Paper SAS420-2018

Tips and Techniques for Using the Random-Number Generators in SAS®

Warren Sarle and Rick Wicklin, SAS Institute Inc.

ABSTRACT

SAS® 9.4M5 introduces new random-number generators (RNGs) and new subroutines that enable you to initialize, rewind, and use multiple random-number streams. This paper describes the new RNGs and provides tips and techniques for using random numbers effectively and efficiently in SAS. Applications of these techniques include statistical sampling, data simulation, Monte Carlo estimation, and random numbers for parallel computation.

INTRODUCTION TO RANDOM-NUMBER GENERATORS

Random quantities are the heart of probability and statistics. Modern statistical programmers need to generate random values from a variety of probability distributions and use them for statistical sampling, bootstrap and resampling methods, Monte Carlo estimation, and data simulation. For these techniques to be statistically valid, the values that are produced by statistical software must contain a high degree of randomness.

In this paper, a *random-number generator* (RNG) refers either to a deterministic algorithm that generates pseudorandom numbers or to a hardware-based RNG. When a distinction needs to be made, an algorithm-based method is called a pseudorandom-number generator (PRNG). An RNG generates uniformly distributed numbers in the interval (0, 1).

A pseudorandom-number generator in SAS is initialized by an integer, called the *seed value*, which sets the initial state of the PRNG. Each iteration of the algorithm produces a pseudorandom number and advances the internal state. The infinite sequence of all numbers that can be generated from a particular seed is called the *stream* for that seed. A PRNG produces reproducible streams because the algorithm produces the same sequence for each specified seed. Thus the seed completely determines the stream. Although it is deterministic, a modern PRNG can generate extremely long sequences of numbers whose statistical properties are virtually indistinguishable from a truly random uniform process.

The internal state of a PRNG requires a certain number of bits. Because each call updates the state, a PRNG will eventually return to a previously seen state if you call it enough times. At that point, the stream will repeat an earlier sequence of values. The number of calls before a stream begins to repeat is called the *period* (or *cycle length*) of the PRNG. High-quality PRNGs provide very long cycle lengths and will not repeat in practice. For example, the shortest cycle length of any PRNG in this paper is about 2^{64}. On a modern eight-core CPU, it would take 1,000 years to generate that many pseudorandom numbers.

In contrast, a hardware-based RNG is not deterministic and does not have a cycle length. A hardware-based RNG samples from an *entropy source*, which is often thermal noise within the silicon of a computer chip. The random entropy source is used to initialize a PRNG in the chip, which generates a short stream of pseudorandom values before resampling from the entropy source and repeating the process. Consequently, a hardware-based stream consists of short pseudorandom streams that are repeatedly initialized from a random source. These streams are not reproducible.

In SAS, you can use the STREAMINIT subroutine to set the seed for a PRNG. You can use the RAND function to produce high-quality streams of random numbers for many common distributions. The first argument to the RAND function specifies the name of the distribution. Subsequent arguments specify parameters for the distribution. For example, the following DATA step initializes the default PRNG with the seed value 12345 and then generates five observations from the uniform distribution on (0,1):

```
data Rand1(drop=i);
   array u[5];
   call streaminit(54321);        /* use default RNG */
   do i = 1 to dim(u);
      u[i] = rand('uniform');
   end;
run;

proc print data=Rand1 noobs; run;
```

The output is shown in Figure 1. These numbers are reproducible: the program generates the same pseudorandom numbers every time you run the program.

Figure 1 Stream from Default RNG

u1	u2	u3	u4	u5
0.43223	0.59780	0.77860	0.17483	0.39415

The STREAMINIT call also enables you to specify the RNG. If you are running on an Intel CPU that supports a hardware-based RNG, you can request it by using the 'RDRAND' option, as follows:

```
data RandMT;
   array u[5];
   call streaminit('RDRAND');     /* request hardware-based RNG */
   do i = 1 to dim(u);
      u[i] = rand('uniform');
   end;
run;
```

Hardware-based RNGs are not reproducible, so the DATA step generates a different set of numbers every time you run the program.

NONUNIFORM RANDOM VARIATES

A *variate* is a realization of a random variable, sometimes called a *random draw* from the distribution. An RNG generates a stream of random uniform variates. Obviously statisticians also need random variates from nonuniform distributions such as the Bernoulli, exponential, and normal distributions, to name a few. Fortunately, you can transform one or more random uniform variates to produce random variates for any other probability distribution (Devroye 1986; Gentle 2003). The RAND function internally implements transformations for about 30 common distributions.

For example, to generate a Bernoulli draw with success probability p, the RAND function generates a uniform variate U and sets $B = 1$ if $U \leq p$. To generate an exponential random variate, the RAND function sets $E = -\log(U)$. Other distributions (such as the normal) might require multiple uniform variates. These transformations are handled automatically by the RAND function. You only need to specify the name of the distribution and the parameter values, as shown in the following example:

```
data Rand3;
   array x[5];
   call streaminit(54321);          /* use default RNG   */
   x[1] = rand('uniform');          /* U(0,1)            */
   x[2] = rand('Bernoulli', 0.5);   /* Bern(p=0.5)       */
   x[3] = rand('exponential');      /* Exp(scale=1)      */
   x[4] = rand('normal');           /* N(mu=0,  sigma=1) */
   x[5] = rand('normal', 10, 2);    /* N(mu=10, sigma=2) */
run;

proc print data=Rand3 noobs; run;
```

The output is shown in Figure 2; compare it with Figure 1.

Figure 2 Nonuniform Random Variates

x1	x2	x3	x4	x5
0.43223	0	0.25026	-1.17215	9.23687

Both DATA steps use the same seed, which means that they use the same underlying stream of uniform variates. You can see that the first uniform variate (**x1**) is the same in both data sets. The Bernoulli value (**x2**) in Figure 2 is obtained from the second uniform variate, which is greater than $p = 0.5$. The exponential value (**x3**) in Figure 2 is equal to $-\log(u_3)$, where u_3 is the third uniform variate. The fourth value (**x4**) is a draw from the standard normal distribution, and the fifth value (**x5**) is a draw from the normal distribution with mean 10 and standard deviation 2.

In addition to the common distributions that are supported by the RAND function, you can combine or transform these distributions in myriad ways to produce random variates from more sophisticated distributions, such as mixture distributions and compound distributions (Wicklin 2013a).

RANDOM-NUMBER GENERATORS IN SAS

Long-time SAS programmers might remember older functions that were used prior to SAS 9.1, including the RANUNI, RANNOR, and RANBIN functions. Those functions are deprecated because they are not suitable for modern simulations that require millions of random numbers. You should use the RAND function in SAS to generate random values from univariate distributions. See Wicklin (2013b) for a list of reasons to prefer the RAND function over the older routines.

When the RAND function was first introduced in SAS 9.1, it supported only the original Mersenne twister (MT) algorithm (Matsumoto and Nishimura 1998). Later versions of SAS introduced the RDRAND RNG for Intel processors (Intel Corporation 2014) and the hybrid MT algorithm (Matsumoto and Nishimura 2002); the latter improves the quality of the MT algorithm for certain seed values. In SAS 9.4M5, the RAND function was enhanced further to support a variety of modern 64-bit RNGs. The following list describes the supported RNGs:

MT1998 — The original 1998 32-bit Mersenne twister algorithm (Matsumoto and Nishimura 1998). This was the default RNG for the RAND function prior to SAS 9.4M3. For a small number of seed values (those exactly divisible by 8,192), this RNG does not produce a sufficiently random stream of numbers. Consequently, other RNGS are preferable.

MTHYBRID — A hybrid method that improves the MT1998 method by using the MT2002 initialization for seeds that are exactly divisible by 8,192. This is the default RNG, beginning with the SAS 9.4M3.

MT2002 — The 2002 32-bit Mersenne twister algorithm (Matsumoto and Nishimura 2002).

MT64 — A 64-bit version of the 2002 Mersenne twister algorithm.

PCG — A 64-bit permuted congruential generator (O'Neill 2014) with good statistical properties.

TF2 — A 2x64-bit counter-based RNG that is based on the Threefish encryption function in the Random123 library (Salmon et al. 2011). The generator is also known as the 2x64 Threefry generator.

TF4 — A 4x64-bit counter-based RNG that is also known as the 4x64 Threefry generator.

RDRAND — A hardware-based RNG that is repeatedly reseeded by using random values from thermal noise in the chip. The RNG requires an Intel processor (Ivy Bridge and later) that supports the RDRAND instruction.

In general, a 64-bit RNG can approximate the tails of nonuniform distributions better than a 32-bit equivalent can. For example, the 64-bit version of MT performs slightly better than the 32-bit version in some randomness tests. The 64-bit RNGs are available even if you are running a 32-bit version of SAS.

You can use the STREAMINIT subroutine to specify an RNG and, optionally, set the seed value. After an RNG is initialized, it remains in use until the end of the DATA step. You cannot use two different RNGs in a single DATA step.

Each of the following DATA steps calls a different RNG to generate five uniform variates. The data sets are then merged for easy comparison.

```
data RandMT(drop=i);
   call streaminit('MTHYBRID', 12345); /* MTHYBRID (default) method */
   do i = 1 to 5;
      MT = rand('uniform'); output;
   end;
run;

data RandPCG(drop=i);
   call streaminit('PCG', 12345);      /* PCG method, same seed */
   do i = 1 to 5;
      PCG = rand('uniform'); output;
   end;
run;

data RandTF(drop=i);
   call streaminit('TF2', 12345);      /* TF2 method, same seed */
   do i = 1 to 5;
      TF2 = rand('uniform'); output;
   end;
run;

data Rand4;
   merge RandMT RandPCG RandTF;
run;

proc print data=Rand4 noobs; run;
```

Figure 3 Streams from Different RNGs

MT	PCG	TF2
0.58330	0.82524	0.33476
0.99363	0.24079	0.39276
0.58789	0.21400	0.91942
0.85747	0.99923	0.29159
0.82469	0.09084	0.26579

Figure 3 shows that the streams from different RNGs are different even if you use the same seed to initialize the RNGs. If you compare Figure 3 with Figure 1, you can see the effect of the seed parameter. The values in Figure 1 are generated by using 54321 as a seed value to the MTHYBRID RNG; the values in the first column of Figure 3 are different because 12345 is the seed value.

CHOOSING SEEDS FOR RNGS

A PRNG generates a reproducible stream when you specify a positive seed value. This leads to two questions:

- How do you choose a seed value that results in a high-quality stream of random variates?
- How do you ensure that a stream is not reproducible?

The answer to the first question is easy: You can choose any value that you want (Wicklin 2017). Whether you use your birthday, your phone number, or an easy-to-remember number such as 1776 or 12345, the underlying PRNG will generate a high-quality pseudorandom stream.

To ensure that a program generates different numbers every time it runs, specify 0 as the seed value. The value 0 causes SAS to internally generate a seed by using the following algorithm:

- If SAS is running on a system that supports a hardware-based RNG, call the hardware-based RNG to generate a random seed value.
- Otherwise, use the current date, time, and possibly other factors to generate a seed value.

If SAS internally generates a seed, it stores that seed in the SAS macro variable, &SYSRANDOM, as shown in the following program:

```
data Rand5;
   call streaminit('PCG', 0);          /* auto-generate seed */
   do i = 1 to 5;
      x = rand('uniform'); output;
   end;
run;

%put &sysrandom;  /* display value of auto-generated seed */

data Rand6;
   call streaminit('PCG', &sysrandom);  /* use the same seed */
   do i = 1 to 5;
      x = rand('uniform'); output;
   end;
run;

proc compare base=Rand5 compare=Rand6 short; /* data sets are identical */
run;
```

Figure 4 Automatically Generate a Seed Value

```
                    The COMPARE Procedure
           Comparison of WORK.RAND5 with WORK.RAND6
                        (Method=EXACT)

NOTE: No unequal values were found. All values compared are exactly equal.
```

Of course, you can also use a hardware-based RNG to generate a stream that is not reproducible. However, a hardware-based RNG tends to be slower than a PRNG because it periodically reinitializes from an entropy source.

APPLICATIONS OF RANDOM VALUES

This section uses examples from Wicklin (2013a) to demonstrate statistical applications of random-number streams. The examples show how to use the STREAMINIT function to initialize an RNG and use the RAND function to generate random samples or to resample from data.

Simulate Tossing a Coin

The simplest simulation is a coin toss. The following DATA step simulates tossing a fair coin 10,000 times. The Bernoulli distribution generates a 1 or 0, which you can interpret as heads or tails, respectively:

```
%let NToss = 10000;                    /* number of coin tosses */
%let p = 0.5;                          /* probability of heads */
data Toss(drop=i);
   call streaminit(123);
   do i = 1 to &NToss;                 /* toss coin NToss times */
      Heads = rand("Bernoulli", &p);   /* Heads=1; Tails=0 */
      output;
   end;
run;
```

The following call to PROC FREQ tabulates the number of heads and tails and performs a hypothesis test for the proportion. The frequencies are shown in Figure 5.

```
proc freq data=Toss;
   tables Heads / nocum binomial(p=&p level='1');
run;
```

Figure 5 Summary of 10,000 Coin Tosses

The FREQ Procedure

Heads	Frequency	Percent
0	5051	50.51
1	4949	49.49

For the specified seed, there were 5,051 tails and 4,949 heads. If you change the seed, you will obtain a different result. The binomial test for proportion (not shown) indicates that there is no reason to reject the null hypothesis that the coin is fair ($p = 0.5$). If you rerun this simulation many times, you should expect the null hypothesis to be rejected in 5% of the random samples, assuming a 0.05 significance level.

Simulate a Survey

In a certain town, 60% of voters support the incumbent mayor. One election day, a pollster interviews voters until she finds one who voted for the mayor. How would you write a simulation that estimates the expected number of voters who are interviewed?

To find the expected probability, imagine a large number of pollsters who all interview voters. The average number of voters interviewed by these imaginary pollsters is an estimate for the expected value.

This simulation can be programmed either by smulating the voters or by simulating the pollsters. To simulate the voters, each of whom has a 60% chance of having voted for the mayor, use the following DATA step to simulate a repeated Bernoulli trial for 10,000 imaginary pollsters:

```
%let NTrials = 10000;                  /* number of trials */
%let p = 0.6;                          /* probability of success */
data Survey(drop=i);
   call streaminit(1234);
   do i = 1 to &NTrials;               /* for each pollster */
      NVoters = 0;
      do until( rand("Bernoulli", &p) = 1 ); /* repeatedly poll voters */
         NVoters + 1;                  /* until find one who voted for mayor */
      end;
      output;
   end;
run;

proc means data=Survey;
   var NVoters;
run;
```

Figure 6 shows the result of the simulation. On average, the imaginary pollsters interviewed 1.64 voters before finding one who voted for the mayor. You could use PROC FREQ to display the distribution of the number of voters who are polled.

Figure 6 Statistics for Simulation

The MEANS Procedure

Analysis Variable : NVoters				
N	**Mean**	**Std Dev**	**Minimum**	**Maximum**
10000	1.6357000	1.0069195	1.0000000	10.0000000

Alternatively, simulating each pollster enables you to program the simulation more efficiently. Recall that repeating a Bernoulli trial until success is the definition of the geometric distribution. Therefore, the number of voters that each pollster asks is a random draw from the geometric distribution, as follows:

```
data Survey2(drop=i);
   call streaminit(1234);
   do i = 1 to &NTrials;                  /* for each pollster */
      NVoters = rand("Geometric", &p);    /* how many voters were asked? */
      output;
   end;
run;
```

Data Simulation and Monte Carlo Estimation

Suppose you want to use Monte Carlo simulation to investigate the variance of least squares regression estimates. The following example from Wicklin (2013a, Chapter 11) simulates 1,000 samples of size 50, where the explanatory variable is uniformly distributed and the response variable is defined by the model $Y_i = 1 - 2X_i + \epsilon_i$, where $\epsilon_i \sim N(0, 0.5)$:

```
%let N = 50;                          /* size of each sample      */
%let NumSamples = 1000;               /* number of samples        */
data RegSim(keep= SampleID x y);
   array xx{&N} _temporary_;
   call streaminit('MT64', 9876);
   do i = 1 to &N;                    /* simulate x values one time */
      xx[i] = rand('Uniform');        /* X ~ U(0,1)                 */
   end;

   do SampleID = 1 to &NumSamples;    /* for each simulated sample  */
      do i = 1 to &N;
         x = xx[i];                   /* each sample uses same values */
         eps = rand('Normal', 0, 0.5);
         y = 1 - 2*x + eps;           /* define Y; params are 1 and -2 */
         output;
      end;
   end;
run;
```

The previous DATA step simulates 1,000 samples. You can use the BY statement in PROC REG to analyze all 1,000 samples with one call, thereby approximating the sampling distribution of parameter estimates. For other simulation tips, see Wicklin (2015).

Simple Random Sample with Replacement

In bootstrap and other resampling methods, you use random sampling with replacement to estimate the sampling distribution of various statistics. In SAS, the simplest way to perform random sampling with replacement is to use the SURVEYSELECT procedure with METHOD=URS (Wicklin 2013a, Chapter 15). However, you can also implement a random sampling scheme by using the RAND function in the DATA step. In the following DATA step, the RAND function generates a random integer k in the range $[1, N]$, where N is the number of observations in a data set. The SET statement with the POINT= option then uses random access to select the kth observation. This is repeated 1,000 times to create 1,000 bootstrap samples.

```
%let MyData = Sashelp.Class;
data BootDS;
   call streaminit('TF4', 123);       /* initialize RNG */
   do SampleID = 1 to 1000;           /* generate bootstrap samples */
      do i = 1 to N;                  /* size of sample = size of data set */
         k = rand('Integer', 1, N);   /* random integer in [1,N] */
         set &MyData point=k nobs=N;  /* random access of selected obs */
         output;
      end;
   end;
   STOP;                              /* stop processing (no EOF condition) */
run;
```

You can then use univariate or multivariate procedures to approximate the sampling distribution of a statistic.

INDEPENDENT STREAMS FOR RNGS

Each of the previous examples uses one DATA step to simulate or resample data. The simulations use a single random number stream from a chosen RNG. Most statistical simulations can be implemented by using this programming paradigm.

However, in rare cases you might need to use multiple streams within a single program. Examples of situations that require multiple seeds include the following:

- A program that contains two different DATA steps, each of which generates random values. If the second step uses the same pseudorandom stream as the first step, the simulation or analysis will be biased.
- Distributing a computation on multiple nodes on a grid.
- A macro loop that generates random values for each iteration.

In previous versions of SAS, there was no easy way to generate independent streams. Programmers would attempt to produce independent by using different seed values. For a Mersenne twister PRNG, two streams that originate from different seeds have a high probability of being independent. However, in theory there is a very small probability that two very long MT streams will overlap. If you generate many thousands of MT streams that have different seeds, the famous birthday problem in probability theory implies a non-negligible probability of *some* streams overlapping. Streams from the PCG PRNG have the same issue when you use different seeds.

SAS 9.4M5 introduces a simple way to generate independent streams, even within a single DATA step. To generate independent streams, use the new STREAM subroutine to set a positive key value for a stream. Streams that have a different key value are statistically independent (for PCG and Threefry PRNGs) or are independent for practical purposes (for the long-period MT PRNGs). The following program demonstrates the STREAM subroutine. Two DATA steps use the same seed value and the same PRNG.

```
%let seed = 54321;
data S0(drop=i);
   key = 0;
   array u[5];
   call streaminit('TF2', &seed);
   /* by default, key=0 so no need to CALL STREAM */
   do i = 1 to dim(u);
      u[i] = rand('uniform');
   end;
run;

data S1(drop=i);
   key = 1;
   array u[5];
   call streaminit('TF2', &seed);
   call stream(key);              /* set key=1 */
   do i = 1 to dim(u);
      u[i] = rand('uniform');
   end;
run;

data rand6;
   set S0 S1;
run;

proc print data=Rand6 noobs; run;
```

Figure 7 shows five uniform values from each stream. The values are different because the steps use different key values. You can use theory to prove that streams with the same seed but different keys are independent for the PCG, TF2, and TF4 PRNGs. Consequently, the stream in the second DATA step is independent from the stream in the first DATA step.

Figure 7 Two Independent Streams

key	u1	u2	u3	u4	u5
0	0.08786	0.39072	0.71361	0.72601	0.47855
1	0.42271	0.17348	0.35770	0.46465	0.51210

INDEPENDENT STREAMS FOR RNGS: MACRO LOOPS

The previous section shows that the STREAM subroutine can generate independent streams from a single seed value. SAS programmers sometimes write a macro that passes in a seed value. By using the STREAM subroutine, you can vary the key according to the value of the macro loop. The following example (which is not efficient) generalizes the example in the previous section. The macro contains two DATA steps, each of which requires random numbers. The first DATA step uses only odd key values; the second uses only even key values. Because a Threefry PRNG is used, the streams are all mutually independent.

```
%macro DoIt(seed=, numIters=);     /* This macro is NOT efficient */
/* delete Sum data set if it exists */
proc datasets nolist; delete Sum; run;

%do k = 1 %to &numIters;
   data S0;
      call streaminit('TF2', &seed); /* use TF2 */
      call stream(2*&k - 1);         /* odd keys: 1, 3, 5, ... */
      /* do something with random numbers */
      array u[5];
      do i = 1 to dim(u);  u[i] = rand("uniform"); end;
   run;

   data S1;
      call streaminit('TF2', &seed);
      call stream(2*&k);             /* even keys: 2, 4, 6, ... */
      /* do something else with random numbers */
      /* such as add new random numbers to previous */
      set S0;
      array u[5] u1-u5;
      do i = 1 to dim(u);  u[i] = u[i] + rand("uniform"); end;
   run;

   /* use PROC APPEND to append rows (not efficient) */
   proc append base=Sum data=S1(drop=i); run;
%end;
%mend;

%DoIt(seed=123, numIters=3);
proc print data=Sum noobs; run;
```

The output is shown in Figure 8. Each cell in the table is the sum of two independent uniform random variates. Each row in the column is from a different iteration of the macro loop.

Figure 8 Sums of Independent Streams

u1	u2	u3	u4	u5
0.88601	1.16386	1.36010	1.24071	1.39139
0.72090	0.92572	0.76661	0.93603	1.04943
0.87703	1.03885	1.07914	0.61115	1.59920

You can often rewrite a simulation program to get rid of the macro loop and the call to PROC APPEND. For example, you can rewrite the previous macro program as a single DATA step, as follows:

```
%let seed = 123;
%let numIters=3;
data Sum2(drop=i k);
   call streaminit('TF2', &seed);
   array u[5];
   do k = 1 to &numIters;
      call stream(2*k - 1);  /* odd keys: 1, 3, 5, ...  */
      /* generate and use random numbers */
      do i = 1 to dim(u); u[i] = rand("uniform"); end;

      call stream(2*k);      /* even keys: 2, 4, 6, ... */
      /* add new random numbers to previous */
      do i = 1 to dim(u); u[i] = u[i] + rand("uniform"); end;
      output;
   end;
run;
```

The SUM2 data is identical to the SUM data, but the version without macro loops is much more efficient. An even more efficient version is to eliminate the CALL STREAM statements and draw all the variates from a single stream.

SAS 9.4M5 also enables you to switch between multiple streams. Suppose you start generating numbers from an RNG with KEY=1, but later create a second stream with KEY=2. When the second stream becomes active, the state of the first stream is preserved. You can call STREAM(1) to reactivate the first stream. It will generate new random variates beginning from its preserved state. For an example, see the section “Using Random-Number Functions and CALL Routines in the DATA Step” in *SAS Functions and CALL Routines: Reference*.

INDEPENDENT STREAMS FOR RNGS: PARALLEL COMPUTATIONS

An impetus for implementing new RNGs in SAS was to support multithreaded and parallel generation of random numbers. For example, the DS2 procedure enables you to define a program that runs in parallel on every thread. The RNG on each thread is initialized to use a key value that is derived from the automatic variable _THREADID_, which varies for each thread. Recall that PROC DS2 does not support subroutines, only functions. Therefore, the STREAMINIT call is a function call in PROC DS2.

In the following program, the main program creates four instances of a threaded program. Each thread uses the same RNG and seed value, but internally SAS sets a key value so that each thread generates an independent stream of 2,500 random values from three different probability distributions.

```
%let NPerThread = 2500;     /* sample size per thread */
%let nThreads = 4;          /* number of threads in PROC DS2 */

proc ds2;
/* the 'thread_pgm' program runs in each thread */
thread thread_pgm / overwrite=yes;
   declare double k Uniform Normal Events;
   method init();
      /* each thread uses same RNG and seed, but a unique internal key */
      streaminit('TF2', 7654321); /* stream is set from _threadID_ */
      do k = 1 to &NPerThread;
         Uniform = rand('Uniform');        /* U(0,1) */
         Normal  = rand('Normal', 50, 10); /* N(50, 10) */
         Events  = rand('Binom', 0.2, 80); /* Bin(p=0.2, N=80) */
         output;
      end;
   end;
endthread;
```

```
/* the main program runs all the threads */
data Rand7(overwrite=yes);
   declare thread thread_pgm t;
   method run();
      set from t threads=&nThreads;
      output;
   end;
enddata;
run; quit;
```

The following call to PROC MEANS analyzes the 10,000 random observations that are generated. The sample statistics for each variable are shown in Figure 9.

```
proc means data=Rand7;
   var Uniform Normal Events;
run;
```

Figure 9 Results of Multithreaded Random-Number Generation

The MEANS Procedure

Variable	N	Mean	Std Dev	Minimum	Maximum
Uniform	10000	0.4976269	0.2882899	0.000101161	0.9999994
Normal	10000	49.9381140	9.9464283	14.6428877	93.0242909
Events	10000	16.0063000	3.5652674	4.0000000	31.0000000

The sample means are close to the expected values for each distribution. The standard deviations are also close to the population values. These statistics indicate that the RNGs are performing well.

In a similar way, you can generate random numbers in parallel in the CAS DATA step that runs in SAS® Viya®. Each node-thread combination is automatically assigned an internal key value so that each thread produces an independent stream.

With a little extra work, you can modify the program so that you can specify the total number of observations to generate. The threaded program can handle the situation where the specified number of observations is not evenly divisible by the number of threads. The following code snippets assign an extra observation to threads that have low-order IDs:

```
%let N = 10003;            /* total sample size */
%let nThreads = 4;         /* number of threads in PROC DS2 */

proc ds2;
   declare double k Uniform Normal Events nPerThread nExtra;
   method init();
      streaminit('TF2', 7654321);
      /* if N not divisible by nThreads, give extra work to first threads */
      nPerThread = int(&N / &nThreads);
      nExtra = &N - nPerThread * &nThreads;
      if _threadid_ <= nExtra then
         nPerThread + 1;

      do k = 1 to nPerThread;
         /* generate random values here */
      end;
   end;

   /* remaining DS2 code is the same */
```

OTHER NEW FEATURES OF RNGS

The new STREAMREWIND subroutine enables you to rewind a stream to the beginning. This feature is convenient for testing. Recall that nonuniform variates are generated from one of more uniform variates. The following DATA step demonstrates the transformations that create Bernoulli and exponential variates. The program generates five uniform variates, rewinds the stream, generates five Bernoulli variates, rewinds the stream, and generates five exponential variates.

```
data Rand8(drop=i);
   length RNG $11;
   array x[5];
   call streaminit(54321);          /* use default RNG */
   RNG = 'Uniform';                 /* generate 5 uniform variates */
      do i = 1 to dim(x); x[i] = rand('uniform'); end;
      output;
   call streamrewind;               /* rewind stream to beginning  */
   RNG = 'Bernoulli';               /* generate 5 Bernoulli trials */
      do i = 1 to dim(x); x[i] = rand('Bernoulli', 0.5); end;
      output;

   call streamrewind;               /* rewind stream to beginning  */
   RNG = 'Exponential';             /* generate 5 exponential variates */
      do i = 1 to dim(x); x[i] = rand('Exponential'); end;
      output;
run;

proc print data=Rand8 noobs; run;
```

Figure 10 shows the result of using the STREAMREWIND subroutine. The second and third rows are obtained by transformations of the first row. A number in the second row is 1 when the corresponding number in the first row is less than or equal to 0.5. The third row is the negative log transform of the first row.

Figure 10 Rewinding the Stream between Generating Variates

RNG	x1	x2	x3	x4	x5
Uniform	0.43223	0.59780	0.77860	0.17483	0.39415
Bernoulli	1.00000	0.00000	0.00000	1.00000	1.00000
Exponential	0.83879	0.51450	0.25026	1.74397	0.93103

Be careful if you use the STREAMREWIND subroutine. It results in dependent streams of numbers, which can result in biases and correlations in statistical computations. It is usually not necessary in simulation studies.

QUALITY OF RNGS

The quality of an RNG measures how well the statistical properties of a pseudorandom stream match those of a truly random uniform process. L'Ecuyer and Simard (2007) published a library of numerical tests of randomness. The library contains three suites of tests. A *Crush-resistant* RNG is one that passes all three suites when called with a wide variety of seeds. A Crush-resistant RNG has excellent statistical properties.

Mersenne twister generators are not Crush-resistant because they systematically fail linear complexity tests (Vigna 2009). However, linear complexity tests have less practical importance than other tests that measure randomness. Of the new RNGs, the 64-bit permuted congruential generator (PCG) is Crush-resistant, as are both Threefry generators (TF2 and TF4).

Of course, other considerations are also important, such as speed, length of period, and the ability to support independent streams for parallel computations. Table 1 summarizes properties of the RNGs that are available in the RAND function in SAS 9.4M5.

Table 1 Quality of RNGs in SAS

Method	Period	Quality	Indep Streams	Performance
MTHYBRID	2^{19937}	Good	No	1.0
MT1998	2^{19937}	Fair	No	1.0
MT2002	2^{19937}	Very Good	No	1.0
MT64	2^{19937}	Very Good	No	1.3
PCG	2^{64}	Excellent	2^{63}	1.2
TF2	2^{129}	Excellent	2^{128}	1.5
TF4	2^{258}	Excellent	2^{256}	1.6
RDRAND	Infinite	Excellent	N/A	4.0

The Performance column indicates the relative performance of each RNG based on 10 runs of a DATA step that generates one billion uniform variates on a Windows PC. (Timings for Linux are similar and are available in the SAS documentation.) The first three rows of Table 1 are 32-bit Mersenne twister RNGs, which are the fastest RNGs. The value 1.0 in the Performance column indicates that these are the basis for the comparisons. For example, the 64-bit MT RNG is assigned the value 1.3, which means it is about 30% slower.

The MT RNGs cannot generate independent streams, which limits their usefulness in distributed computations. The PCG RNG supports independent streams, and it is the fastest of the new Crush-resistant algorithms. It has a relative performance of 1.2, which means it is only about 20% slower than the 32-bit MT RNGs. Consequently, PCG is highly recommended as an RNG that provides both speed and statistical quality. For applications that generate very long streams, the TF2 RNG offers a longer period. It is about 50% slower than the 32-bit MT RNGs, or about 25% slower than the PCG.

The hardware-based RDRAND method is the slowest RNG. The streams for RDRAND are not reproducible, and they do not have a period.

SUMMARY

This paper introduces the new random-number generators in SAS 9.4M5. In addition to the new RNGs, SAS also supports new subroutines that enable you to use multiple streams (CALL STREAM) or even to rewind an existing stream (CALL STREAMREWIND). This paper shows several examples of how to initialize and generate random numbers effectively and efficiently in SAS, including how to generate random numbers in parallel.

REFERENCES

Devroye, L. (1986). *Non-uniform Random Variate Generation*. New York: Springer-Verlag. http://luc.devroye.org/rnbookindex.html.

Gentle, J. E. (2003). *Random Number Generation and Monte Carlo Methods*. 2nd ed. Berlin: Springer-Verlag.

Intel Corporation (2014). "Intel Digital Random Number Generator (DRNG) Software Implementation Guide." Accessed December 23, 2017. https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide.

L'Ecuyer, P., and Simard, R. (2007). "TestU01: A C Library for Empirical Testing of Random Number Generators." *ACM Transactions on Mathematical Software* 33:40 pages.

Matsumoto, M., and Nishimura, T. (1998). "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-random Number Generator." *ACM Transactions on Modeling and Computer Simulation* 8:3–30.

Matsumoto, M., and Nishimura, T. (2002). "Mersenne Twister with Improved Initialization." Accessed April 10, 2015. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html.

O'Neill, M. E. (2014). *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Technical report HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA. Accessed December 23, 2017. https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf.

Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. (2011). "Parallel Random Numbers: As Easy as 1, 2, 3." In *SC '11: 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12. Washington, DC: IEEE Computer Society.

Vigna, S. (2009). "PRNG Shootout." Accessed December 24, 2017. http://xoroshiro.di.unimi.it/.

Wicklin, R. (2013a). *Simulating Data with SAS*. Cary, NC: SAS Institute Inc.

Wicklin, R. (2013b). "Six Reasons You Should Stop Using the RANUNI Function to Generate Random Numbers." July. http://blogs.sas.com/content/iml/2013/07/10/stop-using-ranuni/.

Wicklin, R. (2015). "Ten Tips for Simulating Data with SAS." In *Proceedings of the SAS Global Forum 2015 Conference*. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings15/SAS1387-2015.pdf.

Wicklin, R. (2017). "How to Choose a Seed for Generating Random Numbers in SAS." June 1. https://blogs.sas.com/content/iml/2017/06/01/choose-seed-random-number.html.

ACKNOWLEDGMENTS

The authors are grateful to Joel Dood and Mark Little for their helpful comments and suggestions. Some of the ideas in this paper also appear in the SAS documentation, which was written by the authors.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the second author:

Rick Wicklin
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513