

Pseudo-Random Numbers in SAS® Model Implementation Platform

Joel Dood, SAS Institute Inc.

ABSTRACT

Monte Carlo simulation can be used in SAS® Model Implementation Platform to estimate the average cash flow from a loan under one or more set of economic assumptions. Some users want an independent simulation for each scenario, while others might want to use the same set of random numbers for each scenario. If you do not set up the random numbers in a specific way, then you might not get the numbers that you are expecting. SAS® 9.4M5 introduced new pseudo-random number generators (PRNGs) and new subroutines that enable you to manipulate multiple pseudo-random number streams. These enhancements enable SAS Model Implementation Platform to generate PRNGs that are well-suited for a wide variety of needs. This paper describes techniques for using these features correctly, according to your specific purpose.

INTRODUCTION

This paper provides a simple example of a Monte Carlo simulation that you can run in SAS Model Implementation Platform. Suppose you are a bank that has issued 1,000 30-year fixed rate mortgages. At any given time, each loan can be in one of these four states: current, 30 days late, defaulted, or paid-off. The last two states are known as terminating states. This is because once a loan has been paid off or defaulted, the loan is off your books. The other two states are more dynamic. For example, a loan can be current for several months, then become 30 days late for two months, and then become current again for the remaining horizons. You have developed statistical models that estimate the probability of going into each state. These models depend on several factors:

1. The current state of the loan.
2. The portfolio variables such as credit score, current balance, and the age of the loan.
3. The economic variables such as unemployment rate and average home values.

You want to estimate how much interest income and principal repayments you can expect during the next twelve months. **Because you don't know for sure which state a loan is going to go to**, you use random draws to simulate the loan based on the probabilities that you have estimated. This simulation, along with your user-defined logic (UDL), determine how much simulated interest and principal repayments you receive for each of the next twelve months. This simulation is repeated several times and the average interest and average principal is calculated for each horizon across the simulations.

SAS Model Implementation requires that the random numbers are independent across the simulations so that you obtain statistically valid estimates of interest and principal. Distributed and multi-thread computing environments are great tools that allow for faster execution of the simulations. However, these extra threads complicate the process of generating the independent pseudo-random numbers that the Monte Carlo models need.

Before SAS 9.4M5, the RAND function used the Mersenne Twister algorithm exclusively to generate pseudo-random numbers. There are a couple of problems with this. The first is that this algorithm has only one stream. Although this stream has an extremely long period,

there is no guarantee (although a very high probability), that two simulations using two different seeds are using disjoint subsets of the stream.

The second problem is that there are sections in the Mersenne Twister algorithm that do not produce good numbers from a statistical point of view. It is possible for the internal state to have an unbalanced number of bits set. In these cases, it can take many draws before a good balance is restored.

PROBLEMS USING ONLY ONE STREAM

When you perform a Monte Carlo analysis in SAS Model Implementation Platform, you often want the draws for each loan to be independent of one another. One way to try to do that is to have a separate seed for each loan. When each loan is evaluated, the stream is initialized using its own seed, and hopefully independent draws will be generated.

However, because you are using only one stream, there is no guarantee that the numbers are independent. See the **"What Are Streams For?"** section in O'Neil (2017) for additional details. For example, consider the two subsets produced by using seed values 1 and 69069 with the default generator. These two seeds, when the first is lagged by one number, produce values that are highly correlated:

```
data sample;
  call streaminit("MTHYBRID"); /* Default generator */
  array seed1[0:250] _temporary_;
  rc = rand("RESETSEED",1);
  do i=0 to 250;
    seed1[i] = rand("UNIFORM"); /* Start at zero to create a lag */
  end;
  rc = rand("RESETSEED",69069);
  do i=1 to 250;
    x = seed1[i];
    y = rand("UNIFORM");
    output;
  end;
run;

proc gplot;
plot y*x;
run;
quit;
```

Figure 1 shows the correlation between the two seeds.

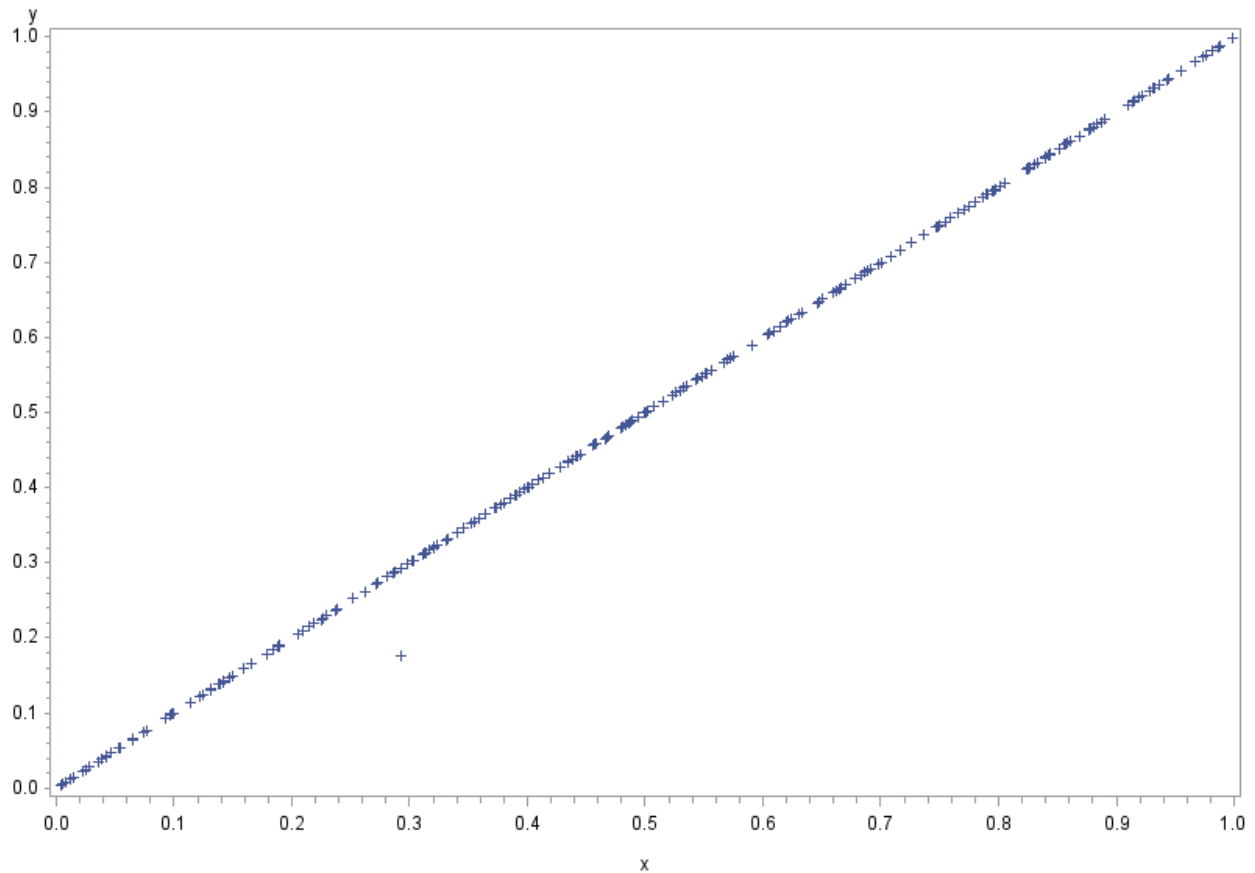


Figure 1. Correlation between Two Subsets Produced Using Different Seed Values and the Default Mersenne Twister Generator

There is no way to tell exactly how far apart within the stream these two subsets are. However, the two subsets are disjoint. If you continue and generate a lot more numbers (for example, one million), then they eventually become independent.

This problem is not restricted to the default Mersenne Twister generator. Consider the Permuted Congruential Generator (PCG) with seed values 2316313810 and 5803484695. These two seeds initialize the generator exactly one step away from each other:

```
data sample;
  call streaminit("PCG");
  array seed1[0:250] _temporary_;
  rc = rand("RESETSEED",2316313810);
  do i=0 to 250;
    seed1[i] = rand("UNIFORM"); /* Start at zero to create a lag */
  end;
  rc = rand("RESETSEED",5803484695);
  do i=1 to 250;
    x = seed1[i];
    y = rand("UNIFORM");
    output;
  end;
run;
```

```
proc gplot;  
plot y*x;  
run;
```

Figure 2 shows the values that coincide.

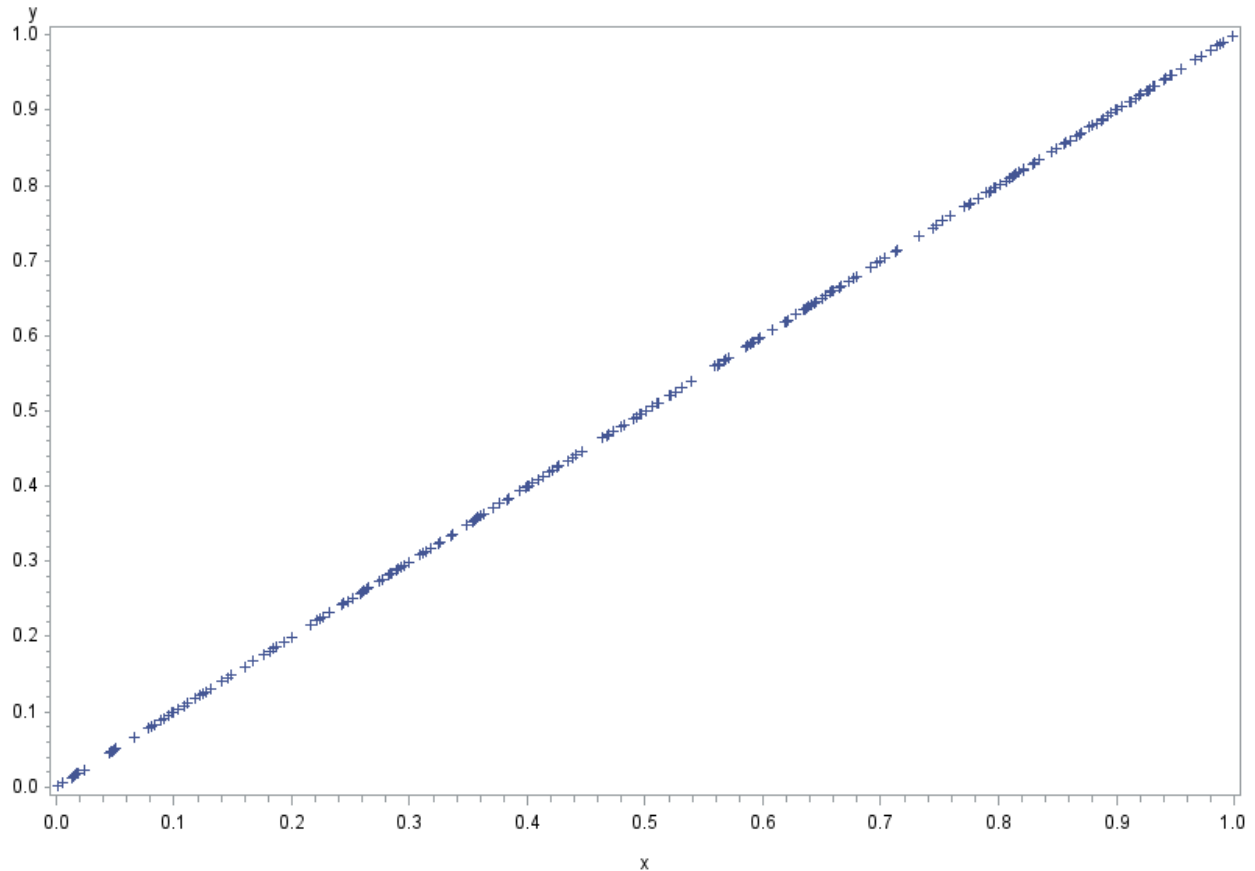


Figure 2. Correlation between Two Subsets Produced Using Different Seed Values and the PCG Generator

In this example, the numbers are not in disjoint parts of the stream. The two subsets overlap, and the lagged numbers will never diverge no matter how many you generate.

The seeds in these two examples were chosen to demonstrate worst-case behavior. If you use other **seed values, then you probably won't run into this problem.** In addition, SAS 9.4M5 allows for distinct streams that allow you to easily avoid this problem.

SUMMARY OF NEW FEATURES

Details of the new pseudo-random number features can be found in Wicklin and Sarle (2018). Random number streams are still initialized with a Seed value. However, now you can use two additional parameters, KEY and ID, with a random number generator. These new parameters, which can be used with any of the SAS generators, allow you to have distinct streams when you use the PCG, ThreeFry2 (TF2), or ThreeFry4 (TF4) generators. Default values for both parameters are zero.

You can use up to 20 bits to specify the Key value, and up to 43 bits to specify the ID value. The functions that use these new values are presented in the following three sections in the order in which they should be used.

NEW GENERATORS

The STREAMINIT call routine has a new argument that allows you to specify the generator that you want to use:

```
CALL STREAMINIT(Seed Value, "GENERATOR");
```

The new generators that can be specified using this routine include a 64-bit version of Mersenne Twister, **Permuted Congruential Generator (O'Neil 2014)**, and **two versions of the ThreeFry generator (Salmon et al. 2011)**.

Two important properties of the CALL STREAMINIT routine remain the same. First, if you specify a Seed value of zero, then SAS auto-generates a seed for you. This seed will not yield reproducible numbers. Each time you run your program, a different seed is auto-generated for you. Second, the CALL STREAMINIT routine is executed only one time per thread. If you make subsequent calls to it within a thread, then the routine exits immediately without doing anything.

KEY VALUE

The new CALL STREAM routine is used to specify the Key value for a stream:

```
CALL STREAM(Key Value);
```

The first time that you call this routine with a specific Key value SAS creates a new instance of a stream in memory and makes that new stream active. Subsequent calls to this routine with that specific Key value make that stream active.

ID VALUE

The new RESETSEED option in the RSKRAND function allows you to specify the Seed value and ID value for the active stream:

```
rc = RSKRAND("RESETSEED",Seed Value, ID Value);
```

This option re-initializes the active stream using the Seed and ID values that you specify. The Key value for the active stream is not changed.

SELECTING A PSEUDO-RANDOM NUMBER GENERATOR

You can use any of the several random number generators that SAS provides. PCG, TF2, TF4 are top choices because they all have excellent statistical properties and distinct streams. You use the CALL STREAMINIT routine to specify the one that you want. There are a couple of good locations for this statement that will execute efficiently.

The code in the following two sections creates one instance of the PCG, or whichever generator you specify, on each thread. Because the seed is not specified, SAS auto-generates one for you. This is okay, because you can specify the seed value when you set the ID value in your UDL. When you do this, you will get reproducible results even if the number of nodes or threads are changed.

PRE-EXECUTION PROGRAM

The first good location for the CALL STREAMINIT routine is inside a pre-execution program:

```

proc risk;
  env new=mipenv.risk_env inherit=(mimp_env.base_risk_env);
  setoptions NOBACKSAVE;
  env save;
run;

proc compile env=mipenv.risk_env
  outlib=mipenv.risk_env
  package=funcs;
method base_project kind=project;
  beginblock thread_init;
    call streaminit("PCG");
  endblock;
endmethod;
run;
quit;

```

This is more efficient than executing inside your UDL because the routine gets executed only one time per thread. This method is also flexible. For example, you can use one pre-execution program to use the Mersenne Twister 2002 algorithm, and another one to use the ThreeFry4 algorithm.

BASE ENVIRONMENT

If your entire organization always wants to use the **same generator and you don't want to** have to run a pre-execution program for every run, you can also select a generator in the base environment for your work group. You would have to re-create the base environment only one time, and then all runs in that work group use that generator for Monte Carlo simulations. To do this, the first task is to modify the `rskmimpl_base_env.sas` file for your workgroup, which is in the `&MI_WORK_GROUP_ROOT/input/mimp_env` directory.

Edit the `rskmimpl_base_env.sas` file and look for the following code:

```

proc risk;
  env new=mimp_env.base_risk_env inherit=(mimp_sas.sas_risk_env)
  label="Customer site specific base environment";

  /*- create default project -*/
  project mipProject projectmethods = base_project;

  env save;
run;

```

Immediately after this code, you can specify the generator using this code:

```

proc compile env=mimp_env.base_risk_env
  outlib=mimp_env.base_risk_env
  package=funcs;
method base_project kind=project;
  beginblock thread_init;
    call streaminit("PCG");
  endblock;
endmethod;
run;
quit;

```

The only differences between this code and the code used in the pre-execution program, is the environment that is specified for the ENV and OUTLIB arguments.

After this code has been saved, you need to submit the create_base_env.sas file for your workgroup, which is in the &MI_WORK_GROUP_ROOT directory.

SELECTING A KEY

Because the number of distinct Key values that you can use is a lot smaller than the number of distinct ID values, it makes sense to associate Key values with scenarios, and ID values with loans. However, if you have fewer than one **million loans**, it doesn't really matter which parameter is used for which input.

As mentioned earlier, you specify a key value using the CALL STREAM routine. To associate this value with your scenarios, it is helpful if you can add a variable to your economic data (for example, RandomKey) **that you can use. It doesn't really matter what particular values you use.** However, it is probably easiest just to number the scenario or economic simulations sequentially starting with 1. A small sample of economic scenario data is shown in Figure 3.

	scenario_name	RandomKey	date
1	base	1	02/01/2019
2	base	1	03/01/2019
3	base	1	04/01/2019
4	base	1	05/01/2019
5	adverse	2	02/01/2019
6	adverse	2	03/01/2019
7	adverse	2	04/01/2019
8	adverse	2	05/01/2019
9	severe	3	02/01/2019
10	severe	3	03/01/2019
11	severe	3	04/01/2019
12	severe	3	05/01/2019

Figure 3. Economic Scenario Data Sample

A small sample of economic simulation data is shown in Figure 4.

	SimulationReplication	SimulationTime	RandomKey	_date_
1	1	1	1	02/01/2019
2	1	2	1	03/01/2019
3	1	3	1	04/01/2019
4	1	4	1	05/01/2019
5	2	1	2	02/01/2019
6	2	2	2	03/01/2019
7	2	3	2	04/01/2019
8	2	4	2	05/01/2019
9	3	1	3	02/01/2019
10	3	2	3	03/01/2019
11	3	3	3	04/01/2019
12	3	4	3	05/01/2019

Figure 4. Economic Simulation Data Sample

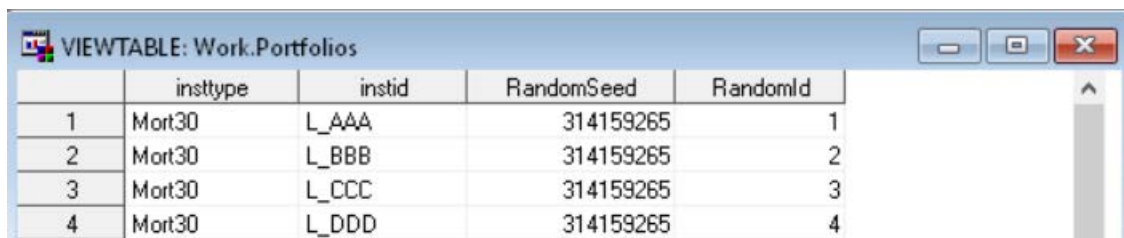
The Key is specified only for the first horizon. Therefore, you only need to put data for RandomKey in the economic data sets for that first horizon. The other horizons can have missing values, or simply repeat the RandomKey of the first horizon. The important point is that each SimulationReplication or scenario_name has its own unique value for RandomKey.

In your UDL, use the economic simulation data with the following code:

```
if SimulationTime = 1 and MCSimulationRep = 1 then do;
    call stream(RandomKey);
end;
```

SELECTING A SEED AND AN ID

As mentioned earlier, you can specify a seed value and an ID value using the RESETSEED option on the RSKRAND function. To associate these values with your loans, it is helpful if you can add variables to your portfolio data (for example, RandomSeed and RandomId) that **you can use. It doesn't really matter what particular values you use. For RandomId**, no two loans should have the same value. Therefore, it is probably easiest just to number the loans sequentially starting with 1. For RandomSeed, **you generally don't** need to have the values unique. In fact, you can just use the same value for all loans. A small sample of portfolio data is shown in Figure 5.



	instype	instid	RandomSeed	RandomId
1	Mort30	L_AAA	314159265	1
2	Mort30	L_BBB	314159265	2
3	Mort30	L_CCC	314159265	3
4	Mort30	L_DDD	314159265	4

Figure 5. Portfolio Data Sample

When you use a good pseudo-**random number generator, it doesn't matter what seed you use** (Wicklin 2017). Because you have separate streams for each different loan and scenario, you can use the same seed for every loan if you want to. You could also use the loan ID as a seed. Either way, you will get reproducible and good quality random numbers.

For more details about the choice of seed and ID **for use with the PCG generator, see** (O'Neil 2017).

In your UDL, use the portfolio data with the following code:

```
if SimulationTime = 1 and MCSimulationRep = 1 then do;
    rc = rskrand("RESETSEED",RandomSeed,RandomId);
end;
```

INDEPENDENT DRAWS FOR ALL SCENARIOS

Here is an example of what to put in your UDL to have independent draws for all scenarios. The following code assumes that you have already selected a generator:

```
beginblock LOAN_INIT;
    Array TransVec{4} / nosymbols;
endblock;

beginblock MAIN;
if SimulationTime = 1 and MCSimulationRep = 1 then do;
    call stream(RandomKey);
    rc = rskrand("RESETSEED",RandomSeed,RandomId);
end;
```



```

end;
Call TRANSMAT_ELEM(TM1, _from, ., "PROB", TransVec);
_to = rantbl_array(TransVec);
end;
_value_ = Interest;
endblock;

```

If you have a RandomKey variable in either your scenario simulation data or your economic simulation data, this code works for both types of runs. This code will generate reproducible results for each instrument, even if scenarios or loans are added or removed.

SCENARIO RUNS USING SAME STREAM FOR EVERY SCENARIO

To use a distinct stream for each loan, but the same numbers for each different scenario, use the following code in your UDL:

```

if (SimulationTime = 1) and (MCSimulationRep = 1) then do;
  call stream(0);
  rc = rskrand("RESETSEED", RandomSeed, RandomId);
end;

```

A disclaimer to using the same stream for each scenario is that each scenario might not use the same exact random numbers. The horizon that a given loan is terminated can change depending on economic factors, even when using the same random number starting point. For example, suppose you are using the PCG generator and a loan has RandomId=11, RandomSeed=123456789, and the probability of a loan terminating in the first horizon is 0.02 for scenario A, and 0.01 for scenario B. The first random number generated is 0.019 for all scenarios. The loan terminates immediately under scenario A and no more random numbers are generated for that MCSimulationRep. However, the loan is still active under scenario B.

ONE SEED FOR ALL LOANS AND SCENARIOS

If you are running your project on only one thread, then you can use just one seed and not have to worry about the problem of overlapping numbers. You will get reproducible results provided that your portfolio and scenario do not change. However, if you sort your loans in a different order, filter out any loans, or change your scenario names, then your results will change. If you are okay with that, then you can use the following pre-execution program to initialize the stream:

```

proc risk;
  env new=mipenv.risk_env inherit=(mimp_env.base_risk_env);
  setoptions NOBACKSAVE;
  env save;
run;

proc compile env=mipenv.risk_env
  outlib=mipenv.risk_env
  package=funcs;
  method base_project kind=project;
  beginblock thread_init;
    call streaminit("PCG", 123456789);
  endblock;
endmethod;
run;
quit;

```

If you use this method on multiple threads, then the draws you get will not be independent across loans. For example, the first loan on the second thread will get the same draws as the first loan on the first thread.

SAME DRAWS FOR EVERY LOAN

If you want each loan in your portfolio to have all the same draws as each other, then you should use the following code in your UDL:

```
beginblock LOAN_INIT;  
  rc = rskrand("RESETSEED",123456789);  
endblock;
```

For each loan, this code resets the stream using a seed value 123456789. Different scenarios for each loan will have different numbers. If you also want the values to be the same across scenarios, then you should instead reset the stream inside the main block using the following code:

```
If (SimulationTime = 1) and (MCSimulationRep = 1) then do;  
  Rc = rskrand("RESETSEED",123456789);  
end;  
quit;
```

This code assumes that you have already selected a generator.

CONCLUSION

The new random number features in SAS allows you flexibility when running Monte Carlo analyses in SAS Model Implementation Platform. No matter which way of generating numbers you chose, you should always reset the seed in your UDL conditional on the first horizon and the first replication. Otherwise, you might use the same random numbers for every replication.

REFERENCES

- O'Neil, 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. Available <https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf>. Accessed on January 29, 2019.
- O'Neil, M. E. "Critiquing PCG's streams (and SplitMix's too)." Available www.pcg-random.org/posts/critiquing-pcg-streams.html. Last modified August 10, 2017. Accessed on January 29, 2019.
- Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. 2011. "Parallel Random Numbers: As Easy as 1, 2, 3." *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 1-12. Washington, DC: IEEE Computer Society.
- Sarle, W. and Wicklin, R. 2018. "Tips and Techniques for Using the Random-Number Generators in SAS." *Proceedings of the SAS Global Forum 2018 Conference*. Cary, NC: SAS Institute Inc.
- Wicklin, R. "How to Choose a Seed for Generating Random Numbers in SAS." Available <https://blogs.sas.com/content/iml/2017/06/01/choose-seed-random-number.html>. Last modified June 1, 2017. Accessed on January 29, 2019.

ACKNOWLEDGMENTS

The author is grateful to Chris Johns, Shannon Clark, and Don Erdman for their helpful comments and suggestions.

CONTACT INFORMATION

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

Joel Dood
SAS Institute Inc.
Joel.Dood@sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.