.dat	1
f2wR7_W2.dat	9
f2wR8_W2.dat	3
f2wR9_W2.dat	3

Directory `LFSR_mindist_max`

Filename	Num of poly.
f2wR2_W5.dat	1
f2wR2_W6.dat	1
f2wR2_W7.dat	2
f2wR2_W8.dat	2
f2wR2_W9.dat	1
f2wR3_W4.dat	2
f2wR3_W5.dat	2
f2wR3_W6.dat	1
f2wR4_W3.dat	1
f2wR4_W4.dat	1
f2wR5_W2.dat	2
f2wR5_W3.dat	1
f2wR6_W2.dat	4
f2wR6_W3.dat	1
f2wR7_W2.dat	1
f2wR8_W2.dat	1
f2wR9_W2.dat	1

Directory `LFSR_mindist_sum`

Filename	Num of poly.
f2wR2_W5.dat	1
f2wR2_W6.dat	1
f2wR2_W7.dat	1
f2wR2_W8.dat	1
f2wR2_W9.dat	1
f2wR3_W4.dat	1
f2wR3_W5.dat	1
f2wR3_W6.dat	1
f2wR4_W3.dat	1
f2wR4_W4.dat	2
f2wR5_W2.dat	2
f2wR5_W3.dat	2
f2wR6_W2.dat	1
f2wR6_W3.dat	1
f2wR7_W2.dat	2
f2wR8_W2.dat	1
f2wR9_W2.dat	2

Directory `LFSR_tvalue_max`

Filename	Num of poly.
f2wR2_W5.dat	7
f2wR2_W6.dat	1
f2wR2_W7.dat	1
f2wR2_W8.dat	1
f2wR2_W9.dat	1
f2wR3_W4.dat	1
f2wR3_W5.dat	1
f2wR3_W6.dat	1
f2wR4_W3.dat	2
f2wR4_W4.dat	1
f2wR5_W2.dat	14
f2wR5_W3.dat	1
f2wR6_W2.dat	2
f2wR6_W3.dat	1
f2wR7_W2.dat	1
f2wR8_W2.dat	1
f2wR9_W2.dat	1

Directory `LFSR_tvalue_sum`

Filename	Num of poly.
f2wR2_W5.dat	15
f2wR2_W6.dat	1
f2wR2_W7.dat	1
f2wR2_W8.dat	2
f2wR2_W9.dat	1
f2wR3_W4.dat	1
f2wR3_W5.dat	1
f2wR3_W6.dat	1
f2wR4_W3.dat	2
f2wR4_W4.dat	1
f2wR5_W2.dat	13
f2wR5_W3.dat	2
f2wR6_W2.dat	12
f2wR6_W3.dat	1
f2wR7_W2.dat	1
f2wR8_W2.dat	1
f2wR9_W2.dat	1

F2wCycleBasedLFSR

This class creates a point set based upon a linear feedback shift register sequence. The recurrence used to produce the point set is

$$m_n = \sum_{i=1}^{r} b_i m_{n-i}$$

where $m_n \in \mathbb{F}_{2^w}$, $n \geq 0$ and $b_i \in \mathbb{F}_{2^w}$. There is a polynomial in $\mathbb{F}_{2^w}[z]$ associated with this recurrence called the *characteristic polynomial*. It is

$$P(z) = z^r + \sum_{i=1}^{r} b_i z^{r-i}.$$

In the implementation, this polynomial is stored in an object `F2wStructure`.

Let $\mathbf{x} = (x^{(0)}, \ldots, x^{(p-1)}) \in \mathbb{F}_2^p$ be a p-bit vector. Let us define the function $\phi(\mathbf{x}) = \sum_{i=1}^{p} 2^{-i} x^{(i-1)}$. The point set in t dimensions produced by this class is

$$\left\{(\phi(\mathbf{y}_0), \phi(\mathbf{y}_s), \ldots, \phi(\mathbf{y}_{s(t-1)}) : (\mathbf{v}_0, \ldots, \mathbf{v}_{r-1}) \in \mathbb{F}_2^{rw}\right\}$$

where $\mathbf{y}_n = \mathrm{trunc}_h(\mathbf{v}_n, \mathbf{v}_{n+1}, \ldots)$, $\mathbf{v}_n$ is the representation of m_n under the polynomial basis of $\mathbb{F}_{2^w}$ over $\mathbb{F}_2$, and $h = w\lfloor 31/w \rfloor$. The parameter s is called the stepping parameter of the recurrence.

```
package umontreal.iro.lecuyer.hups;

public class F2wCycleBasedLFSR extends CycleBasedPointSetBase2
```

Constructors

`public F2wCycleBasedLFSR (int w, int r, int modQ, int step, int nbcoeff, int coeff[], int nocoeff[])`

Constructs a point set with 2^{rw} points. See the description of the class `F2wStructure` for the meaning of the parameters.

`public F2wCycleBasedLFSR (String filename, int no)`

Constructs a point set after reading its parameters from file `filename`; the parameters are located at line numbered `no` of `filename`. The available files are listed in the description of class `F2wStructure`.

F2wCycleBasedPolyLCG

This class creates a point set based upon a linear congruential sequence in the finite field $\mathbb{F}_{2^w}[z]/P(z)$. The recurrence is

$$q_n(z) = z^s q_{n-1}(z) \mod P(z)$$

where $P(z) \in \mathbb{F}_{2^w}[z]$ has degree r and $q_n(z) = q_{n,1}z^{r-1} + \cdots + q_{n,r} \in \mathbb{F}_{2^w}[z]/P(z)$. The parameter s is called the stepping parameter of the recurrence. The polynomial $P(z)$ is not necessarily the characteristic polynomial of this recurrence, but it can still be used to store the parameters of the recurrence. In the implementation, it is stored in an object of the class `F2wStructure`. See the description of this class for more details on how the polynomial is stored.

Let $\mathbf{x} = (x^{(0)}, \ldots, x^{(p-1)}) \in \mathbb{F}_2^p$ be a p-bit vector. Let us define the function $\phi(\mathbf{x}) = \sum_{i=1}^{p} 2^{-i} x^{(i-1)}$. The point set in t dimensions produced by this class is

$$\{(\phi(\mathbf{y}_0), \phi(\mathbf{y}_1), \ldots, \phi(\mathbf{y}_{t-1}) : (\mathbf{q}_{0,1}, \ldots, \mathbf{q}_{0,r-1}) \in \mathbb{F}_2^{rw}\}$$

where $\mathbf{y}_n = (\mathbf{q}_{n,1}, \ldots, \mathbf{q}_{n,r})$, $\mathbf{q}_{n,i}$ is the representation of $q_{n,i}$ under the polynomial basis of $\mathbb{F}_{2^w}$ over $\mathbb{F}_2$.

```
package umontreal.iro.lecuyer.hups;

public class F2wCycleBasedPolyLCG extends CycleBasedPointSetBase2
```

Constructors

```
public F2wCycleBasedPolyLCG (int w, int r, int modQ, int step, int nbcoeff,
                             int coeff[], int nocoeff[])
```

Constructs a point set with 2^{rw} points. See the description of the class `F2wStructure` for the meaning of the parameters.

```
public F2wCycleBasedPolyLCG (String filename, int no)
```

Constructs a point set after reading its parameters from file `filename`; the parameters are located at line numbered `no` of `filename`. The available files are listed in the description of class `F2wStructure`.

DigitalNet

This class provides the basic structures for storing and manipulating *linear digital nets in base b*, for an arbitrary base $b \geq 2$. We recall that a net contains $n = b^k$ points in s dimensions, where the ith point $\mathbf{u}_i$, for $i = 0, \ldots, b^k - 1$, is defined as follows:

$$\begin{aligned} i &= \sum_{\ell=0}^{k-1} a_{i,\ell} b^\ell, \\ \begin{pmatrix} u_{i,j,1} \\ u_{i,j,2} \\ \vdots \end{pmatrix} &= \mathbf{C}_j \begin{pmatrix} a_{i,0} \\ a_{i,1} \\ \vdots \\ a_{i,k-1} \end{pmatrix}, \\ u_{i,j} &= \sum_{\ell=1}^{\infty} u_{i,j,\ell} b^{-\ell}, \\ \boldsymbol{u}_i &= (u_{i,0}, \ldots, u_{i,s-1}). \end{aligned}$$

In our implementation, the matrices $\mathbf{C}_j$ are $r \times k$, so the expansion of $u_{i,j}$ is truncated to its first r terms. The points are stored implicitly by storing the generator matrices $\mathbf{C}_j$ in a large two-dimensional array of integers, with srk elements.

The points $\mathbf{u}_i$ are enumerated using the Gray code technique as proposed in [1, 38] (see also [9, 15]). With this technique, the b-ary representation of i, $\mathbf{a}_i = (a_{i,0}, \ldots, a_{i,k-1})$, is replaced in Equation (7) by a Gray code representation of i, $\mathbf{g}_i = (g_{i,0}, \ldots, g_{i,k-1})$. The Gray code $\mathbf{g}_i$ used here is defined by $g_{i,k-1} = a_{i,k-1}$ and $g_{i,\ell} = (a_{i,\ell} - a_{i,\ell+1}) \bmod b$ for $\ell = 0, \ldots, k-2$. It has the property that $\mathbf{g}_i = (g_{i,0}, \ldots, g_{i,k-1})$ and $\mathbf{g}_{i+1} = (g_{i+1,0}, \ldots, g_{i+1,k-1})$ differ only in the position of the smallest index ℓ such that $a_{i,\ell} < b - 1$, and we have $g_{i+1,\ell} = (g_{i,\ell} + 1) \bmod b$ in that position.

This Gray code representation permits a more efficient enumeration of the points by the iterators. It changes the order in which the points $\mathbf{u}_i$ are enumerated, but the first b^m points remain the same for every integer m. The ith point of the sequence with the Gray enumeration is the i'th point of the original enumeration, where i' is the integer whose b-ary representation $\mathbf{a}_{i'}$ is given by the Gray code $\mathbf{g}_i$. To enumerate all the points successively, we never need to compute the Gray codes explicitly. It suffices to know the position ℓ of the Gray code digit that changes at each step, and this can be found quickly from the b-ary representation $\mathbf{a}_i$. The digits of each coordinate j of the current point can be updated by adding column ℓ of the generator matrix $\mathbf{C}_j$ to the old digits, modulo b.

One should avoid using the method `getCoordinate`(i, j) for arbitrary values of `i` and `j`, because this is much slower than using an iterator to access successive coordinates.

Digital nets can be randomized in various ways [25, 6, 21, 29]. Several types of randomizations specialized for nets are implemented directly in this class.

A simple but important randomization is the *random digital shift* in base b, defined as follows: replace each digit $u_{i,j,\ell}$ in (8) by $(u_{i,j,\ell}+d_{j,\ell})$ mod b, where the $d_{j,\ell}$'s are i.i.d. uniform over $\{0,\ldots,b-1\}$. This is equivalent to applying a single random shift to all the points in a formal series representation of their coordinates [21, 24]. In practice, the digital shift is truncated to w digits, for some integer $w \geq r$. Applying a digital shift does not change the equidistribution and (t,m,s)-net properties of a point set [15, 19, 24]. Moreover, with the random shift, each point is uniformly distributed over the unit hypercube (but the points are not independent, of course).

A second class of randomizations specialized for digital nets are the *linear matrix scrambles* [25, 6, 15, 29], which multiply the matrices $\mathbf{C}_j$ by a random invertible matrix $\mathbf{M}_j$, modulo b. There are several variants, depending on how $\mathbf{M}_j$ is generated, and on whether $\mathbf{C}_j$ is multiplied on the left or on the right. In our implementation, the linear matrix scrambles are incorporated directly into the matrices $\mathbf{C}_j$ (as in [15]), so they do not slow down the enumeration of points. Methods are available for applying linear matrix scrambles and for removing these randomizations. These methods generate the appropriate random numbers and make the corresponding changes to the $\mathbf{C}_j$'s. A copy of the original $\mathbf{C}_j$'s is maintained, so the point set can be returned to its original unscrambled state at any time. When a new linear matrix scramble is applied, it is always applied to the *original* generator matrices. The method `resetGeneratorMatrices` removes the current matrix scramble by resetting the generator matrices to their original state. On the other hand, the method `eraseOriginalGeneratorMatrices` replaces the original generator matrices by the current ones, making the changes permanent. This is useful if one wishes to apply two or more linear matrix scrambles on top of each other.

Linear matrix scrambles are usually combined with a random digital shift; this combination is called an *affine matrix scramble* [29]. These two randomizations are applied via separate methods. The linear matrix scrambles are incorporated into the matrices $\mathbf{C}_j$ whereas the digital random shift is stored and applied separately, independently of the other scramblings.

Applying a digital shift or a linear matrix scramble to a digital net invalidates all iterators for that randomized point, because each iterator uses a *cached* copy of the current point, which is updated only when the current point index of that iterator changes, and the update also depends on the cached copy of the previous point. After applying any kind of scrambling, the iterators must be reinitialized to the *initial point* by invoking `resetCurPointIndex` or reinstantiated by the `iterator` method (this is not done automatically).

```
package umontreal.iro.lecuyer.hups;

public class DigitalNet extends PointSet
```

Constructor

`public DigitalNet ()`

Empty constructor.

Methods

`public double getCoordinateNoGray (int i, int j)`

Returns $u_{i,j}$, the coordinate j of point i, the points being enumerated in the standard order (no Gray code).

`public PointSetIterator iteratorNoGray()`

This iterator does not use the Gray code. Thus the points are enumerated in the order of their first coordinate before randomization.

`public void addRandomShift (int d1, int d2, RandomStream stream)`

Adds a random digital shift to all the points of the point set, using stream `stream` to generate the random numbers. For each coordinate j from `d1` to `d2-1`, the shift vector $(d_{j,0}, \ldots, d_{j,k-1})$ is generated uniformly over $\{0, \ldots, b-1\}^k$ and added modulo b to the digits of all the points. After adding a digital shift, all iterators must be reconstructed or reset to zero.

`public void addRandomShift (RandomStream stream)`

Same as `addRandomShift(0, dim, stream)`, where `dim` is the dimension of the digital net.

`public void leftMatrixScramble (RandomStream stream)`

Applies a linear scramble by multiplying each $\mathbf{C}_j$ on the left by a $w \times w$ nonsingular lower-triangular matrix $\mathbf{M}_j$, as suggested by Matoušek [25] and implemented by Hong and Hickernell [15]. The diagonal entries of each matrix $\mathbf{M}_j$ are generated uniformly over $\{1, \ldots, b-1\}$, the entries below the diagonal are generated uniformly over $\{0, \ldots, b-1\}$, and all these entries are generated independently. This means that in base $b = 2$, all diagonal elements are equal to 1. [3]

`public void leftMatrixScrambleDiag (RandomStream stream)`

Similar to `leftMatrixScramble` except that all the off-diagonal elements of the $\mathbf{M}_j$ are 0.

`public void leftMatrixScrambleFaurePermut (RandomStream stream, int sb)`

Similar to `leftMatrixScramble` except that the diagonal elements of each matrix $\mathbf{M}_j$ are chosen from a restricted set of the best integers as calculated by Faure [**?**]. They are generated uniformly over the first `sb` elements of array F, where F is made up of a permutation of the integers $[1..(b-1)]$. These integers are sorted by increasing order of the upper bounds of the extreme discrepancy for the given integer.

`public void leftMatrixScrambleFaurePermutDiag (RandomStream stream, int sb)`

Similar to `leftMatrixScrambleFaurePermut` except that all off-diagonal elements are 0.

`public void leftMatrixScrambleFaurePermutAll (RandomStream stream, int sb)`

Similar to `leftMatrixScrambleFaurePermut` except that the elements under the diagonal are also chosen from the same restricted set as the diagonal elements.

[3] From Richard: Les matrices de `leftMatrixScramble` sont carrées et triangulaires inférieures. PL pense qu'il faut considérer la possibilité de rajouter des lignes à ces matrices pour pouvoir randomiser plus les derniers chiffres ou les derniers bits.

`public void iBinomialMatrixScramble (RandomStream stream)`

Applies the i-binomial matrix scramble proposed by Tezuka [39] (see also [29]). This multiplies each $\mathbf{C}_j$ on the left by a $w \times w$ nonsingular lower-triangular matrix $\mathbf{M}_j$ as in `leftMatrixScramble`, but with the additional constraint that all entries on any given diagonal or subdiagonal of $\mathbf{M}_j$ are identical.

`public void iBinomialMatrixScrambleFaurePermut (RandomStream stream, int sb)`

Similar to `iBinomialMatrixScramble` except that the diagonal elements of each matrix $\mathbf{M}_j$ are chosen as in `leftMatrixScrambleFaurePermut`.

`public void iBinomialMatrixScrambleFaurePermutDiag (RandomStream stream, int sb)`

Similar to `iBinomialMatrixScrambleFaurePermut` except that all the off-diagonal elements are 0.

`public void iBinomialMatrixScrambleFaurePermutAll (RandomStream stream, int sb)`

Similar to `iBinomialMatrixScrambleFaurePermut` except that the elements under the diagonal are also chosen from the same restricted set as the diagonal elements.

`public void stripedMatrixScramble (RandomStream stream)`

Applies the *striped matrix scramble* proposed by Owen [29]. It multiplies each $\mathbf{C}_j$ on the left by a $w \times w$ nonsingular lower-triangular matrix $\mathbf{M}_j$ as in `leftMatrixScramble`, but with the additional constraint that in any column, all entries below the diagonal are equal to the diagonal entry, which is generated randomly over $\{1, \ldots, b-1\}$. Note that for $b = 2$, the matrices $\mathbf{M}_j$ become deterministic, with all entries on and below the diagonal equal to 1.

`public void stripedMatrixScrambleFaurePermutAll (RandomStream stream, int sb)`

Similar to `stripedMatrixScramble` except that the elements on and under the diagonal of each matrix $\mathbf{M}_j$ are chosen as in `leftMatrixScrambleFaurePermut`.

`public void rightMatrixScramble (RandomStream stream)`

Applies a linear scramble by multiplying each $\mathbf{C}_j$ on the right by a single $k \times k$ nonsingular upper-triangular matrix $\mathbf{M}$, as suggested by Faure and Tezuka [6] (see also [15]). The diagonal entries of the matrix $\mathbf{M}$ are generated uniformly over $\{1, \ldots, b-1\}$, the entries above the diagonal are generated uniformly over $\{0, \ldots, b-1\}$, and all the entries are generated independently. The effect of this scramble is only to change the order in which the points are generated. If one computes the average value of a function over *all* the points of a given digital net, or over a number of points that is a power of the basis, then this scramble makes no difference.

`public void resetGeneratorMatrices()`

Restores the original generator matrices. This removes the current linear matrix scrambles.

`public void eraseOriginalGeneratorMatrices()`

Erases the original generator matrices and replaces them by the current ones. The current linear matrix scrambles thus become *permanent*. This is useful if we want to apply several scrambles in succession to a given digital net.

`public void printGeneratorMatrices (int s)`

Prints the generator matrices in standard form for dimensions 1 to s.

DigitalSequence

This abstract class describes methods specific to digital sequences. Concrete classes must implement the `extendSequence` method that increases the number of points of the digital sequence. Calling the methods `toNet` or `toNetShiftCj` will transform the digital sequence into a digital net, which has a fixed number of points n.

```
package umontreal.iro.lecuyer.hups;

public abstract class DigitalSequence extends DigitalNet
```

`public abstract void extendSequence (int k);`

Increases the number of points to $n = b^k$ from now on.

`public DigitalNet toNet()`

Transforms this digital sequence into a digital net without changing the coordinates of the points. Returns the digital net.

`public DigitalNet toNetShiftCj()`

Transforms this digital sequence into a digital net by adding one dimension and shifting all coordinates by one position. The first coordinate of point i is i/n, where n is the total number of points. Thus if the coordinates of a point of the digital sequence were $(x_0, x_1, x_2, \ldots, x_{s-1})$, then the coordinates of the point of the digital net will be $(i/n, x_0, x_1, \ldots, x_{s-1})$. In other words, for the digital net, $\mathbf{C}_0$ is the reflected identity and for $j \geq 1$, the $\mathbf{C}_j$ used is the $\mathbf{C}_{j-1}$ of the digital sequence. If the digital sequence uses a digital shift, then the digital net will include the digital shift with one more dimension also. Returns the digital net.

`public PointSetIterator iteratorShift()`

Similar to `iterator`, except that the first coordinate of the points is i/n, the second coordinate is obtained via the generating matrix $\mathbf{C}_0$, the next one via $\mathbf{C}_1$, and so on. Thus, this iterator shifts all coordinates of each point one position to the right and sets the first coordinate of point i to i/n, so that the points enumerated with this iterator have one more dimension. A digital shift, if present, will have one more dimension also. This iterator uses the Gray code.

`public PointSetIterator iteratorShiftNoGray()`

This iterator shifts all coordinates of each point one position to the right and sets the first coordinate of point i to i/n, so that the points enumerated with this iterator have one more dimension. This iterator does not use the Gray code; the points are enumerated in the order of their first coordinate before randomization. A digital shift, if present, will have one more dimension also.

```
// Any number of comment lines starting with //
b              // Base
k              // Number of columns
r              // Maximal number of rows
n              // Number of points = b^k
s              // Maximal dimension of points

// dim = 1
c11  c21  ...  cr1
c12  c22  ...  cr2
      :
c1k  c2k  ...  crk

// dim = 2
      :

// dim = s
c11  c21  ...  cr1
c12  c22  ...  cr2
      :
c1k  c2k  ...  crk
```

DigitalNetFromFile

This class allows us to read the parameters defining a digital net either from a file, or from a URL address on the World Wide Web. The parameters used in building the net are those defined in class `DigitalNet`. The format of the data files must be the following:

The figure above gives the general format of the data file needed by `DigitalNetFromFile`. The values of the parameters on the left must appear in the file as integers. On the right of each parameter, there is an optional comment that is disregarded by the reader program. In general, the Java line comments `//` are accepted anywhere and will ensure that the rest of the line is dropped by the reader. Blank lines are also disregarded by the reader program. For each dimension, there must be a $k \times r$ matrix of integers in $\{0, 1, \ldots, b-1\}$ (note that the matrices must appear in transposed form).

The predefined files of parameters are kept in different directories, depending on the criteria used in the searches for the parameters defining the digital net. These files have all been stored at the address `http://www.iro.umontreal.ca/~simardr/ssj/data`. Each file contains the parameters for a specific digital net. The name of the files gives information about the main parameters of the digital net. For example, the file named `Edel/OOA2/B3S13R9C9St6` contains the parameters for a digital net proposed by Yves Edel

(see `http://www.mathi.uni-heidelberg.de/~yves/index.html`) based on ordered orthogonal arrays; the digital net has base `B = 3`, dimension `S = 13`, the generating matrices have `R = 9` rows and `C = 9` columns, and the strength of the net is `St = 6`.

```
package umontreal.iro.lecuyer.hups;

public class DigitalNetFromFile extends DigitalNet
```

Constructors

```
public DigitalNetFromFile (String filename, int r1, int w, int s1)
       throws MalformedURLException, IOException
```

Constructs a digital net after reading its parameters from file `filename`. If a file named `filename` can be found relative to the program's directory, then the parameters will be read from this file; otherwise, they will be read from the file named `filename` in the `ssj.jar` archive. If `filename` is a URL string, it will be read on the World Wide Web. For example, to construct a digital net from the parameters in file `B3S13R9C9St6` in the current directory, one must give the string `"B3S13R9C9St6"` as argument to the constructor. As an example of a file read from the WWW, one may give as argument to the constructor the string `"http://www.iro.umontreal.ca/~simardr/ssj/data/Edel/OOA3/B3S13R6C6St4"`. Parameter `w` gives the number of digits of resolution, `r1` is the number of rows, and `s1` is the dimension. Restrictions: `s1` must be less than the maximal dimension, and `r1` less than the maximal number of rows in the data file. Also $\texttt{w} \geq \texttt{r1}$.

```
public DigitalNetFromFile (String filename, int s)
       throws MalformedURLException, IOException
```

Same as `DigitalNetFromFile(filename, r, r, s)` where `s` is the dimension and `r` is given in data file `filename`.

Methods

```
public String toStringDetailed()
```

Writes the parameters and the generating matrices of this digital net to a string. This is useful to check that the file parameters have been read correctly.

```
public static String listDir (String dirname) throws IOException
```

Lists all files (or directories) in directory `dirname`. Only relative pathnames should be used. The files are parameter files used in defining digital nets. For example, calling `listDir("")` will give the list of the main data directory in SSJ, while calling `listDir("Edel/OOA2")` will give the list of all files in directory `Edel/OOA2`.

```
public static void listDirHTML (String dirname, String filename)
       throws IOException
```

Creates a list of all data files in directory `dirname` and writes that list in format HTML in output file `filename`. Each data file contains the parameters required to build a digital net. The resulting list contains a line for each data file giving the name of the file, the base, the dimension, the number of rows and the number of columns of the corresponding digital net.

FaureSequence

This class implements digital nets or digital sequences formed by the first $n = b^k$ points of the Faure sequence in base b. Values of n up to 2^{31} are allowed. One has $r = k$. The generator matrices are

$$\mathbf{C}_j = \mathbf{P}^j \mod b \tag{11}$$

for $j = 0, \ldots, s-1$, where $\mathbf{P}$ is a $k \times k$ upper triangular matrix whose entry (l, c) is the number of combinations of l objects among c, $\binom{c}{l}$, for $l \leq c$ and is 0 for $l > c$. The matrix $\mathbf{C}_0$ is the identity, $\mathbf{C}_1 = \mathbf{P}$, and the other $\mathbf{C}_j$'s can be defined recursively via $\mathbf{C}_j = \mathbf{P}\mathbf{C}_{j-1} \mod b$. Our implementation uses the recursion

$$\binom{c}{l} = \binom{c-1}{l} + \binom{c-1}{l-1} \tag{12}$$

to evaluate the binomial coefficients in the matrices $\mathbf{C}_j$, as suggested by Fox [7] (see also [9], page 301). The entries $x_{j,l,c}$ of $\mathbf{C}_j$ are computed as follows:

$$\begin{aligned}
x_{j,c,c} &= 1 && \text{for } c = 0, \ldots, k-1, \\
x_{j,0,c} &= j x_{j,0,c-1} && \text{for } c = 1, \ldots, k-1, \\
x_{j,l,c} &= x_{j,l-1,c-1} + j x_{j,l,c-1} && \text{for } 2 \leq c < l \leq k-1, \\
x_{j,l,c} &= 0 && \text{for } c > l \text{ or } l \geq k.
\end{aligned}$$

For any integer $m > 0$ and $\nu \geq 0$, if we look at the vector $(u_{i,j,1}, \ldots, u_{i,j,m})$ (the first m digits of coordinate j of the output) when i goes from νb^m to $(\nu+1)b^m - 1$, this vector takes each of its b^m possible values exactly once. In particular, for $\nu = 0$, $u_{i,j}$ visits each value in the set $\{0, 1/b^m, 2/b^m, \ldots, (b^m-1)/b^m\}$ exactly once, so all one-dimensional projections of the point set are identical. However, the values are visited in a different order for the different values of j (otherwise all coordinates would be identical). For $j = 0$, they are visited in the same order as in the van der Corput sequence in base b.

An important property of Faure nets is that for any integers $m > 0$ and $\nu \geq 0$, the point set $\{\boldsymbol{u}_i \text{ for } i = \nu b^m, \ldots, (\nu+1)b^m - 1\}$ is a $(0, m, s)$-net in base b. In particular, for $n = b^k$, the first n points form a $(0, k, s)$-net in base b. The Faure nets are also projection-regular and dimension-stationary (see [21] for definitions of these properties).

To obtain digital nets from the *generalized Faure sequence* [38], where $\mathbf{P}_j$ is left-multiplied by some invertible matrix $\mathbf{A}_j$, it suffices to apply an appropriate matrix scramble (e.g., via `leftMatrixScramble`). This changes the order in which $u_{i,j}$ visits its different values, for each coordinate j, but does not change the set of values that are visited. The $(0, m, s)$-net property stated above remains valid.

```
package umontreal.iro.lecuyer.hups;

public class FaureSequence extends DigitalSequence
```

Constructors

`public FaureSequence (int b, int k, int r, int w, int dim)`

Constructs a digital net in base b, with $n = b^k$ points and w output digits, in `dim` dimensions. The points are the first n points of the Faure sequence. The generator matrices $\mathbf{C}_j$ are $r \times k$. Unless, one plans to apply a randomization on more than k digits (e.g., a random digital shift for $w > k$ digits, or a linear scramble yielding $r > k$ digits), one should take $w = r = k$ for better computational efficiency. Restrictions: `dim` ≤ 500 and $b^k \leq 2^{31}$.

`public FaureSequence (int n, int dim)`

Same as `FaureSequence(b, k, w, w, dim)` with base b equal to the smallest prime larger or equal to `dim`, and with *at least* `n` points. The values of k, r, and w are taken as $k = \lceil \log_b n \rceil$ and $r = w = \max(k, \lfloor 30/\log_2 b \rfloor)$.

DigitalNetBase2

A special case of `DigitalNet` for the base $b = 2$. The implementation exploit the binary nature of computers and is much more efficient than for the general case. Binary expansions are easy to obtain because the computer already uses them internally. The generator matrices $\mathbf{C}_j$ are stored in a large array of size sk. The c-th column of $\mathbf{C}_j$, for $c = 0, \ldots, k-1$, is stored at position $jk + c$ of that array, as a 32-bit integer. For all derived classes, the above 32-bit integer must be of the form $[00\cdots C_0 C_1 \cdots C_{r-1}]$. The value of k cannot exceed 31 (32 is not allowed because Java does not have 32-bit unsigned integers). The value of w is always 31. [4]

The random digital shift in base 2 corresponds to a random XOR. It can be applied via the method `addRandomShift`.

```
package umontreal.iro.lecuyer.hups;

public class DigitalNetBase2 extends DigitalNet
```

`public void printGeneratorMatrices (int s)`

Prints the generator matrices as bit matrices in standard form for dimensions 1 to s.

`public void printGeneratorMatricesTrans (int s)`

Prints the generator matrices transposed in the form of integers for dimensions 1 to s. Each integer corresponds to a column of bits.

`public PointSetIterator iteratorNoGray()`

This iterator does not use the Gray code. Thus the points are enumerated in the order of their first coordinate before randomization.

[4] From Pierre: In this implementation, w is always used in place of r so the value of r is not used.

DigitalSequenceBase2

This abstract class describes methods specific to digital sequences in base 2. Concrete classes must implement the `extendSequence` method that increases the number of points of the digital sequence. Calling the methods `toNet` or `toNetShiftCj` will transform the digital sequence into a digital net, which has a fixed number of points n.

```
package umontreal.iro.lecuyer.hups;

public abstract class DigitalSequenceBase2 extends DigitalNetBase2
```

`public abstract void extendSequence (int k);`

Increases the number of points to $n = 2^k$ from now on.

`public DigitalNetBase2 toNet()`

Transforms this digital sequence into a digital net without changing the coordinates of the points. Returns the digital net.

`public DigitalNetBase2 toNetShiftCj()`

Transforms this digital sequence into a digital net by adding one dimension and shifting all coordinates by one position. The first coordinate of point i is i/n, where n is the total number of points. Thus if the coordinates of a point of the digital sequence were $(x_0, x_1, x_2, \ldots, x_{s-1})$, then the coordinates of the point of the digital net will be $(i/n, x_0, x_1, \ldots, x_{s-1})$. In other words, for the digital net, $\mathbf{C}_0$ is the reflected identity and for $j \geq 1$, the $\mathbf{C}_j$ used is the $\mathbf{C}_{j-1}$ of the digital sequence. If the digital sequence uses a digital shift, then the digital net will include the digital shift with one more dimension also. Returns the digital net.

`public PointSetIterator iteratorShift()`

Similar to `iterator`, except that the first coordinate of the points is i/n, the second coordinate is obtained via the generating matrix $\mathbf{C}_0$, the next one via $\mathbf{C}_1$, and so on. Thus, this iterator shifts all coordinates of each point one position to the right and sets the first coordinate of point i to i/n, so that the points enumerated with this iterator have one more dimension. A digital shift, if present, will have one more dimension also. This iterator uses the Gray code.

`public PointSetIterator iteratorShiftNoGray()`

This iterator shifts all coordinates of each point one position to the right and sets the first coordinate of point i to i/n, so that the points enumerated with this iterator have one more dimension. This iterator does not use the Gray code: the points are enumerated in the order of their first coordinate before randomization. A digital shift, if present, will have one more dimension also.

```
// Any number of comment lines starting with //
2   // Base
k   // Number of columns
r   // Number of rows
n   // Number of points = 2^k
s   // Dimension of points

// dim = 1
a_1  // = 2^30 B_11 + 2^29 B_21 + ... + 2^(31-r) B_r1
a_2  // = 2^30 B_12 + 2^29 B_22 + ... + 2^(31-r) B_r2
...
a_k

// dim = 2
...

// dim = s
a_1
a_2
...
a_k
```

DigitalNetBase2FromFile

This class allows us to read the parameters defining a digital net *in base 2* either from a file, or from a URL address on the World Wide Web. See the documentation in `DigitalNetFromFile`. The parameters used in building the net are those defined in class `DigitalNetBase2`. The format of the data files must be the following (where B is any C_j):

For each dimension j, there must be a k-vector of 32-bit integers (the a_i) corresponding to the columns of $\mathbf{C}_j$. The correspondance is such that integer $a_i = 2^{30}(\mathbf{C}_j)_{1i} + 2^{29}(\mathbf{C}_j)_{2i} + \cdots + 2^{31-r}(\mathbf{C}_j)_{ri}$.

```
package umontreal.iro.lecuyer.hups;

public class DigitalNetBase2FromFile extends DigitalNetBase2
```

Constructor

```
public DigitalNetBase2FromFile (String filename, int r1, int w, int s1)
      throws IOException, MalformedURLException
```

Constructs a digital net in base 2 after reading its parameters from file `filename`. See the documentation in `DigitalNetFromFile`. Parameter `w` gives the number of bits of resolution,

`r1` is the number of rows, and `s1` is the dimension. Restrictions: `s1` must be less than the maximal dimension, and `r1` less than the maximal number of rows in the data file. Also `w` $\geq$ `r1`.

```
public DigitalNetBase2FromFile (String filename, int s1)
     throws IOException, MalformedURLException
```

Same as `DigitalNetBase2FromFile(filename, r, 31, s1)` where `s1` is the dimension and `r` is given in data file `filename`.

Methods

```
public String toStringDetailed()
```

Writes the parameters and the generating matrices of this digital net to a string. This is useful to check that the file parameters have been read correctly.

```
public static String listDir (String dirname) throws IOException
```

Lists all files (or directories) in directory `dirname`. Only relative pathnames should be used. The files are parameter files used in defining digital nets. For example, calling `listDir("")` will give the list of the main data directory in SSJ, while calling `listDir("Edel/OOA2")` will give the list of all files in directory `Edel/OOA2`.

SobolSequence

This class implements digital nets or digital sequences in base 2 formed by the first $n = 2^k$ points of a Sobol' sequence [35, 36]. Values of n up to 2^{31} are allowed, in a maximum of 360 dimensions.

In Sobol's proposal, the generator matrices $\mathbf{C}_j$ are upper triangular matrices defined by a set of *direction numbers*

$$v_{j,c} = m_{j,c}2^{-c} = \sum_{l=1}^{c} v_{j,c,l}2^{-l},$$

where each $m_{j,c}$ is an *odd* integer smaller than 2^c, for $c = 1, \ldots, k$ and $j = 0, \ldots, s-1$. The digit $v_{j,c,l}$ is the element (l, c) of $\mathbf{C}_j$, so $v_{j,c}$ represents column c of $\mathbf{C}_j$. One can also write

$$m_{j,c} = \sum_{l=1}^{c} v_{j,c,l}2^{c-l},$$

so column c of $\mathbf{C}_j$ contains the c digits of the binary expansion of $m_{j,c}$, from the most to the least significant, followed by $w - c$ zeros, where w is the number of output digits. Since each $m_{j,c}$ is odd, the first k rows of each $\mathbf{C}_j$ form a non-singular upper triangular matrix whose diagonal elements are all ones.

For each dimension j, the integers $m_{j,c}$ are defined by selecting a primitive polynomial over $\mathbb{F}_2$ of degree c_j,

$$f_j(z) = z^{c_j} + a_{j,1}z^{c_j-1} + \cdots + a_{j,c_j},$$

and the first c_j integers $m_{j,0}, \ldots, m_{j,c_j-1}$. Then the following integers $m_{j,c_j}, m_{j,c_j+1}, \ldots$ are determined by the recurrence

$$m_{j,c} = 2a_{j,1}m_{j,c-1} \oplus \cdots \oplus 2^{c_j-1}a_{j,c_j-1}m_{j,c-c_j+1} \oplus 2^{c_j}m_{j,c-c_j} \oplus m_{j,c-c_j}$$

for $c \geq c_j$, or equivalently,

$$v_{j,c,l} = a_{j,1}v_{j,c-1,l} \oplus \cdots \oplus a_{j,c_j-1}v_{j,c-c_j+1,l} \oplus v_{j,c-c_j,l} \oplus v_{j,c-c_j,l+c_j}$$

for $l \geq 0$, where $\oplus$ means bitwise exclusive or (i.e., bitwise addition modulo 2). Sobol' [35] has shown that with this construction, if the primitive polynomials $f_j(z)$ are all distinct, one obtains a (t, s)-sequence whose t-value does not exceed $c_0 + \cdots + c_{s-1} + 1 - s$. He then suggested to list the set of all primitive polynomials over $\mathbb{F}_2$ by increasing order of degree, starting with $f_0(z) \equiv 1$ (whose corresponding matrix $\mathbf{C}_0$ is the identity), and take $f_j(z)$ as the $(j+1)$th polynomial in the list, for $j \geq 0$.

This list of primitive polynomials, as well as default choices for the direction numbers, are stored in precomputed tables. The ordered list of primitive polynomials is the same as in [23] and was taken from Florent Chabaud's web site, at `http://fchabaud.free.fr/`. Each polynomial $f_j(z)$ is stored in the form of the integer $2^{c_j} + a_{j,1}2^{c_j-1} + \cdots + a_{j,c_j}$, whose binary representation gives the polynomial coefficients.

For the set of direction numbers, there are several possibilities, based on different selection criteria. The original values proposed by Sobol', and implemented in the code of Bratley and Fox [2] for $j \leq 40$, were selected in terms of his properties A and A', which are equivalent to s-distribution with one and two bits of accuracy, respectively. The default direction numbers used here have been taken from [23]. For $j \leq 40$, they are the same as in [2]. [5]

```
package umontreal.iro.lecuyer.hups;

public class SobolSequence extends DigitalSequenceBase2
```

Constructors

`public SobolSequence (int k, int w, int dim)`

Constructs a new digital net with $n = 2^k$ points and w output digits, in `dim` dimensions, formed by taking the first n points of the Sobol' sequence. The generator matrices $\mathbf{C}_j$ are $w \times k$. Restrictions: $0 \leq k \leq 30$ and $k \leq w$.

`public SobolSequence (int n, int dim)`

Constructs a Sobol point set with *at least* `n` points and 31 output digits, in `dim` dimensions. Equivalent to `SobolSequence (k, 31, dim)` with $k = \lceil \log_2 n \rceil$.

[5] From Pierre: We should eventually have other choices for the direction numbers.

NiedSequenceBase2

This class implements digital sequences constructed from the Niederreiter sequence in base 2. For details on these point sets, see [3].

```
package umontreal.iro.lecuyer.hups;

public class NiedSequenceBase2 extends DigitalSequenceBase2
```

Constructor

`public NiedSequenceBase2 (int k, int w, int dim)`

Constructs a new digital sequence in base 2 from the first $n = 2^k$ points of the Niederreiter sequence, with w output digits, in `dim` dimensions. The generator matrices $\mathbf{C}_j$ are $w \times k$. Restrictions: $0 \leq k \leq 30$, $k \leq w$, and `dim` ≤ 318.

NiedXingSequenceBase2

This class implements digital sequences based on the Niederreiter-Xing sequence in base 2. For details on these point sets, see [27, 33].

```
package umontreal.iro.lecuyer.hups;

public class NiedXingSequenceBase2 extends DigitalSequenceBase2
```

Constructors

`public NiedXingSequenceBase2 (int k, int w, int dim)`

Constructs a new Niederreiter-Xing digital sequence in base 2 with $n = 2^k$ points and w output digits, in `dim` dimensions. The generator matrices $\mathbf{C}_j$ are $w \times k$ and the numbers making the bit matrices are taken from Pirsic's site. The bit matrices from Pirsic's site are transposed to be consistent with SSJ, and at most 30 bits of the matrices are used. Restrictions: $0 \leq k \leq 30$, $k \leq w$, and `dim` ≤ 32.

F2wNetLFSR

This class implements a digital net in base 2 starting from a linear feedback shift register generator. It is exactly the same point set as the one defined in the class `F2wCycleBasedLFSR`. See the description of this class for more details on the way the point set is constructed.

Constructing a point set using this class, instead of using `F2wCycleBasedLFSR`, makes SSJ construct a digital net in base 2. This is useful if one wants to randomize using one of the randomizations included in the class `DigitalNet`.

```
package umontreal.iro.lecuyer.hups;

public class F2wNetLFSR extends DigitalNetBase2
```

Constructors

```
public F2wNetLFSR (int w, int r, int modQ, int step, int nbcoeff,
                   int coeff[], int nocoeff[], int dim)
```

Constructs a point set with 2^{rw} points. See the description of the class `F2wStructure` for the meaning of the parameters.

```
public F2wNetLFSR (String filename, int no, int dim)
```

Constructs a point set after reading its parameters from file `filename`; the parameters are located at line numbered `no` of `filename`. The available files are listed in the description of class `F2wStructure`.

F2wNetPolyLCG

This class implements a digital net in base 2 starting from a polynomial LCG in $\mathbb{F}_{2^w}[z]/P(z)$. It is exactly the same point set as the one defined in the class `F2wCycleBasedPolyLCG`. See the description of this class for more details on the way the point set is constructed.

Constructing a point set using this class, instead of using `F2wCycleBasedPolyLCG`, makes SSJ construct a digital net in base 2. This is useful if one wants to randomize using one of the randomizations included in the class `DigitalNet`.

Note: This class in not operational yet!

```
package umontreal.iro.lecuyer.hups;

public class F2wNetPolyLCG extends DigitalNetBase2
```

Constructors

```
public F2wNetPolyLCG (int type, int w, int r, int modQ, int step,
                      int nbcoeff, int coeff[], int nocoeff[], int dim)
```

Constructs a point set with 2^{rw} points. See the description of the class `F2wStructure` for the meaning of the parameters.

```
public F2wNetPolyLCG (String filename, int no, int dim)
```

Constructs a point set after reading its parameters from file `filename`; the parameters are located at line numbered `no` of `filename`. The available files are listed in the description of class `F2wStructure`.

RadicalInverse

This class implements basic methods for working with radical inverses of integers in an arbitrary basis b. These methods are used in classes that implement point sets and sequences based on the van der Corput sequence (the Hammersley nets and the Halton sequence, for example).

We recall that for a k-digit integer i whose digital b-ary expansion is

$$i = a_0 + a_1 b + \ldots + a_{k-1} b^{k-1},$$

the *radical inverse* in base b is

$$\psi_b(i) = a_0 b^{-1} + a_1 b^{-2} + \cdots + a_{k-1} b^{-k}. \tag{13}$$

The *van der Corput sequence in base b* is the sequence $\psi_b(0), \psi_b(1), \psi_b(2), \ldots$

Note that $\psi_b(i)$ cannot always be represented exactly as a floating-point number on the computer (e.g., if b is not a power of two). For an exact representation, one can use the integer

$$b^k \psi_b(i) = a_{k-1} + \cdots + a_1 b^{k-2} + a_0 b^{k-1},$$

which we called the *integer radical inverse* representation. This representation is simply a mirror image of the digits of the usual b-ary representation of i.

It is common practice to permute locally the values of the van der Corput sequence. One way of doing this is to apply a permutation to the digits of i before computing $\psi_b(i)$. That is, for a permutation π of the digits $\{0, \ldots, b-1\}$,

$$\psi_b(i) = \sum_{r=0}^{k-1} a_r b^{-r-1}$$

is replaced by

$$\sum_{r=0}^{k-1} \pi(a_r) b^{-r-1}.$$

Applying such a permutation only changes the order in which the values of $\psi_b(i)$ are enumerated. For every integer k, the first b^k values that are enumerated remain the same (they are the values of $\psi_b(i)$ for $i = 0, \ldots, b^k - 1$), but they are enumerated in a different order. Often, different permutations π will be applied for different coordinates of a point set.

The permutation π can be deterministic or random. One (deterministic) possibility implemented here is the Faure permutation σ_b of $\{0, \ldots, b-1\}$ defined as follows [5]. For $b = 2$, take $\sigma = I$, the identical permutation. For *even* $b = 2c > 2$, take

$$\sigma[i] = 2\tau[i] \qquad i = 0, 1, \ldots, c-1 \tag{14}$$

$$\sigma[i+c] = 2\tau[i] + 1 \qquad i = 0, 1, \ldots, c-1 \tag{15}$$

where $\tau[i]$ is the Faure permutation for base c. For *odd* $b = 2c + 1$, take

$$\begin{aligned} \sigma[c] &= c & (16)\\ \sigma[i] &= \tau[i], \qquad \text{if } 0 \le \tau[i] < c & (17)\\ \sigma[i] &= \tau[i] + 1, \qquad \text{if } c \le \tau[i] < 2c & (18) \end{aligned}$$

for $0 \le i < c$, and take

$$\begin{aligned} \sigma[i] &= \tau[i-1], \qquad \text{if } 0 \le \tau[i-1] < c & (19)\\ \sigma[i] &= \tau[i-1] + 1, \qquad \text{if } c \le \tau[i-1] < 2c & (20) \end{aligned}$$

for $c < i \le 2c$, and where $\tau[i]$ is the Faure permutation for base c. The Faure permutations give very small discrepancies (amongst the best known today) for small bases.

```
package umontreal.iro.lecuyer.hups;

public class RadicalInverse
```

Constructor

`public RadicalInverse (int b, double x0)`

Initializes the base of this object to b and its first value of x to `x0`.

Methods

`public static int[] getPrimes (int n)`

Provides an elementary method for obtaining the first n prime numbers larger than 1. Creates and returns an array that contains these numbers. This is useful for determining the prime bases for the different coordinates of the Halton sequence and Hammersley nets.

`public static double radicalInverse (int b, long i)`

Computes the radical inverse of i in base b. If $i = \sum_{r=0}^{k-1} a_r b^r$, the method computes and returns

$$x = \sum_{r=0}^{k-1} a_r b^{-r-1}.$$

`public static double nextRadicalInverse (double invb, double x)`

A fast method that incrementally computes the radical inverse x_{i+1} in base b from $x_i =$ `x` $= \psi_b(i)$, using addition with *rigthward carry*. The parameter `invb` is equal to $1/b$. Using long incremental streams (i.e., calling this method several times in a row) cause increasing inaccuracy in x. Thus the user should recompute the radical inverse directly by calling `radicalInverse` every once in a while (i.e. in every few thousand calls).

`public double nextRadicalInverse ()`

A fast method that incrementally computes the radical inverse x_{i+1} in base b from $x_i = \psi_b(i)$, using addition with *rigthward carry* as described in [40]. Since using long incremental

streams (i.e., calling this method several times in a row) cause increasing inaccuracy in x, the method recomputes the radical inverse directly from i by calling `radicalInverse` once in every 1000 calls.

`public static void reverseDigits (int k, int bdigits[], int idigits[])`

Given the k b-ary digits of i in `bdigits`, returns the k digits of the integer radical inverse of i in `idigits`. This simply reverses the order of the digits.

`public static int integerRadicalInverse (int b, int i)`

Computes the integer radical inverse of i in base b, equal to $b^k\psi_b(i)$ if i has k b-ary digits.

`public static int nextRadicalInverseDigits (int b, int k, int idigits[])`

Given the k digits of the integer radical inverse of i in `bdigits`, in base b, this method replaces them by the digits of the integer radical inverse of $i + 1$ and returns their number. The array must be large enough to hold this new number of digits.

`public static void getFaurePermutation (int b, int[] pi)`

Computes the Faure permutation [5] σ_b of the set $\{0, \ldots, b-1\}$ and puts it in array `pi`. See the description in the introduction above.

`public static double permutedRadicalInverse (int b, int[] pi, int i)`

Computes the radical inverse of i in base b, where the digits are permuted using the permutation π. If $i = \sum_{r=0}^{k-1} a_r b^r$, the method will compute and return

$$x = \sum_{r=0}^{k-1} \pi[a_r] b^{-r-1}.$$

HammersleyPointSet

This class implements *Hammersley point sets*, which are defined as follows. Let $2 = b_1 < b_2 < \cdots$ denote the sequence of all prime numbers by increasing order. The Hammersley point set with n points in s dimensions contains the points

$$\mathbf{u}_i = (i/n, \psi_{b_1}(i), \psi_{b_2}(i), \ldots, \psi_{b_{s-1}}(i)), \tag{21}$$

for $i = 0, \ldots, n-1$, where ψ_b is the radical inverse function in base b, defined in `RadicalInverse`. This class is not a subclass of `DigitalNet`, because the basis is not the same for all coordinates. We do obtain a net in a generalized sense if $n = b_1^{k_1} = b_2^{k_2} = \cdots = b_{s-1}^{k_{s-1}}$ for some integers $k_1, \ldots, k_{s-1}$.

The points of a Hammersley point set can be "scrambled" by applying a permutation to the digits of i before computing each coordinate via (5). If

$$i = a_0 + a_1 b_j + \ldots + a_{k_j-1} b_j^{k_j-1},$$

and π_j is a permutation of the digits $\{0, \ldots, b_j - 1\}$, then

$$\psi_{b_j}(i) = \sum_{r=0}^{k_j-1} a_r b_j^{-r-1}$$

is replaced in (5) by

$$u_{i,j} = \sum_{r=0}^{k_j-1} \pi_j[a_r] b_j^{-r-1}.$$

The permutations π_j can be deterministic or random. One (deterministic) possibility implemented here is to use the Faure permutation of $\{0, \ldots, b_j\}$ for π_j, for each coordinate $j > 0$.

```
package umontreal.iro.lecuyer.hups;

public class HammersleyPointSet extends PointSet
```

Constructor

```
public HammersleyPointSet (int n, int dim)
```

Constructs a new Hammersley point set with `n` points in `dim` dimensions.

Methods

`public void addFaurePermutations()`

Permutes the digits using Faure permutations for all coordinates. After the method is called, the coordinates $u_{i,j}$ are generated via

$$u_{i,j} = \sum_{r=0}^{k-1} \pi_j[a_r] b_j^{-r-1},$$

for $j = 1, \ldots, s-1$ and $u_{i,0} = i/n$, where π_j is the Faure permutation of $\{0, \ldots, b_j - 1\}$.

`public void ErasePermutations()`

Erases the Faure permutations: from now on, the digits will not be Faure permuted.

HaltonSequence

This class implements the sequence of Halton [10], which is essentially a modification of Hammersley nets for producing an infinite sequence of points having low discrepancy. The ith point in s dimensions is

$$\boldsymbol{u}_i = (\psi_{b_1}(i), \psi_{b_2}(i), \ldots, \psi_{b_s}(i)), \tag{22}$$

for $i = 0, 1, 2, \ldots$, where ψ_b is the radical inverse function in base b, defined in class `RadicalInverse`, and where $2 = b_1 < \cdots < b_s$ are the s smallest prime numbers in increasing order.

A fast method is implemented to generate randomized Halton sequences [37, 40], starting from an arbitrary points x_0.

The points can be "scrambled" by applying a permutation to the digits of i before computing each coordinate via (4), in the same way as for the class `HammersleyPointSet`, for all coordinates $j \geq 0$.

```
package umontreal.iro.lecuyer.hups;

public class HaltonSequence extends PointSet
```

Constructor

`public HaltonSequence (int dim)`

Constructs a new Halton sequence in `dim` dimensions.

Methods

`public void init (double[] x0)`

Initializes the Halton sequence starting at point `x0`. The dimension of `x0` must be at least as large as the dimension of this object.

`public void addFaurePermutations()`

Permutes the digits using Faure permutations for all coordinates. After the method is called, the coordinates $u_{i,j}$ are generated via

$$u_{i,j} = \sum_{r=0}^{k-1} \pi_j[a_r] b_j^{-r-1},$$

for $j = 0, \ldots, s-1$, where π_j is the Faure permutation of $\{0, \ldots, b_j - 1\}$.

`public void ErasePermutations()`

Erases the Faure permutations: from now on, the digits will not be Faure permuted.

Rank1Lattice

This class implements point sets defined by integration lattices of rank 1, defined as follows [34]. One selects an arbitrary positive integer n and a s-dimensional integer vector $(a_0, \ldots, a_{s-1})$, where $0 \leq a_j < n$ for each j. Usually, $a_0 = 1$. The points are defined by

$$\mathbf{u}_i = (i/n)(a_0, a_1, \ldots, a_{s-1}) \bmod 1 \tag{23}$$

for $i = 0, \ldots, n-1$. These n points are distinct provided that n and the a_j's have no common factor.

```
package umontreal.iro.lecuyer.hups;

public class Rank1Lattice extends PointSet
```

Constructor

`public Rank1Lattice (int n, int[] a, int s)`

Instantiates a `Rank1Lattice` with n points and lattice vector a of dimension s.

`public void addRandomShift (int d1, int d2, RandomStream stream)`

Adds a random shift to all the points of the point set, using stream `stream` to generate the random numbers. For each coordinate j from `d1` to `d2-1`, the shift d_j is generated uniformly over $[0, 1)$ and added modulo 1 to all the coordinates of all the points.

`public void clearRandomShift()`

Clears the random shift.

KorobovLattice

This class implements *Korobov lattices*, which are the same point sets as in class `LCGPointSet`, but implemented differently. The parameters are the modulus n and the multiplier a, for arbitrary integers $1 < a < n$. The number of points is n, their dimension is s, and they are defined by

$$\mathbf{u}_i = (i/n)(1, a, a^2, \ldots, a^{s-1}) \bmod 1$$

for $i = 0, \ldots, n-1$.

It is also possible to build a "shifted" Korobov lattice with the first t coordinates rejected. The s-dimensionnal points are then defined as

$$\mathbf{u}_i = (i/n)(a^t, a^{t+1}, a^{t+2}, \ldots, a^{t+s-1}) \bmod 1$$

for $i = 0, \ldots, n-1$ and fixed t.

```
package umontreal.iro.lecuyer.hups;

public class KorobovLattice extends Rank1Lattice
```

Constructors

`public KorobovLattice (int n, int a, int s)`

Instantiates a Korobov lattice point set with modulus n and multiplier a in dimension s.

`public KorobovLattice (int n, int a, int s, int t)`

Instantiates a shifted Korobov lattice point set with modulus n and multiplier a in dimension s. The first t coordinates of a standard Korobov lattice are dropped as described above. The case $t = 0$ corresponds to the standard Korobov lattice.

KorobovLatticeSequence

[7]

This class implements Korobov lattice sequences, defined as follows. One selects a *basis* b and a (large) multiplier a. For each integer $k \geq 0$, we may consider the n-point Korobov lattice with modulus $n = b^k$ and multiplier $\tilde{a} = a \bmod n$. Its points have the form

$$\mathbf{u}_i = (a^i(1, a, a^2, \ldots) \bmod n)/n = (\tilde{a}^i(1, \tilde{a}, \tilde{a}^2, \ldots) \bmod n)/n \tag{24}$$

for $i = 0, \ldots, n-1$. For $k = 0, 1, \ldots$, we have an increasing sequence of lattices contained in one another.

These embedded lattices contain an infinite sequence of points that can be enumerated as follows [14]:

$$\mathbf{u}_i = \psi_b(i)\left(1, a, a^2, \ldots\right) \bmod 1. \tag{25}$$

where $\psi_b(i)$ is the radical inverse function in base b, defined in `RadicalInverse`. The first $n = b^k$ points in this sequence are exactly the same as the n points in (24), for each $k \geq 0$.

```
package umontreal.iro.lecuyer.hups;

public class KorobovLatticeSequence extends KorobovLattice
```

Constructor

```
public KorobovLatticeSequence (int b, int a)
```

Constructs a new lattice sequence with base `b` and `generator` $= a$.

[7] From Pierre: This class is not yet fully implemented

References

[1] I. A. Antonov and V. M. Saleev. An economic method of computing LP_τ-sequences. *Zh. Vychisl. Mat. i. Mat. Fiz.*, 19:243–245, 1979. In Russian.

[2] P. Bratley and B. L. Fox. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software*, 14(1):88–100, 1988.

[3] P. Bratley, B. L. Fox, and H. Niederreiter. Implementation and tests of low-discrepancy sequences. *ACM Transactions on Modeling and Computer Simulation*, 2:195–213, 1992.

[4] R. Cranley and T. N. L. Patterson. Randomization of number theoretic methods for multiple integration. *SIAM Journal on Numerical Analysis*, 13(6):904–914, 1976.

[5] H. Faure. Good permutations for extreme discrepancy. *Journal of Number Theory*, 42:47–56, 1992.

[6] H. Faure and S. Tezuka. Another random scrambling of digital (t, s)-sequences. In K.-T. Fang, F. J. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 242–256, Berlin, 2002. Springer-Verlag.

[7] B. L. Fox. Implementation and relative efficiency of quasirandom sequence generators. *ACM Transactions on Mathematical Software*, 12:362–376, 1986.

[8] B. L. Fox. *Strategies for Quasi-Monte Carlo*. Kluwer Academic, Boston, MA, 1999.

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

[10] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.

[11] J. M. Hammersley. Monte Carlo methods for solving multivariate problems. *Annals of the New York Academy of Science*, 86:844–874, 1960.

[12] P. Hellekalek and H. Niederreiter. The weighted spectral test: Diaphony. *ACM Transactions on Modeling and Computer Simulation*, 8(1):43–60, 1998.

[13] F. J. Hickernell. Obtaining $O(N^{-2+\epsilon})$ convergence for lattice quadrature rules. In K.-T. Fang, F. J. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 274–289, Berlin, 2002. Springer-Verlag.

[14] F. J. Hickernell, H. S. Hong, P. L'Ecuyer, and C. Lemieux. Extensible lattice sequences for quasi-Monte Carlo quadrature. *SIAM Journal on Scientific Computing*, 22(3):1117–1138, 2001.

[15] H. S. Hong and F. H. Hickernell. Algorithm 823: Implementing scrambled digital sequences. *ACM Transactions on Mathematical Software*, 29:95–109, 2003.

[16] N. M. Korobov. The approximate computation of multiple integrals. *Dokl. Akad. Nauk SSSR*, 124:1207–1210, 1959. in Russian.

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

[18] P. L'Ecuyer. Quasi-Monte Carlo methods for simulation. In *Proceedings of the 2003 Winter Simulation Conference*, pages 81–90, Piscataway, NJ, 2003. IEEE Press.

[19] P. L'Ecuyer and C. Lemieux. Quasi-Monte Carlo via linear shift-register sequences. In *Proceedings of the 1999 Winter Simulation Conference*, pages 632–639. IEEE Press, 1999.

[20] P. L'Ecuyer and C. Lemieux. Variance reduction via lattice rules. *Management Science*, 46(9):1214–1235, 2000.

[21] P. L'Ecuyer and C. Lemieux. Recent advances in randomized quasi-Monte Carlo methods. In M. Dror, P. L'Ecuyer, and F. Szidarovszky, editors, *Modeling Uncertainty: An Examination of Stochastic Theory, Methods, and Applications*, pages 419–474. Kluwer Academic, Boston, 2002.

[22] P. L'Ecuyer and F. Panneton. Construction of equidistributed generators based on linear recurrences modulo 2. In K.-T. Fang, F. J. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 318–330. Springer-Verlag, Berlin, 2002.

[23] C. Lemieux, M. Cieslak, and K. Luttmer. *RandQMC User's Guide: A Package for Randomized Quasi-Monte Carlo Methods in C*, 2004. Software user's guide, available at `http://www.math.uwaterloo.ca/~lemieux/randqmc.html`.

[24] C. Lemieux and P. L'Ecuyer. Randomized polynomial lattice rules for multivariate integration and simulation. *SIAM Journal on Scientific Computing*, 24(5):1768–1789, 2003.

[25] J. Matoušek. *Geometric Discrepancy: An Illustrated Guide*. Springer-Verlag, Berlin, 1999.

[26] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, PA, 1992.

[27] H. Niederreiter and C. Xing. Nets, (t, s)-sequences, and algebraic geometry. In P. Hellekalek and G. Larcher, editors, *Random and Quasi-Random Point Sets*, volume 138 of *Lecture Notes in Statistics*, pages 267–302. Springer, New York, NY, 1998.

[28] A. B. Owen. Latin supercube sampling for very high-dimensional simulations. *ACM Transactions on Modeling and Computer Simulation*, 8(1):71–102, 1998.

[29] A. B. Owen. Variance with alternative scramblings of digital nets. *ACM Transactions on Modeling and Computer Simulation*, 13(4):363–378, 2003.

[30] 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.

[31] 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.

[32] F. Panneton and P. L'Ecuyer. Infinite-dimensional point sets based on linear recurrences over GF(2^w). In H. Niederreiter and D. Talay, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2004*, Berlin, 2005. Springer-Verlag. to appear.

[33] G. Pirsic. A software implementation of Niederreiter-Xing sequences. In K.-T. Fang, F. J. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*. Springer, 2001. To appear.

[34] I. H. Sloan and S. Joe. *Lattice Methods for Multiple Integration.* Clarendon Press, Oxford, 1994.

[35] I. M. Sobol'. The distribution of points in a cube and the approximate evaluation of integrals. *U.S.S.R. Comput. Math. and Math. Phys.*, 7:86–112, 1967.

[36] I. M. Sobol'. Uniformly distributed sequences with an additional uniform property. *USSR Comput. Math. Math. Phys. Academy of Sciences*, 16:236–242, 1976.

[37] J. Struckmeier. Fast generation of low-discrepancy sequences. *Journal of Computational and Applied Mathematics*, 61:29–41, 1995.

[38] S. Tezuka. *Uniform Random Numbers: Theory and Practice.* Kluwer Academic Publishers, Norwell, MA, 1995.

[39] S. Tezuka and H. Faure. i-binomial scrambling of digital nets and sequences. Technical report, IBM Research, Tokyo Research Laboratory, 2002.

[40] X. Wang and F. J. Hickernell. Randomized Halton sequences. *Mathematical and Computer Modelling*, 32:887–899, 2000.