Random Numbers and RQMC Points in SSJ

SSJ (Stochastic Simulation in Java) as shown here was designed over 25 years ago.

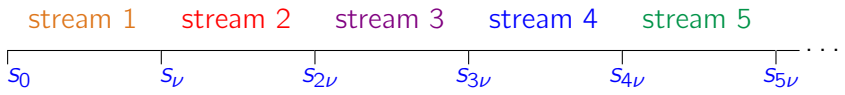
Multiple Streams of Random Numbers in SSJ

A single stream of random numbers is often not sufficient for simulation.
One may need independent streams for example:

1. To run simulations on parallel processors.
2. To compare systems with well synchronized common random numbers (CRNs). Can be complicated to implement and manage when different configurations do not need the same number of U_j's.

From a single RNG, one can create multiple "`random stream`" **objects** that behave as "independent" virtual RNGs (or streams of random numbers).

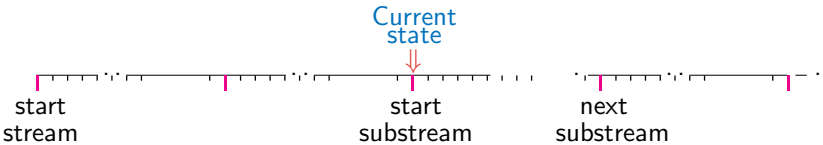
Simple approach: partition the entire sequence into disjoint segments (streams) of length ν.
If s_i is the RNG state at step i, we have:



A `RandomStream` object is essentially a virtual RNG that "produces" (imitates) a sequence of independent random numbers. We can create as many as we want.

In SSJ, each `RandomStream` is also partitioned in substreams (which are not objects).

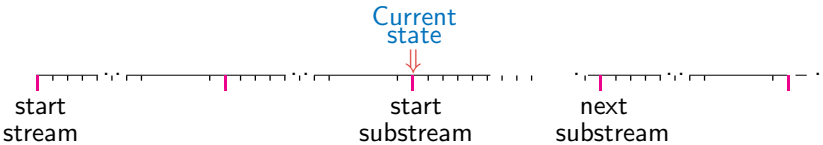
One stream:



A `RandomStream` object is essentially a virtual RNG that "produces" (imitates) a sequence of independent random numbers. We can create as many as we want.

In SSJ, each `RandomStream` is also partitioned in substreams (which are not objects).

One stream:



For example, the MRG32k3a generator has a cycle of length (near) 2^{191} cut in streams that start 2^{127} positions apart. Each stream is also partitioned in 2^{51} substreams of length 2^{76}.

```
RandomStream stream1 = new MRG32k3a();
double u = stream1.nextDouble(); ....

stream1.resetNextSubstream(); ....
stream1.resetStartSubStream();
stream1.resetStartStream();
```

RandomStream Interface in SSJ

```
public interface RandomStream {
```

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

Réinitialise la suite à son état initial.

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

Réinitialise la suite au début de sa sous-suite courante. substream.

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

Réinitialise la suite au début de sa prochaine sous-suite. substream.

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

Retourne une v.a. $U(0,1)$ de cette suite et avance d'un pas.

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

Retourne une v.a. uniforme sur $\{i, i+1, \ldots, j\}$.

```
}
```

```
public class MRG32k3a implements RandomStream {
```

Une implantation particulière, basée sur un générateur de période $\approx 2^{191}$, partitionnée en suites disjointes de longueur 2^{127}, et sous-suites de longueur 2^{76}.

```
  public MRG32k3a();
```

Construit une nouvelle suite (stream).

```
}
```

```
public class LFSR113 implements RandomStream {
```

Une implantation basée sur une combinaison de 4 "LFSR" combinés. de période $\approx 2^{113}$.

```
  public LFSR113();
```

Construit une nouvelle suite (stream).

```
}
```

Générateurs non-uniformes dans SSJ

Lois continues ou discrètes générales. Méthode par défaut: inversion.

```
public class RandomVariateGen
```

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

Crée un générateur pour la loi `dist`, avec la suite `s`.

```
public double nextDouble()
```

Génère une nouvelle valeur, par défaut par inversion.

Générateurs non-uniformes dans SSJ

Lois continues ou discrètes générales. Méthode par défaut: inversion.

```
public class RandomVariateGen
```

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

Crée un générateur pour la loi `dist`, avec la suite `s`.

```
public double nextDouble()
```

Génère une nouvelle valeur, par défaut par inversion.

Il y a aussi des générateurs spécialisés pour plusieurs distributions.

```
public class NormalGen extends RandomVariateGen
```

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

Crée un générateur de v.a. normales.

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

Génère une nouvelle v.a. normale, en utilisant la suite `s`.

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 $\{0, 1, \ldots, L\}$;
$\min(D_j, X_j)$ 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
$= c \cdot \min(D_j, X_j) - h \cdot Y_j - (K + k \cdot (S - Yj)) \cdot \mathbb{I}[\text{an order arrives}]$.

Number of calls to RNG for order arrivals is random!

Two streams of random numbers, one substream for each run.
Same streams and substreams for all policies (s, S).

Inventory example: code to simulate m days with two streams

```
double inventorySimulateOneRun (int m, int s, int S,
       RngStream *stream_demand, RngStream *stream_order) {
   // Simulates inventory model for m days, with the (s,S) policy.
   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 - RandInt (stream_demand, 0, L);
      if (Yj < 0) Yj = 0; // Lost demand.
      profit += c * (Xj - Yj) - h * Yj;
      if ((Yj < s) && (RandU01 (stream_order) < p)) {
         // We have a successful order, we pay for it.
         profit -= K + k * (S - Yj);
         Xj = S;
      } else
         Xj = Yj;          // Order not received.
   }
   return profit / m;      // Return 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.
RngStream* stream_demand = CreateStream();
RngStream* stream_order  = CreateStream();
for (int k = 0; k < p; k++) {   // for each policy
   for (int i = 0; i < n; i++) {    // perform n runs
      stat_profit[k, i] = inventorySimulateOneRun (m, s[k], S[k],
                                     stream_demand, stream_order);
      // Realign starting points so they are the same for all policies
      ResetNextSubstream (stream_demand);
      ResetNextSubstream (stream_order);
   }
   ResetStartStream (stream_demand);
   ResetStartStream (stream_order);
}

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

Comparing p policies with CRNs (using a single processor)

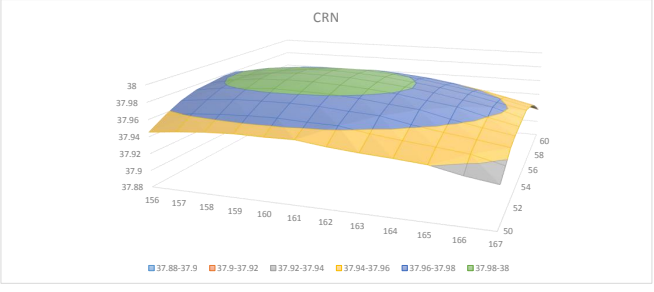
```
// Simulate n runs with CRNs for p policies (s[k], S[k]), k=0,...,p-1.
RngStream* stream_demand = CreateStream();
RngStream* stream_order  = CreateStream();
for (int k = 0; k < p; k++) {   // for each policy
   for (int i = 0; i < n; i++) {    // perform n runs
      stat_profit[k, i] = inventorySimulateOneRun (m, s[k], S[k],
                                    stream_demand, stream_order);
      // Realign starting points so they are the same for all policies
      ResetNextSubstream (stream_demand);
      ResetNextSubstream (stream_order);
   }
   ResetStartStream (stream_demand);
   ResetStartStream (stream_order);
}

// 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.

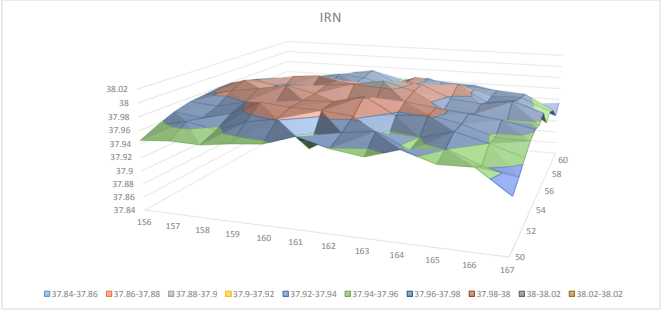
Comparison with common random numbers

	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 just saw. Very convenient.

My students have used that a lot for simulation and optimization of service systems such as call centers, reliability models, and also financial contracts and systems.

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

QMC Point Sets in SSJ: the hups Package

Several types of QMC point sets (`Rank1Lattice`, `DigitalNet`, `DigitalNetBase2`, `SobolSequence`, etc.) are available in SSJ. In general, the parameters should be passed to the constructor. All of them extend the abstract class `PointSet`.

For RQMC, the point set must be randomized by applying a randomization, which is an object of a class that must implement the interface `PointSetRandomization`. `RandomShift` and `LMSScrambleShift` are examples of such classes.

By combining a `PointSet` with a `PointSetRandomization`, we obtain a `RQMCPointSet`. Not all combinations are compatible.

To generate the coordinates of all the points of a point set, we create and use a `PointSetIterator` for that point set. Such iterators actually implement the `RandomStream` interface, so they can be used in the exact same way as a `RandomStream` object. With this, one can replace MC (random numbers) by QMC/RQMC without changing the code of the model.

Small Examples for MC and RQMC

In the tutorial: see `AsianGBM` and `AsianGBMRQMC`.

Then see `AsianGBM2` and `AsianGBMRQMC2`, which implement `MonteCarloModelDouble`.

Small Examples for MC and RQMC

In the tutorial: see `AsianGBM` and `AsianGBMRQMC`.

Then see `AsianGBM2` and `AsianGBMRQMC2`, which implement `MonteCarloModelDouble`.

See also `ProductExpCosRQMC` in package `etics24qmc`.

Simple Scripts for MC/QMC/RQMC Experiments in SSJ

See the package `mcqmctools`.

Interfaces `MonteCarloModel` and `MonteCarloModelDouble`.

Classes `MonteCarloExperiment` and `RQMCExperiment`.

Illustration using `RepsRQMC` in package `etics24QMC`.

RandomStream Interface in SSJ