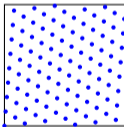


SSJ Tools for Random Numbers and Quasi-Random Numbers

Pierre L'Ecuyer
Université de Montréal, Canada



seed \mathbf{x}_0 ,
transition $\mathbf{x}_n = f(\mathbf{x}_{n-1})$,
output $u_n = g(\mathbf{x}_n)$



ETICS 2024, Saissac

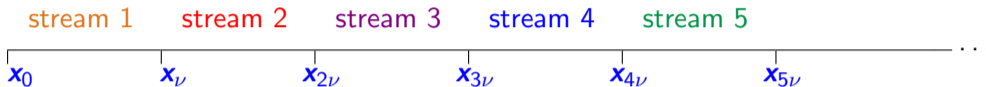
Random number generators (RNGs) with multiple streams

- ▶ For simulation, a single RNG object that produces all the required random numbers in a **single sequence** is inadequate.
- ▶ We want the ability to create objects that produce **independent streams** of random numbers and act as virtual independent RNGs. We may need just a few, or perhaps millions of them.
- ▶ It could be to run simulations on **parallel processors**. Using a single stream of random numbers brings too much overhead in that setting, and the results are also non reproducible.
- ▶ Multiple streams are also useful to compare systems with well synchronized **common random numbers** (CRNs), even on a single processor.

The design described here was implemented in SSJ (in Java) about 25 years ago (1999). It was introduced earlier for RNGs in **SIMOD** (a Modula-2 library) (L and Giroux 1987).

Breaking a single recurrence-based RNG into multiple streams

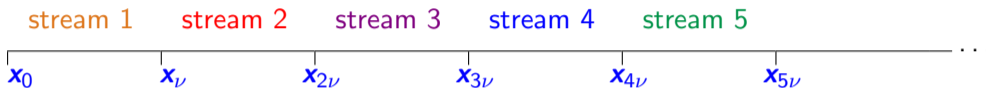
Partition the sequence of a long-period RNG into disjoint streams (segments) of length ν :



where $x_n =$ state of the RNG at step n .

Breaking a single recurrence-based RNG into multiple streams

Partition the sequence of a long-period RNG into disjoint streams (segments) of length ν :



where $x_n = \text{state}$ of the RNG at step n .

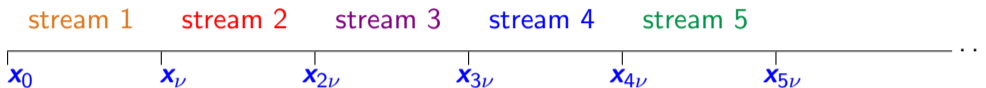
Jumping ahead by ν steps is easy when f is linear; if x_n is a vector and \mathbf{A} a matrix:

$$\begin{aligned} x_{n+1} &= f(x_n) = \mathbf{A}x_n \bmod m \\ x_{n+\nu} &= (\mathbf{A}^\nu \bmod m)x_n \bmod m \end{aligned}$$

with the matrix $(\mathbf{A}^\nu \bmod m)$ precomputed once for all.

Breaking a single recurrence-based RNG into multiple streams

Partition the sequence of a long-period RNG into disjoint streams (segments) of length ν :



where $x_n =$ state of the RNG at step n .

Jumping ahead by ν steps is easy when f is linear; if x_n is a vector and \mathbf{A} a matrix:

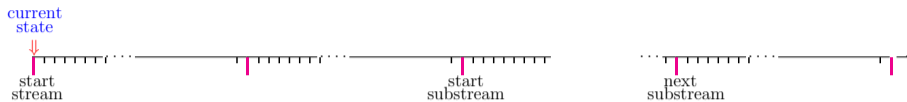
$$\begin{aligned} x_{n+1} &= f(x_n) = \mathbf{A}x_n \bmod m \\ x_{n+\nu} &= (\mathbf{A}^\nu \bmod m)x_n \bmod m \end{aligned}$$

with the matrix $(\mathbf{A}^\nu \bmod m)$ precomputed once for all.

Alternative: pick the starting points of streams at random. Possibility of overlap.

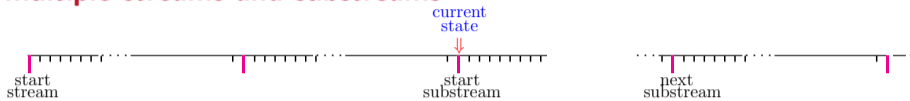
RNGs with multiple streams and substreams

One stream:



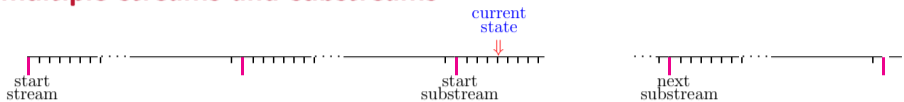
RNGs with multiple streams and substreams

One stream:



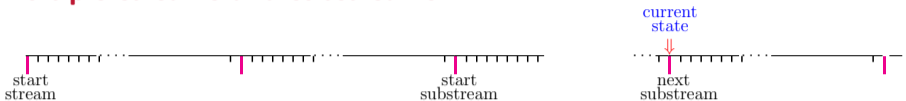
RNGs with multiple streams and substreams

One stream:



RNGs with multiple streams and substreams

One stream:



```
RngStream stream1 = createStream();
```

```
double u = randU01(stream1);    int i = randInt(stream1, 1, 6);
ResetStartSubstream(stream1);  ResetNextSubstream(stream1);
ResetStartStream(stream1);
```

RngStreams is a specific implementation introduced 25 years ago (L et al. 2000). The multiple streams are further partitioned in **substreams** (which are **not objects**).

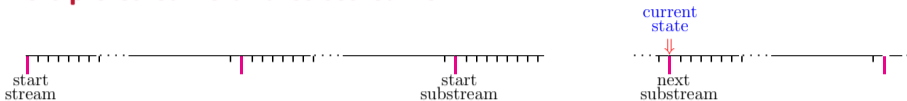
RngStreams is based on the **MRG32k3a** generator, with period $\approx 2^{191}$.

Streams start $\nu = 2^{127}$ values apart and substreams have length $\delta = 2^{76}$.

It is available in C, C++, FORTRAN, Java, Python, Julia, R, Matlab, Cuda, etc.

RNGs with multiple streams and substreams

One stream:



```
RngStream stream1 = createStream();
```

```
double u = randU01(stream1);      int i = randInt(stream1, 1, 6);
ResetStartSubstream(stream1);    ResetNextSubstream(stream1);
ResetStartStream(stream1);
```

RngStreams is a specific implementation introduced 25 years ago (L et al. 2000). The multiple streams are further partitioned in **substreams** (which are **not objects**).

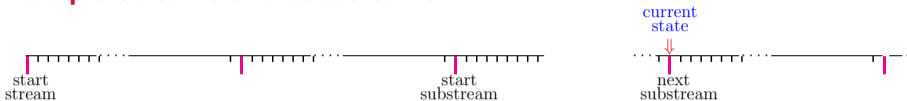
RngStreams is based on the **MRG32k3a** generator, with period $\approx 2^{191}$.

Streams start $\nu = 2^{127}$ values apart and substreams have length $\delta = 2^{76}$.

It is available in C, C++, FORTRAN, Java, Python, Julia, R, Matlab, Cuda, etc.

RNGs with multiple streams and substreams

One stream:



```
RngStream stream1 = createStream();
```

```
double u = randU01(stream1);      int i = randInt(stream1, 1, 6);
ResetStartSubstream(stream1);    ResetNextSubstream(stream1);
ResetStartStream(stream1);
```

RngStreams is a specific implementation introduced 25 years ago (L et al. 2000). The multiple streams are further partitioned in **substreams** (which are **not objects**).

RngStreams is based on the **MRG32k3a** generator, with period $\approx 2^{191}$.

Streams start $\nu = 2^{127}$ values apart and substreams have length $\delta = 2^{76}$.

It is available in C, C++, FORTRAN, Java, Python, Julia, R, Matlab, Cuda, etc.

Stochastic Simulation in Java: SSJ Library

Offers different types of `RandomStream` objects. Each type must implement at least:

```
public interface RandomStream {
```

```
    public void resetStartStream();
```

Resets the stream to its **initial state**.

```
    public void resetStartSubstream();
```

Resets the stream to the beginning of its **current substream**.

```
    public void resetNextSubstream();
```

Resets the stream to the beginning of its **next substream**.

```
    public double nextDouble();
```

Returns a **uniform** random number over $(0, 1)$ and moves ahead by one step.

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

Returns a uniform **random integer** over $\{i, i + 1, \dots, j\}$ and moves ahead.

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

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

```
}
```

```
public class MRG32k3a implements RandomStream
```

One particular implementation, based on the 32-bit MRG32k3a generator.

```
public MRG32k3a(); // Constructs and returns a new stream object.
```

```
public class LFSR113 implements RandomStream
```

Another (faster) 32-bit implementation, based on a combined LFSR with period near 2^{113} .

```
public LFSR113(); // Constructs and returns a new stream object.
```

```
public class LFSR258 implements RandomStream
```

A 64-bit implementation, based on a combined LFSR with period near 2^{258} .

```
public LFSR258(); // Constructs and returns a new stream object.
```

```
RandomStream stream1 = new LFSR113();
```

```
RandomStream stream2 = new LFSR113();
```

```
RandomStream stream3 = new LFSR258();
```

```
double u = stream2.nextDouble(); // Uniform over (0,1)
```

```
int i = stream1.nextInt (1, 6); // Uniform over {1,2,3,4,5,6} (a die)
```

```
int j = stream2.nextInt (1, 6);
```

```
int k = stream2.nextInt (0, 1); // Uniform over {0,1} (a coin)
```

Non-uniform random variates in SSJ

Probability distributions are represented by `Distribution` types of objects, classified in subtypes `DiscreteDistribution`, `ContinuousDistribution`, `ContinuousDistributionMulti` (multivariate), `StochasticProcess`, etc. Each subtype has several subclasses (Poisson, binomial, normal, Student, gamma, ...).

Non-uniform random variates in SSJ

Probability distributions are represented by `Distribution` types of objects, classified in subtypes `DiscreteDistribution`, `ContinuousDistribution`, `ContinuousDistributionMulti` (multivariate), `StochasticProcess`, etc. Each subtype has several subclasses (Poisson, binomial, normal, Student, gamma, ...).

Non-uniform random variate generators are objects of type `RandomVariateGen`. The most general way to define one is to match a `Distribution` object with a `RandomStream` object. See the Online Doc. By default, the generator will use *inversion*: $X = F^{-1}(U)$.

```
public class RandomVariateGen
```

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

Creates a generator for distribution `dist` with stream `s`.

```
    public double nextDouble()
```

Generates a new variate from distribution `dist` with this generator.

We have specialized subclasses of `RandomVariateGen`, for many distributions. Some have `static methods`, in case we do not want to create an object.

```
public class NormalGen extends RandomVariateGen
```

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

Creates a normal generator.

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

Generates a new normal variate.

We have specialized subclasses of `RandomVariateGen`, for many distributions. Some have `static methods`, in case we do not want to create an object.

```
public class NormalGen extends RandomVariateGen
```

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

Creates a normal generator.

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

Generates a new normal variate.

Several ways of creating (or not) a normal or Poisson generator:

```
RandomVariateGen ng1 = new RandomVariateGen (stream3, new NormalDist (mu, sigma));
RandomVariateGen ng2 = new NormalGen (stream3, mu, sigma); // Accurate inversion
RandomVariateGen ng3 = new NormalACRGen (stream3, mu, sigma); // Fast rejection method
RandomVariateGen ng4 = new NormalBoxMullerGen (stream3, mu, sigma);
```

```
double x = ng1.nextDouble(); // Use a generator object
double x = NormalGen.nextDouble (stream3, mu, sigma); // Static method, no object
```

```
RandomVariateGenInt pg1 = new PoissonGen (stream1, new PoissonDist (lambda));
RandomVariateGenInt pg2 = new PoissonGen (stream1, lambda); // Precomputes tables
```

```
int n = pg1.nextInt(); // Uses the pg1 object (with precomputed tables)
int n = PoissonGen.nextInt(stream1, lambda); // Static method, no object
```

Comparing systems with common random numbers: a simple inventory example

X_j = inventory level in morning of day j ;

D_j = demand (random) on day j , uniform over $\{0, 1, \dots, L\}$;

$\min(D_j, X_j)$ sales on day j ;

$Y_j = \max(0, X_j - D_j)$ inventory at end of day j ;

Orders follow a (s, S) policy: If $Y_j < s$, order $S - Y_j$ items.

Each order arrives (random) for next morning with probability p .

Revenue for day j : sales – inventory costs – order costs
 $= c \cdot \min(D_j, X_j) - h \cdot Y_j - (K + k \cdot (S - Y_j)) \cdot \mathbb{I}[\text{an order arrives}]$.

We want to compare several policies (s, S) , using the same random numbers for the same purpose for all policies. But the number of calls to RNG for order arrivals is random! To have “same purpose”, we use two streams of random numbers, and one substream per run.

Inventory example: Java code to simulate m days with two streams

```
// Simulates inventory model for m days, with the (s,S) policy.
double simulateOneRun (int m, int s, int S,
    RandomStream streamDemand, RandomStream streamOrderArrive) {
    int Xj = S, Yj;          // Stock Xj in morning and Yj in evening.
    double profit = 0.0;    // Cumulated profit.
    for (int j = 0; j < m; j++) {
        // Generate and subtract the demand for the day.
        Yj = Xj - streamDemand.nextInt (0, L);
        if (Yj < 0) Yj = 0; // Lost demand.
        profit += c * (Xj - Yj) - h * Yj;
        if ((Yj < s) && (streamOrderArrive.nextDouble() < p)) {
            // The order has arrived, we pay for it.
            profit -= K + k * (S - Yj);
            Xj = S;
        } else
            Xj = Yj;        // Order not received.
    }
    return profit / m;     // Average profit per day.
}
```

Comparing p policies with CRNs (using a single processor)

```
// Simulate n runs with CRNs for p policies (s[k], S[k]), k=0,...,p-1.
RandomStream streamDemand = new MRG32k3a();
RandomStream streamOrderArrive = new MRG32k3a();
for (int k = 0; k < p; k++) { // for each policy
    for (int i = 0; i < n; i++) { // perform n runs
        statProfit[k, i] = simulateOneRun (m, s[k], S[k], streamDemand, streamOrderArrive);
        // Realign starting points so they are the same for all policies
        streamDemand.resetNextSubstream();
        streamOrderArrive.resetNextSubstream();
    }
    streamDemand.resetStartStream();
    streamOrderArrive.resetStartStream();
}

// Print and plot results ...
...
```

Comparing p policies with CRNs (using a single processor)

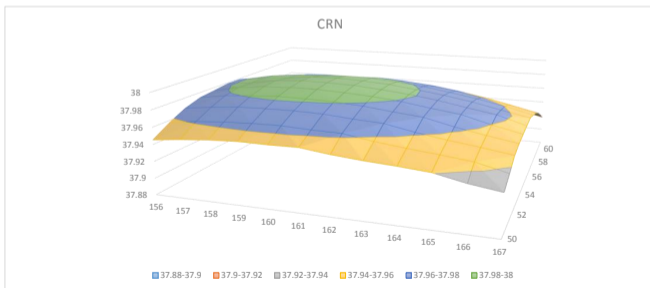
```
// Simulate n runs with CRNs for p policies (s[k], S[k]), k=0,...,p-1.
RandomStream streamDemand = new MRG32k3a();
RandomStream streamOrderArrive = new MRG32k3a();
for (int k = 0; k < p; k++) { // for each policy
    for (int i = 0; i < n; i++) { // perform n runs
        statProfit[k, i] = simulateOneRun (m, s[k], S[k], streamDemand, streamOrderArrive);
        // Realign starting points so they are the same for all policies
        streamDemand.resetNextSubstream();
        streamOrderArrive.resetNextSubstream();
    }
    streamDemand.resetStartStream();
    streamOrderArrive.resetStartStream();
}

// Print and plot results ...
...
```

Only two streams suffice for the entire simulation experiment. If we use different streams for the n different runs, we would need $2n$ stream objects instead. Would be less efficient.

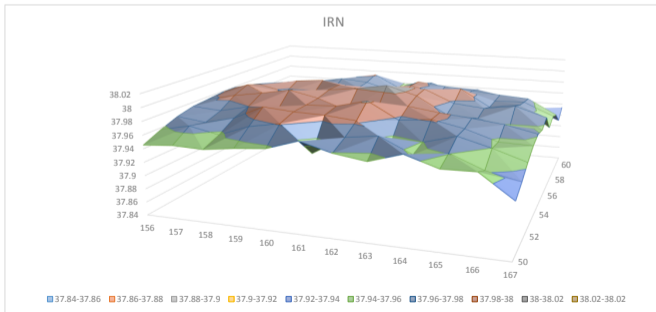
Comparison with common random numbers

	156	157	158	159	160	161	162	163	164	165	166	167
50	37.94537	37.94888	37.95166	37.95319	37.95274	37.95318	37.94887	37.94584	37.94361	37.94074	37.93335	37.92832
51	37.9574	37.96169	37.96379	37.96524	37.96546	37.96379	37.96293	37.95726	37.95295	37.94944	37.94536	37.93685
52	37.96725	37.97117	37.97402	37.97476	37.97492	37.97387	37.971	37.96879	37.96184	37.95627	37.95154	37.94626
53	37.97356	37.97852	37.98098	37.98243	37.98187	37.98079	37.97848	37.97436	37.97088	37.96268	37.95589	37.94995
54	37.97593	37.98241	37.98589	37.98692	37.98703	37.98522	37.9829	37.97931	37.97397	37.96925	37.95986	37.95186
55	37.97865	37.98235	37.9874	37.9894	37.98909	37.9879	37.98483	37.98125	37.97641	37.96992	37.96401	37.95343
56	37.97871	37.98269	37.98494	37.98857	37.98917	37.98757	37.98507	37.98073	37.97594	37.96989	37.96227	37.95519
57	37.97414	37.98035	37.98293	37.98377	37.98603	37.98528	37.98239	37.97858	37.97299	37.96703	37.95981	37.95107
58	37.96869	37.97207	37.97825	37.97944	37.97895	37.97987	37.97776	37.97358	37.96848	37.9617	37.95461	37.94622
59	37.95772	37.96302	37.9663	37.97245	37.97234	37.97055	37.9701	37.96664	37.96122	37.95487	37.94695	37.93871
60	37.94434	37.94861	37.95371	37.95691	37.96309	37.96167	37.9586	37.95678	37.95202	37.9454	37.93785	37.92875
61	37.922	37.93169	37.93591	37.94085	37.94401	37.95021	37.94751	37.94312	37.94	37.93398	37.92621	37.91742



Comparison with independent random numbers

	156	157	158	159	160	161	162	163	164	165	166	167
50	37.94537	37.94888	37.94736	37.95314	37.95718	37.97194	37.95955	37.95281	37.96711	37.95221	37.95325	37.92063
51	37.9574	37.9665	37.95732	37.97337	37.98137	37.94273	37.96965	37.97573	37.95425	37.96074	37.94185	37.93139
52	37.96725	37.96166	37.97192	37.99236	37.98856	37.98708	37.98266	37.94671	37.95961	37.97238	37.95982	37.94465
53	37.97356	37.96999	37.97977	37.97611	37.98929	37.99089	38.00219	37.97693	37.98191	37.97217	37.95713	37.95575
54	37.97593	37.9852	37.99233	38.00043	37.99056	37.9744	37.98008	37.98817	37.98168	37.97703	37.97145	37.96138
55	37.97865	37.9946	37.97297	37.98383	37.99527	38.00068	38.00826	37.99519	37.96897	37.96675	37.9577	37.95672
56	37.97871	37.9867	37.97672	37.9744	37.9955	37.9712	37.96967	37.99717	37.97736	37.97275	37.97968	37.96523
57	37.97414	37.97797	37.98816	37.99192	37.9678	37.98415	37.97774	37.97844	37.99203	37.96531	37.97226	37.93934
58	37.96869	37.97435	37.9625	37.96581	37.97331	37.95655	37.98382	37.97144	37.97409	37.96631	37.96764	37.94759
59	37.95772	37.94725	37.9711	37.97905	37.97504	37.96237	37.98182	37.97656	37.97212	37.96762	37.96429	37.93976
60	37.94434	37.95081	37.94275	37.95515	37.98134	37.95863	37.96581	37.95548	37.96573	37.93949	37.93839	37.9203
61	37.922	37.93006	37.92656	37.93281	37.94999	37.95799	37.96368	37.94849	37.954	37.92439	37.90535	37.93375



Larger and more complicated systems

May require thousands of different streams, even for a simulation on a single CPU.

Substreams can be used for the independent replications, as we saw. Very convenient.

We have used that successfully for simulation and optimization of **service systems** such as call centers, **reliability** models, and **financial** contracts and systems.

One may also think of factories, transportation networks, logistic systems, supply chains, etc.

Multiple streams for parallel processors

This would be a separate talk. Some references at the end.

Highly-uniform point sets (LHS, QMC, RQMC, ...)

See the `hups` package in SSJ. Some of the object types:

```
abstract class PointSet // A set of  $n$  points in  $[0,1]^s$ 
    int getDimension() // Dimension  $s$ 
    int getNumPoints() // Number  $n$  of points
    double getCoordinate(int i, int j) // Coordinate  $j$  of point  $i$ 
    PointSetIterator iterator() // To enumerate the points
    :
```

```
abstract class PointSetIterator implements RandomStream
```

This is a `RandomStream` for which each substream represents one point and `nextDouble()` computes and returns the next coordinate of the current point.

The points are usually (re)computed on the fly, not stored.

```
class PointSetRandomization // A randomization that can be applied to a point set
    void randomize(PointSet p) // Generates a fresh randomization
```

```
RQMCPointSet(PointSet p, PointSetRandomization rand)
```

A point set matched with a randomization.

More point sets (partial tree)

DigitalNet

 DigitalNetBase2

 DigitalSequenceBase2

 SobolSequence

 NiedXingSequenceBase2

Rank1Lattice

 KorobovLattice

HaltonSequence

CycleBasedPointSet

 LCGPointSet

 LFSRPointSet

 CycleBasedPointSetBase2

ContainerPointSet

 BakerTransformedPointSet

 AntitheticPointSet

CachedPointSet

 LatinHypercube

 StratifiedUnitCube

 IndependentCachedPoints Same as Monte Carlo

Each type has a specialized method to construct one (or more) iterator(s) that acts as a `RandomStream` to generate the points.

Randomizations

RandomShift // Random shift mod 1 or random digital shift

NestedUniformScrambling // Owen's scrambling

LeftMatrixScrambling // Matoušek linear scrambling

LMSScrambleShift // LMS + random digital shift

Etc.

Fast generation of the points, examples

Rank-1 lattice with generating vector $\mathbf{a} = (a_1, \dots, a_s)$, n points.

$$\mathbf{u}_0 = \mathbf{0}; \quad \mathbf{u}_i = (\mathbf{u}_{i-1} + \mathbf{a}/n) \bmod 1; \quad // \text{ the factor } 1/n \text{ is precomputed}$$

Fast generation of the points, examples

Rank-1 lattice with generating vector $\mathbf{a} = (a_1, \dots, a_s)$, n points.

$$\mathbf{u}_0 = \mathbf{0}; \quad \mathbf{u}_i = (\mathbf{u}_{i-1} + \mathbf{a}/n) \bmod 1; \quad // \text{ the factor } 1/n \text{ is precomputed}$$

Korobov lattice: $\mathbf{a} = (1, a, \dots, a^{s-1} \bmod n)$.

$$\mathbf{u}_i = (u_{i,1}, \dots, u_{i,s-1}, u_{i,s}) = (u_{i-1,2}, \dots, u_{i-1,s}, au_{i-1,s} \bmod 1);$$

Fast generation of the points, examples

Rank-1 lattice with generating vector $\mathbf{a} = (a_1, \dots, a_s)$, n points.

$$\mathbf{u}_0 = \mathbf{0}; \quad \mathbf{u}_i = (\mathbf{u}_{i-1} + \mathbf{a}/n) \bmod 1; \quad // \text{ the factor } 1/n \text{ is precomputed}$$

Korobov lattice: $\mathbf{a} = (1, a, \dots, a^{s-1} \bmod n)$.

$$\mathbf{u}_i = (u_{i,1}, \dots, u_{i,s-1}, u_{i,s}) = (u_{i-1,2}, \dots, u_{i-1,s}, au_{i-1,s} \bmod 1);$$

Digital net in base 2, $n = 2^k$ points with w bits of accuracy.

The k columns of generating matrix \mathbf{C}_j represented by k w -bit integers $C_{j,1}, \dots, C_{j,k}$.

To compute coordinate j of point i :

$$x = 0; \quad // \text{ integer}$$

for $c = 0$ to $k - 1$

$$x = x \oplus ((i \gg c) \& 1) * C_{j,k};$$

$$u_{i,j} = x * 2^{-w}. \quad // \text{ the factor } 2^{-w} \text{ is precomputed.}$$

Fast generation of the points, examples

Rank-1 lattice with generating vector $\mathbf{a} = (a_1, \dots, a_s)$, n points.

$$\mathbf{u}_0 = \mathbf{0}; \quad \mathbf{u}_i = (\mathbf{u}_{i-1} + \mathbf{a}/n) \bmod 1; \quad // \text{ the factor } 1/n \text{ is precomputed}$$

Korobov lattice: $\mathbf{a} = (1, a, \dots, a^{s-1} \bmod n)$.

$$\mathbf{u}_i = (u_{i,1}, \dots, u_{i,s-1}, u_{i,s}) = (u_{i-1,2}, \dots, u_{i-1,s}, au_{i-1,s} \bmod 1);$$

Digital net in base 2, $n = 2^k$ points with w bits of accuracy.

The k columns of generating matrix \mathbf{C}_j represented by k w -bit integers $C_{j,1}, \dots, C_{j,k}$.

To compute coordinate j of point i :

$$x = 0; \quad // \text{ integer}$$

for $c = 0$ to $k - 1$

$$x = x \oplus ((i \gg c) \& 1) * C_{j,k};$$

$$u_{i,j} = x * 2^{-w}. \quad // \text{ the factor } 2^{-w} \text{ is precomputed.}$$

Random shift $\mathbf{U} = (U_1, \dots, U_s) \in (0, 1)^s$: // we store this vector \mathbf{U} .

$$u_{i,j} = (u_{i,j} + U_j) \bmod 1 \text{ (lattice)} \quad \text{or} \quad u_{i,j} = u_{i,j} \oplus U_j \text{ (digital net base 2).}$$

Basic tools for MC and RQMC experiments

```
interface MonteCarloModel<OutType> // Interface for a simple simulation model
    void simulate(RandomStream stream) // Simulate the model once
    OutType getPerformance() // Returns the simulation output (e.g., a double)
```

```
static class MonteCarloExperiment // Methods to perform simulation experiments
    static void simulateRuns (MonteCarloModel model, int n, RandomStream stream,
                             Tally statValue)
```

Performs n simulation runs, returns results in `statValue`.

```
static class RQMCExperiment // Methods to perform RQMC experiments
    static void simulReplicatesRQMC (MonteCarloModel model, PointSet p,
                                     PointSetRandomization rand, int m, Tally statReps)
```

Performs m RQMC replicates with given model and point set, put results in `statReps`.


```
// Performs n simulation runs
simulateRuns (MonteCarloModel model, int n, RandomStream stream, Tally statValue)
    statValue.init();
    for (int i = 0; i < n; i++)
        model.simulate(stream); // Simulate the model with this stream
        statValue.add(model.getPerformance());
        stream.resetNextSubstream(); // One substream per simulation run

// Performs m RQMC replicates
simulReplicatesRQMC (MonteCarloModel model, PointSet p, PointSetRandomization rand,
                    int m, Tally statReps)
    statReps.init();
    int n = p.getNumPoints();
    Tally statValue = new Tally(); // Internal collector for stats, for each replication
    PointSetIterator stream = p.iterator();
    for (int rep = 0; rep < m; rep++) {
        rand.randomize(p);
        stream.resetStartStream(); // Reset to first point
        simulateRuns(model, n, stream, statValue); // Simulate n runs with RQMC points
        statReps.add(statValue.average()); // For the estimator of the mean
    }
```

Code snippets for an example

```
// A simulation model. Exact same code for MC and RQMC.
public class FinancialOption implements MonteCarloModelDouble
    public void simulate(RandomStream stream) ...
    public double getPerformance() ...
        :
MonteCarloModelDouble model = new FinancialOption ( ... );
RandomStream noise = new LFSR113();
simulateRuns (model, n, stream, statValue); // Monte Carlo, n runs

DigitalNet p1 = new SobolSequence(16, 31, s); //  $n = 2^{16}$  points in s dim
PointSetRandomization rand1 = new LMScrambleShift(noise);
simulReplicatesRQMC (model, p1, rand1, m, statReps); // m RQMC replicates

KorobovLattice pkor = new KorobovLattice(16381, 5693, s); // n = 16381 points in s dim
BakerTransformedPointSet p2 = new BakerTransformedPointSet(pkor);
PointSetRandomization rand2 = new RandomShift(noise);
System.out.println (simulReplicatesRQMCDefaultReport (model, p2, rand2, m, statReps));

// statReps will contain the m independent RQMC estimates.
// The last function also prints statistics and a confidence interval.
```

Conclusion

- ▶ SSJ offers an [integrated framework](#) for Monte Carlo and quasi-Monte Carlo. It has RNGs with multiple streams, many distributions and stochastic processes, and a variety of tools for RQMC. The software runs pretty fast.
- ▶ It was developed about 20 years ago for 32-bit processors and a single CPU. We plan to make a 64-bit version that also better exploits multicore CPUs in the code.
- ▶ Similar frameworks can be done easily in C++, Python, Julia, etc.
- ▶ Similar tools for [highly-parallel computing systems](#) (GPUs, TPUs, etc.) would require a different design in some parts. For example, the way we create the RNG streams in SSJ is inherently sequential (one after the other). On GPUs, we would like to create them in parallel. This is a separate (interesting) topic.

Self-references related to my software and RNG/RQMC work

Most are available at <http://www.iro.umontreal.ca/~lecuyer/papers.html>



L'Ecuyer, P. 2000. "SIMOD-99: Définition fonctionnelle et guide d'utilisation".
<http://www.iro.umontreal.ca/~lecuyer/myftp/papers/guide-simod-2000.pdf>.



L'Ecuyer, P. 2006. "Uniform Random Number Generation". In *Simulation*, Edited by S. G. Henderson and B. L. Nelson, Handbooks in Operations Research and Management Science, 55–81. Amsterdam, The Netherlands: Elsevier. Chapter 3.



L'Ecuyer, P. 2015. "Random Number Generation with Multiple Streams for Sequential and Parallel Computers". In *Proceedings of the 2015 Winter Simulation Conference*, 31–44: IEEE Press.









L'Ecuyer, P. 2017. "History of Uniform Random Number Generation". In *Proceedings of the 2017 Winter Simulation Conference*, 202–230: IEEE Press.



L'Ecuyer, P. 2023. "SSJ: Stochastic Simulation in Java". <https://github.com/umontreal-simul/ssj>.



L'Ecuyer, P., and E. Buist. 2005. "Simulation in Java with SSJ". In *Proceedings of the 2005 Winter Simulation Conference*, 611–620: IEEE Press.

- 
- L'Ecuyer, P., and N. Giroux. 1987. "A process-oriented simulation package based on Modula-2". In *Proceedings of the 1987 Winter Simulation Conference*, 165–174: IEEE Press.
- 
- L'Ecuyer, P., P. Marion, M. Godin, and F. Puchhammer. 2022. "A Tool for Custom Construction of QMC and RQMC Point Sets". In *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC 2020*, Edited by A. Keller, 51–70. Berlin: Springer. <https://arxiv.org/abs/2012.10263>.
- 
- L'Ecuyer, P., D. Munger, and N. Kemerchou. 2015. "cIRNG: A Random Number API with Multiple Streams for OpenCL". <http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clrng-api.pdf>.
- 
- L'Ecuyer, P., D. Munger, B. Oreshkin, and R. Simard. 2017. "Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on GPUs". *Mathematics and Computers in Simulation* 135:3–17.
- 
- L'Ecuyer, P., O. Nadeau-Chamard, Y.-F. Chen, and J. Lebar. 2021. "Multiple Streams with Recurrence-Based, Counter-Based, and Splittable Random Number Generators". In *Proceedings of the 2021 Winter Simulation Conference*, 1–16: IEEE Press.
- 
- L'Ecuyer, P., R. Simard, E. J. Chen, and W. D. Kelton. 2002. "An Object-Oriented Random-Number Package with Many Long Streams and Substreams". *Operations Research* 50 (6): 1073–1075.