

Are Your Random Numbers Created by SAS® Good Enough?

Yanze Zhang, MSD, Shanghai, China

ABSTRACT

Random numbers are crucial to many statistical analyses and simulations. SAS generates pseudorandom numbers based on deterministic algorithms. There are two ways to create random number streams in SAS, Random-number functions and CALL routines. The stream of random numbers is controlled by a seed. If Random-number functions are used in a DATA step to generate the stream, the only seed matters is the first seed in the DATA step. Examples in this paper demonstrate this characteristic of the Random-number functions. This can be easily overlooked by programmers or statisticians, which may lead to the discrepancies in the results. This is actually the advantage of pseudorandom numbers since the stream of random numbers can be reproduced, and thus the results based on those random numbers can be compared and validated. On the other hand, CALL routines can generate multiple streams of random numbers in a DATA step, which may bring some potential risk in stream overlapping when the stream is long. Examples in this paper also demonstrate the loss of independence of streams in some special cases. The Mersenne-Twister pseudo-random number generator (PRNG) used by SAS is also used by many other software, such as C++, R, MATLAB, Stata, Python. This paper also explores the random number generation in R, and the random generation tests in R package "randtoolbox" are carried out to test the quality of the random numbers generated in SAS.

INTRODUCTION

Random number plays an important role in many scientific areas such as computer simulation, statistical sampling and cryptography. It is the cornerstone of randomization widely used in state of the art clinical trials. Historically people used simple device like dice or roulette wheels to generate random numbers. In modern science, much more complex physical phenomenon like radioactive decay and thermal noise has been used. These are known as "True Random Number Generators" (TRNGs). However, the systematic bias may cause the output not uniformly random, and usually it is at the price of efficiency. The alternative method is to use certain algorithm to generate a long sequence of numbers which appear random, but actually they are not truly random. The approach is deterministic, meaning the sequence of numbers are determined by a set of initial values and mathematical formula. So it is known as "Pseudo-Random Number Generators" (PRNGs). Compared with TRNG, PRNG is more efficient and the reproducibility makes it ideal for simulation and modeling application.

According to L'Ecuyer (1990), generally a PRNG is defined by a structure (S, μ, f, U, g) where

- S : a finite set of states,
- μ : a probability distribution on S , called the initial distribution,
- f : a transition function $S \rightarrow S$,
- U : a finite set of output symbols,
- g : an output function $S \rightarrow U$.

Then the steps of generating random numbers are:

1. generate the initial state (called the seed) s_0 according to μ and compute $u_0 = g(s_0)$,
2. iterate for $i = 1, \dots, s_i = f(s_{i-1})$ and $u_i = g(s_i)$.

The most important characteristic of PRNG is its period, which is the number of occurrences before the pseudo-random number sequence repeats. Modern PRNGs have a period so long that it can be ignored for most practical purposes. Since PRNG is a deterministic approach, once the initial states are given, the sequence of numbers or "stream" is always the same. In other words, the pseudo-random numbers are "predictable".

There are two ways to generate pseudo-random numbers in SAS, the random-number functions and CALL routines. The major difference between them is that random-number functions generate single stream within a DATA step, while CALL routines can generate multiple streams within a DATA step. The difference will be illustrated in more detail later in the article. All codes in this article are run using SAS 9.3 and R 2.15.3.

RANDOM-NUMBER FUNCTION IN SAS

There are two types of Random-Number functions in SAS. The older one includes UNIFORM, NORMAL, RANUNI, RANNOR and other RANxxx functions. It is based on a Multiplicative Congruential Random Number Generator which has a period of $2^{31}-1$ (Fishman and Moore 1982). Since version 8.2, a new RAND function is introduced. It is based on a random number generator called “Mersenne-Twister” by Matsumoto & Nishimura (1998). It has excellent statistical properties and very long period of $2^{19937}-1$. That is why it is also known as “MT19937” RNG. The random-number function generates single stream in a DATA step. Please compare the following two data sets from different DATA steps:

```
data example1;
  rannum = ranuni(101);output;
  rannum = ranuni(101);output;
run;

data example2;
  do i = 101 to 110;
    rannum = ranuni(i);
    output;
  end;
  drop i;
run;
```

The first DATA step uses the same seed ‘101’ to generate the 10 random numbers with uniform distribution on [0,1], while the second DATA step use seed ‘101’ to generate the first random number but different seeds for the rest random numbers. But if you compare data set ‘example1’ and ‘example2’, the result is that they are actually identical.

```
The COMPARE Procedure
Comparison of WORK.EXAMPLE1 with WORK.EXAMPLE2
(Method=EXACT)
```

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

Output 1. PROC COMPARE results of data set ‘example1’ vs ‘example2’

This demonstrates that only the first seed in the DATA step matters, since RANUNI function generates single stream in the data. This is a simple fact but sometimes it can be easily overlooked by statisticians and programmers. Here is an example from my own experience. The task was to perform analysis on certain characteristics of coronary artery lesions. Some subjects have more than one lesions and the decision is to randomly select one lesion for analysis. The statistician suggests using a unique ID variable associated with each lesion as the seed. The lesion with the largest random number generated by RANUNI function is selected. However, we couldn’t get a match on the results. It turned out the very first values of the ID variable happened to be different. So actually we generated different streams in the DATA step. The following SAS code creates two data sets with different order of the first two observations.

```
data lesion1;
  input subject $ lesion id val;
  datalines;
12345 1 25 0.10
12345 2 35 0.12
12350 1 70 0.28
12358 1 42 0.31
12370 1 18 0.05
12370 2 15 0.08
12372 1 58 0.24
```

```

12372 2 37 0.19;
run;

data lesion2;
  input subject $ lesion id val;
  datalines;
12345 2 35 0.12
12345 1 25 0.10
12350 1 70 0.28
12358 1 42 0.31
12370 1 18 0.05
12370 2 15 0.08
12372 1 58 0.24
12372 2 37 0.19;
run;

```

Variable 'lesion' is the lesion number. Variable "id" is used as seed of RANUNI function, and the lesion with the maximum value of random number is selected in following code.

```

data lesion_ran;
  set lesion;
  rannum=ranuni(id);
run;

proc sort data=lesion_ran;
  by subject rannum;
run;

data lesion_selected;
  set lesion_ran;
  by subject rannum;
  if last.subject;
  keep subject lesion;
run;

```

The lesion selection results are shown in Table 1. As the table shows, due to the difference of the first observation, different lesions are selected for subject '12372'. This will lead to the discrepancies in the final analysis results. Understanding this intrinsic property of the random-number function will avoid the confusion and save some efforts in such cases.

Lesion1 Result		Lesion2 Result	
subject	lesion	subject	lesion
12345	1	12345	1
12350	1	12350	1
12358	1	12358	1
12370	2	12370	2
12372	1	12372	2

Table 1. Lesion selection results

Different from RANxxx functions, for RAND function to have the reproducible stream a seed needs to be specified by CALL STREAMINIT routine. Here is an example:

```
data example3;
  call streaminit(101);
  do i = 1 to 10;
    rannum = rand('uniform');
    output;
  end;
  drop i;
run;
```

The same seed '101' is used in data set 'example1' and 'example2'. But data set 'example3' is totally different from the two data sets, since as mentioned above, RAND function is based on a different PRNG. If we specify multiple seeds by CALL STREAMINIT in a DATA step, will the RAND function generate different sequence of random numbers? Let's find out in the following example:

```
data example4;
  do i = 1 to 10;
    seed = 100 + i;
    call streaminit(seed);
    rannum = rand('uniform');
    output;
  end;
  drop seed i;
run;
```

Below is the result of comparing data set 'example3' and 'example4':

```

                                The COMPARE Procedure
          Comparison of WORK.EXAMPLE3 with WORK.EXAMPLE4
                                (Method=EXACT)

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

Output 2. PROC COMPARE results of data set 'example3' vs 'example4'

This shows again only the first seed specified in CALL STREAMINIT matters.

GENERATING RANDOM NUMBERS USING CALL ROUTINES

Another way to generate pseudo-random number in SAS is random-number CALL routine. The major difference between random-number CALL routines and random-number functions is that CALL routines can generate multiple streams in a DATA step. Each time the CALL routine is executed a new value of seed is returned. This gives random-number CALL routines greater control of the seed and the stream than the random-number function. Please note the seed value of random-number function is also updated each time the function is executed, but it is updated internally and we can't trace it. We can only see the initial seed that we assign at beginning. Please look at the following example:

```
data example5;
  seed=101;
  do i = 1 to 10;
    call ranuni(seed, rannum);
    output;
  end;
  drop i;
run;
```

The initial seed is the same one used in data set 'example1' and 'example2'. Thus, we expect the random numbers are also the same. As you can see in data set 'example5', variable 'seed' keeps the value of the seed during the execution.

Obs	seed	rannum
1	1462907848	0.68122
2	2102301194	0.97896
3	820886179	0.38225
4	427914427	0.19926
5	175793794	0.08186
6	1926319299	0.89701
7	1805811457	0.84090
8	1505093989	0.70086
9	62699322	0.02920
10	1553106799	0.72322

Output 3. Data set 'example5'

The next example will show you what happens if we have two initial seeds for random-number CALL routine in a DATA step.

```
data example6;
  seed1=100;
  seed2=150;
  do i = 1 to 10;
    call ranuni(seed1,rannum1);
    call ranuni(seed2,rannum2);
    output;
  end;
  drop i;
run;
```

Obs	seed1	seed2	rannum1	rannum2
1	1065703754	1598555631	0.49626	0.74439
2	19051541	1102319135	0.00887	0.51331
3	2109753667	2090888677	0.98243	0.97365
4	2018343725	1953773764	0.93986	0.90980
5	344150976	516226464	0.16026	0.24039
6	1992295688	840959885	0.92773	0.39160
7	639772562	959658843	0.29792	0.44688
8	363294709	1618683887	0.16917	0.75376
9	2103251112	1007393021	0.97940	0.46910
10	1410156218	2115234327	0.65666	0.98498

Output 4. Data set 'example6'

In data set 'example6' there are two different streams generated. If we create the scatter plot of variable 'rannum1' and 'rannum2' we will find the two streams seem not completely independent. If we increase the length of the stream to 1000, it is very obvious that the two streams are related, as Figure 1 shows.

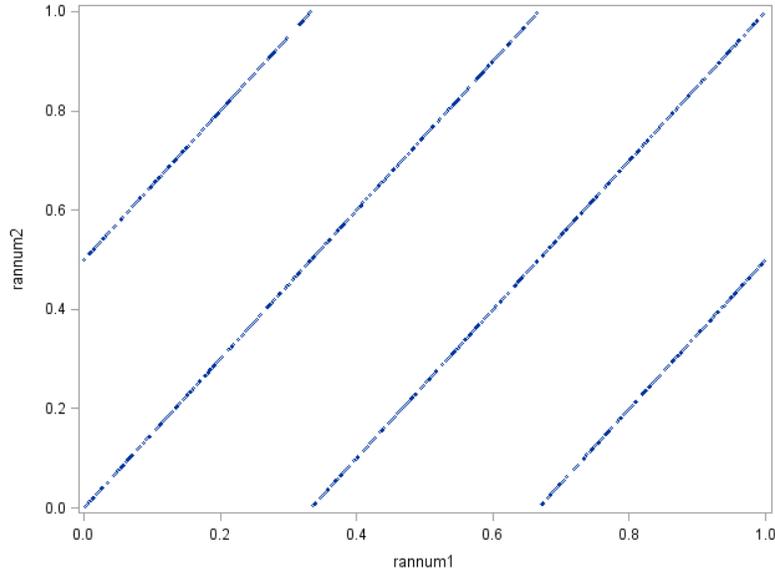


Figure 1. 'rannum1' vs 'rannum2'

This is caused by the limitation of the random number generator used by RANxxx functions. The worse situation is that the streams may overlap when the streams get very long. The SAS Knowledge Base “Using Random-Number Functions and CALL Routines” page provides an extreme case with deliberately selected initial seeds, in which the streams almost overlap completely. So the suggestion is to avoid creating independent multiple variables using multiple streams by random-number CALL routines. Readers can find more examples on “Using Random-Number Functions and CALL Routines” page on how to effectively use random-number CALL routines to control the stream.

RANDOM NUMBER GENERATION IN R

The Mersenne-Twister pseudo-random number generator used by RAND function in SAS is actually used by many other software, such as C++, R, MATLAB, Stata, Python. R is a very popular statistical computing software among academic researchers. The default pseudo-random number generator in R is also Mersenne-Twister. The seed can be initialized by the following command:

```
> set.seed(101)
```

If no seed is specified, one is created from current time and process ID. The following command generates 10 random numbers from uniform distribution on [0, 1]

```
> runif(10)
[1] 0.37219838 0.04382482 0.70968402 0.65769040 0.24985572 0.30005483 0.58486663
0.33346714 0.62201196 0.54582855
```

The random numbers from other distributions are generated by other functions like 'rnorm' (normal distribution), 'rbinom' (binomial distribution), etc. They are all converted from random numbers of uniform distribution by certain algorithm.

TESTING SAS RANDOM NUMBER GENERATOR IN R

Besides period, the other key characteristics of a good random number generator are that it has no predictability, no dependencies and the random numbers are uniformly distributed. We can evaluate the random number generator by testing if the random numbers could be considered as independent and identically distributed uniform variates at a given confidence level. Many random number generator testing methods have been developed and among them the DIEHARD Battery of Tests developed by Marsaglia (1996) and the statistical test suite from the National Institute of Standards and Technology (NIST) are considered the “gold standard” methods. They perform tests on binary set {0,1} and the input requires binary data files of integers. If these methods are used to test the random numbers generated by SAS, a lot of conversion need to be done first. SAS doesn't have built-in functions or procedures to perform random number tests. Fortunately there are some R packages available for use. There is an R package “RDieHarder” which performs a suite of random number tests as an extended work of the original “Diehard” tests, but it only works under GNU systems. Due to the limited conditions, and to test the random numbers on interval [0,1], five statistical tests included in R package “Randtoolbox” are carried out on the random numbers generated by RAND and RANUNI function in SAS. The five tests are the gap test, the order test, the frequency test, the serial test and the

poker test. The details of the five tests can be found in the related R package documentation. The link is provided in “RECOMMENDED READING” part of this paper.

1000 sequences of random numbers with a length of 10000 for each sequence are generated by RAND and RANUNI function respectively. The machine time is used as the initial seed. The SAS data sets of random numbers are exported to R through SAS/IML® interface. After the R codes are executed there, the R data frame is imported to SAS as SAS data set. Then the significance level of 0.01 is selected which is used by the NIST Special Publication 800-22 (2001). If the p-value from each test is lower than 0.01 the null hypothesis that the random numbers are truly random is rejected, otherwise the sequence of random numbers pass the test successfully. Since the five tests do not compose a suite like the “Diehard” tests which has a systematical design, no inference on passing all five tests as a whole is drawn. The passing ratio calculated from the 1000 sequences for each test is summarized and the results of RAND and RANUNI function are compared as following.

RNG Test	Passing Ratio
Gap Test	89.2%
Order Test	99.2%
Frequency Test	99.2%
Serial Test	98.9%
Poker Test	99.1%

Table 2. Test Results of Random Numbers Generated by RAND Function

RNG Test	Passing Ratio
Gap Test	88.2%
Order Test	99.1%
Frequency Test	98.8%
Serial Test	98.8%
Poker Test	99.1%

Table 3. Test Results of Random Numbers Generated by RANUNI Function

As Table 2 and 3 show, the passing ratio is very high for both RAND and RANUNI except Gap test. The ratios for RAND function are higher than the ones for RANUNI function, which shows the more advanced Mersenne-Twister RNG used by RAND function does have better performance than the Multiplicative Congruential RNG used by RANUNI function. According to Rotz, W.(2004), RAND function passed all of their NIST and DIEHARD tests, while RANUNI function failed both suites of the tests in some trials. The SAS code of data preparation and analysis is not included in this paper and it is available upon request.

CONCLUSION

Random numbers are the foundation of modern cryptography, simulation, sampling and many other fields. Pseudo-RNGs generate random numbers using deterministic formulas. Compared with True RNGs, Pseudo-RNGs have the advantages of being more efficient and reproducible. SAS generate Pseudo-random numbers using random-number functions and CALL routines. It is important to keep in mind that only the first seed determines the random number stream generated by random-number function in a DATA step. For RNGs used by RANUNI function or CALL routines there is no guarantee that the multiple streams are independent, and there is even potential risk of overlapping of streams. To further test the quality or randomness of the random numbers generated by RAND and RANUNI function in SAS, the statistical tests in R package “randtoolbox” are performed on the sequences. The result shows that RAND function has better performance than the RANUNI function. Thus, RAND function is highly recommended instead of RANUNI function as a trustworthy PRNG in SAS.

REFERENCES

1. L'Ecuyer, P. (1990), “Random numbers for simulation”, Communications of the ACM 33, 85-98.

2. Fishman, G.S. and Moore, L.R. (1982), "A Statistical Evaluation of Multiplicative Congruential Generators with Modulus (231 - 1)," *Journal of the American Statistical Association*, 77, 129-136.
3. Matsumoto, M. and Nishimura, T. (1998), "Mersenne twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator", *ACM Trans. On Modelling and Computer Simulation* 8(1), 3-30.
4. Marsaglia, G. (1996), "The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness", available at <http://stat.fsu.edu/pub/diehard>
5. NIST, Special Publication 800-22 (2001), "A Statistical Test Suite for Random and Pseudo-random Number Generators for Cryptographic Applications", available at <http://csrc.nist.gov/publications/PubsSPs.html>
6. Rotz, W. and Falk, E. (2004), "A Comparison of Random Number Generators Used in Business - 2004 Update", *ASA Section on Business & Economic Statistics Section*.

RECOMMENDED READING

- "Using Random-Number Functions and CALL Routines", available at <http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a001281561.htm>
- R Package "randtoolbox", available at <https://cran.r-project.org/web/packages/randtoolbox/index.html>

CONTACT INFORMATION

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

Name: Yanze Zhang
Enterprise: MSD R&D (China) Co., Ltd.
Address: No. 1582 Gumei Rd.
City, State ZIP: Shanghai, 200233
Work Phone: +86 21 2211 8538
Fax: +86 21 2211 8899
E-mail: yan.ze.zhang@merck.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.