Computational Statistics and Data Analysis 52 (2008) 4587–4593

Contents lists available at ScienceDirect

Computational Statistics and Data Analysis

journal homepage: www.elsevier.com/locate/csda

Microsoft Excel's 'Not The Wichmann–Hill' random number generators

B.D. McCullough

LeBow College of Business, Department of Decision Sciences, Drexel University, 19104 Philadelphia, PA, United States

ARTICLE INFO

Article history:
Available online 13 March 2008

ABSTRACT

Microsoft attempted to implement the Wichmann–Hill RNG in Excel 2003 and failed; it did not just produce numbers between zero and unity, it would also produce negative numbers. Microsoft issued a patch that allegedly fixed the problem so that the patched Excel 2003 and Excel 2007 now implement the Wichmann–Hill RNG, as least according to Microsoft. We show that whatever RNG it is that Microsoft has implemented in these versions of Excel, it is not the Wichmann–Hill RNG. Microsoft has now failed twice to implement the dozen lines of code that define the Wichmann–Hill RNG.

1. Introduction

The desiderata for a (pseudo-)random number generator (RNG) are simple (Ripley, 1990):

1. produce numbers that are approximately uniform;
2. produce numbers that are approximately independent in a moderate number of dimensions;
3. have a sufficiently long period; and
4. be reproducible from a simply-specified starting point but, unless a starting point is specified, should be unpredictable.

Point (1) has to do with the quality of the random numbers (RNs). If the RNs are not uniform, then a simulation that requires uniform input will not yield the correct conclusion. For example, a simple univariate Monte Carlo integration on the interval [0, 1] might produce an incorrect answer if the RNG's output, that is used as input to the integration process, is not uniform on [0, 1]. Point (2) also has to do with the quality of the RNs. Truly random numbers are uncorrelated across dimensions; if pseudo-random numbers are correlated across dimensions then an incorrect answer can be produced. For example, if a particular RNG produces uniform RNs but these RNs are correlated in two dimensions, it might produce an accurate answer for a univariate Monte Carlo integration, but an inaccurate answer for a bivariate Monte Carlo integration. To determine whether an RNG satisfies Points (1) and (2), theoretical and empirical considerations must be entertained. See L'Ecuyer (1994, Section 2.2) for a discussion of theoretical properties of an RNG. Empirical properties are assessed using a collection of randomness tests. A decade ago, the primary collection of tests was Marsaglia's (1996) DIEHARD suite of randomness tests. However, it was later supplanted by a much better program, TESTU01 (L'Ecuyer and Simard, 2007), which was first released in 2002. TESTU01 offers three batteries of tests. If an RNG passes the "Small Crush" battery (which takes a few minutes of CPU time) then apply the "Crush" battery (which can take a few hours) and if it passes all those, apply "Big Crush" (which can take an entire day). The DIEHARD tests, in contrast, take only several seconds to run on a modern PC. Since its release, TESTU01 has been the preferred program for empirical testing of RNGs since it can uncover flaws in RNGs that pass other suites of tests (Gentle, 2003, p. 85). See McCullough (2006) for a review of TESTU01.

Point (3) is important because Points (1) and (2) are generally valid only for a fraction of the period length, so if one wants to generate a million RNs, an RNG with a period of several million should be used. L'Ecuyer (1992, p. 306) warned that "No generator should be used for any serious purpose if its period (or, at least, a lower bound of it) is unknown." Over ten years

E-mail address: bdmccullough@drexel.edu.

0167-9473/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.csda.2008.03.006

Table 1
First five values from VBA RNG

	VBA RN	VBA RN * 2^{24}
1	0.705547511577606	11837123
2	0.533424019813537	8949370
3	0.579518616199493	9722709
4	0.289562463760376	4858052
5	0.301948010921478	5065847

ago the minimum accepted period length for an RNG was 2^{60} (L'Ecuyer, 1994). For example, Ripley (1990) suggests that the period (p) should be an order of magnitude larger than the number of calls (n) to the RNG, $p \gg 200n^2$. Finally, Point (4) is important for purposes of debugging and testing, as well as reproducibility. This is typically effected by allowing the user to "set the seed" of the RNG, so that every time the same seed is set, the result is that the same sequence of RNs is produced by the RNG.

McCullough and Wilson (1999) criticized Excel 97, as well as Excel 2000 and Excel XP (McCullough and Wilson, 2002), for failing to meet any of these standards. Microsoft, with Excel 2003, attempted to remedy 1–3 by implementing the Wichmann and Hill (1982) (WH) RNG, whose 37 lines of code (23 of which are comments) are presented in the Appendix to this paper.[1] As noted by McCullough and Wilson (2005), Microsoft failed to implement correctly the WH RNG: instead of producing numbers between zero and unity, Excel would eventually produce negative numbers. Since the WH RNG cannot produce negative numbers, it was fair to conclude that whatever RNG Microsoft implemented in Excel 2003, it was not the WH RNG. Microsoft never disclosed the actual algorithm that was implemented or the reason that its implementation produced negative numbers (a seeming impossibility if one examines the WH code). We shall refer to this unknown RNG as "Microsoft Excel's Not The Wichmann–Hill - 1" (MENTWH1) since its true algorithm and other properties are unknown. With respect to MENTWH1, Microsoft could not achieve the desiderata for an RNG.

Microsoft soon issued a patch for Excel (see Microsoft Knowledge Base Article 834520), claiming that the patched version of Excel 2003 implements the WH RNG; further, this same RNG exists in Excel 2007 (see Microsoft Knowledge Base Article 828795). We show that this RNG also is not the WH RNG — Microsoft has failed again to implement correctly the dozen lines of code that constitute the WH RNG. We shall refer to this new Excel RNG as MENTWH2. In Section 2 we show one way to verify that an RNG has been correctly implemented. In Section 3 we show how this method can be applied to the WH RNG. In Section 4 we show the results of applying this method to the WH and other RNGs, including the MENTWH2 RNG in Excel 2007. Section 5 offers the conclusions.

2. How to check an RNG

Given a pseudo-random number generator whose period is less than about 2^{60} and the values produced in the range 0.0 to 1.0 are given in double precision, then it may well be possible to predict the sequence from just a few values. For a simple example, consider the RNG in Microsoft Visual Basic which is not a good RNG. It fails all the tests in Small Crush, and L'Ecuyer and Simard (2007, p. 30) refer to it as a "toy generator". However, its ubiquity makes it useful for illustrative purposes. It is given by the relationship:

$$X_{n+1} = (1140671485 \times X_n + 12820163) \bmod 2^{24}. \tag{1}$$

The value of the random number (in the range 0–1) is given by $X/2^{24}$; hence we can recover the value of X_n from the floating point value given. This depends upon the floating point values having at least 24 bits of accuracy.

Since VBA always uses the same initial value (*i.e.*, seed) $X_0 = 327\,680$, it always produces the same string of numbers. The first five numbers produced by this algorithm are presented in Table 1.

To verify that the algorithm in (1) is used to produce these numbers, we apply the following steps:

1. multiply the first uniform RN, 0.705547511577606, by the modulus (2^{24}) to obtain 11837123
2. substitute 11837123 for X_n in (1) to obtain 8949372
3. divide 8949372 by 2^{24} to obtain the second RN, 0.533424139.

Note that this answer agrees with the VBA answer to only six digits. One possible reason is that our calculations have been in double precision while VBA RNs apparently are produced via single precision.

This procedure for determining whether a sequence of RNs is generated by (1) will work for any two successive RNs produced by the RNG given by (1). The reader can easily verify that it applies to the 999th and 1000th RNs from the VBA RNG, which are 0.311508715152740 and 0.467859745025635, respectively. However, when this method is applied to the WH RNG, even double precision will not be sufficient to achieve more than a couple of digits of accuracy, and so specialized computation methods will be necessary.

[1] This RNG is not to be confused with the modern, long-period version of the WH RNG described in Wichmann and Hill (2006).

3. The Wichmann–Hill RNG

Wichmann and Hill (1982) published a random number generator with an approximate period of 2^{43} that became very popular. It uses a combination of three linear congruential RNGs to obtain a uniform random number W. First, the "seeds" I_X, I_Y and I_Z are initialized to take on integer values between 1 and 30,000. Then the following equations are executed to advance the seeds and compute W:

$I_X := 171 \times (I_X \bmod 177) - 2 \times (I_X \div 177)$;
$I_Y := 172 \times (I_Y \bmod 176) - 35 \times (I_Y \div 176)$;
$I_Z := 170 \times (I_Z \bmod 178) - 63 \times (I_Z \div 178)$;
if $I_X < 0$ **then**
    $I_X := I_X + 30\,269$;
if $I_Y < 0$ **then**
    $I_Y := I_Y + 30\,307$;
if $I_Z < 0$ **then**
    $I_Z := I_Z + 30\,323$;
$W := I_X/30269.0 + I_Y/30307.0 + I_Z/30323.0$;
return $W - \lfloor W \rfloor$;

If users could set the seeds, it would be an easy matter to compute successive values of the WH RNG and thus ascertain whether Excel is correctly generating WH RNGs. We pointedly note that Microsoft programmers obviously have the ability to set the seeds and to verify the output from the RNG; for some reason they did not do so. Given Microsoft's previous failure to implement correctly the WH RNG, that the Microsoft programmers did not take this easy and obvious opportunity to check their code for the patch is absolutely astounding. Since Microsoft does not enable users to set the seeds, computing successive values is problematic as it requires inverting the above equations; that is, finding I_X, I_Y and I_Z from the values of W. What makes this problem difficult is the rounding error introduced in the floating point calculation of W. Fortunately, reverse-engineering the RNG is not necessary to determine whether a sequence of RNs has been produced by the WH RNG.

Zeisel (1986) has shown that the WH RNG can be written as a simple linear congruence generator in the form:

$$X_{n+1} = 16555425264690 \times X_n \bmod 27817185604309 \tag{2}$$

which we will call the Zeisel recursion. An implementation of the generator would not actually compute the advancing of the seed using this formula because it would overflow on most standard computers, but this does not matter to us, as the above equation is more than suitable for present purposes. The "seed" X_n can be recovered from the random value by multiplying the random value by the modulus, 27817185604309. Then the Zeisel recursion can be used to compute X_{n+1} which, divided by the modulus, yields the next (tentative) random value. This tentative next random value can be compared to the actual next random value and, if they are the same, it can be safely concluded that the RNs are produced by the WH RNG. If the next tentative random value and the next actual random value differ, then it can be concluded that the RNs are not produced by the WH RNG.

The difficulty with using the Zeisel recursion to compute random values is that it requires long integer calculations. However, these can easily be undertaken using a simple Python program since the Python language handles large integers. The slight complication in this is that the first guess at the "seed" will be an approximation to the exact value due to the rounding of the floating point operations. We can allow for this by trying a few integer values either side of the initial guess that is formed by the product of the random value and the modulus.

4. Results

To test the efficacy of the program, in a single-blind experiment, the author sent seven strings of RNs to a colleague, who was to use the program and report back whether the RNs were from a WH RNG or not. The set of strings included three WH RNG strings (one from *R* and two hand-coded in different languages) and four other RNGs. In all cases, the program correctly identified each string of RNs as being from a WH RNG or some other RNG. To demonstrate the program's methodology, Table 2 includes three sets of RNs, a Wichmann–Hill from *R*, Marsaglia's Super-Duper from *R*, and the Microsoft Excel 2007 RNG. We are, quite sure that the RNs from "*R*" are double precision. Microsoft claims to perform calculations in double precision (Microsoft Knowledge Base Article 78113), and we will take Microsoft at its word, though Kahan (2006) has some interesting observations on this point.

The interested reader can verify our RNs from *R* version 2.4.0, for which we used the following code:

```
set.seed(123,kind="Wichmann--Hill")
#set.seed(123,kind="Super-Duper")
x <- runif(5)
format(as.data.frame(x),digits=15)
```

Table 2
Random numbers from various generators

Row	WH from *R*	Super-Duper from *R*	MENTWH2
1	0.4462944498771981	0.5741969865221056	0.602128391977151
2	0.8905663455043975	0.9862140465961334	0.024490859665315
3	0.1452252163781164	0.9770346416572653	0.191836077065749
4	0.3407396508907700	0.7969776286736543	0.958234220735553
5	0.7031855932854678	0.9892580336866101	0.854492632692052

As Microsoft does not permit a seed to be set for the Excel RNG, there can be no audit trail for Excel RNs, and the reader will just have to trust us that the numbers we present come from Excel 2007.

4.1. WH RNG from R

As a preliminary, we observe that using double precision on a standard PC does not provide sufficient accuracy. First we attempt to recover the seed by multiplying an RN by the modulus,

$$0.4462944498771981 \times 27817185604309 = 12414655546407.$$

Maybe this tentative seed is corrupted by rounding error; if so we will have to try nearby values (in fact this is the correct seed). Next we use this value as X_n in the Zeisel recursion and form the product $16555425264690 \times 12414655546407$ which cannot be accurately computed using standard methods on a PC. Standard methods can only handle sixteen digits before resorting to scientific notation, and so for this product we get $2.055299020854103E26$ (instead of the correct answer 205529902085410284553468830). Consequently, applying the *mod* operation to this incorrect product will not produce a very accurate answer, but let us do so:

$$2.055299020854103E26 \bmod 27817185604309 = 24780451348480$$

whereas the correct answer is:

$$205529902085410284553468830 \bmod 27817185604309 = 24773049325847.$$

We divide the incorrect result by the modulus in an attempt to obtain the next RN in the sequence, $24780451348480/27817185604309 = 0.890832440814624$ which is accurate only to three digits (as opposed to the correct answer, which is $24773049325847/27817185604309 = 0.8905663455043975$). Three digits of accuracy is insufficient for present purposes. Python can effect all the calculations quite accurately, and we should be able to reproduce several digits of a WH RN. Below is some output from running the program. It first calculates a tentative seed of 12414655546406, rejects it, and then tries ten values on either side of the first guess (we show only four on either side), eventually settling on 12414655546407:

```
seed 12414655546402 fail -4
seed 12414655546403 fail -3
seed 12414655546404 fail -2
seed 12414655546405 fail -1
seed 12414655546406 fail 0
seed 12414655546407 pass 1
seed 12414655546408 fail 2
seed 12414655546409 fail 3
seed 12414655546410 fail 4
```

Having identified the correct seed, the program then computes the Zeisel recursion a few times:

```
Starting seed value:  12414655546407
0.890566345504
0.145225216378
0.340739650891
0.703185593285
0.261453152604
```

Comparing these numbers to the first column of Table 2 shows, to no great surprise, that the implementation of the WH RNG in *R* is correct.

4.2. Super-duper RNG from R

We undertake exactly the same steps as in the previous subsection. First we attempt to recover the seed by multiplying an RN by the modulus, $0.5741969865221056 \times 27817185604309 = 15972544147520.3$ which we round to 15972544147520. We tried 10 values either side as the seed (though we show results for only four), but each of these possible seeds failed to be consistent with the Zeisel recursion.

```
seed 15972544147516 fail -4
seed 15972544147517 fail -3
seed 15972544147518 fail -2
seed 15972544147519 fail -1
seed 15972544147520 fail 0
seed 15972544147521 fail 1
seed 15972544147522 fail 2
seed 15972544147523 fail 3
seed 15972544147524 fail 4
```

This is to be expected because these Super-Duper RNs are not generated by the Zeisel recursion. Hence we conclude, to no great surprise, that Marsaglia's Super-Duper RNG does not produce WH RNs.

4.3. MENTWH2 RNG

We repeat the same steps as before, multiplying the first random value by the modulus: $0.602128391977151 \times 27817185604309 = 16749517237252.53$ which we round to 16749517237252. Checking ten values (though we show only four) on either side of this produces no seed that is consistent with the Zeisel recursion.

```
seed 16749517237248 fail -4
seed 16749517237249 fail -3
seed 16749517237250 fail -2
seed 16749517237251 fail -1
seed 16749517237252 fail 0
seed 16749517237253 fail 1
seed 16749517237254 fail 2
seed 16749517237255 fail 3
seed 16749517237256 fail 4
```

Thus, to great surprise we show that Excel's the so-called WH RNG does *not* produce WH RNs; to wit: Microsoft has again failed to implement correctly the WH RNG. Hence we call it not the WH RNG, but MENTWH2 RNG.

A referee pointed out that perhaps our inability to locate the starting point for the generator was due to problems with the rounding in Excel. Kahan (2006) has documented some rounding problems, but in our case, no actual calculation is being performed in Excel, just the output of the random values. To be sure that the rounding to 15 digits has not caused any problem, the program was re-run by trying over 1000 possible values for a match, without success.

5. Conclusions

Twice Microsoft has attempted to implement the dozen lines of code that define the Wichmann and Hill (1982) RNG, and twice Microsoft has failed, apparently not using standard methods for verifying that an RNG has been correctly implemented. Consequently, users of Excel's "rand" function have been using random numbers from an unknown and undocumented RNG of unknown period that is not known to pass any standard tests of randomness. Given Microsoft's first failure to implement the WH RNG correctly, Microsoft should have taken special care to ensure that it was done correctly the second time. Microsoft did not do this. The second failure demonstrates clearly that the first failure (and the second, as well) was not some sort of "unfortunate accident" but, rather, a systemic failure on Microsoft's part.

Of course, one has to wonder why Microsoft chose an antiquated RNG instead of a modern RNG. In the first place, its period is only 2^{43} which, following Ripley's (1990) suggestion, can only support a couple of hundred thousand calls to the RNG. Further, Microsoft appealed to Marsaglia's (1996) DIEHARD suite of randomness tests (see Microsoft Knowledge Base Article 828795) instead of the much more stringent tests offered by L'Ecuyer and Simard's (2007) TESTU01 which has been available since 2002. Perhaps TESTU01 was released too late for Microsoft to make changes in Excel 2003, but certainly it was released in time for Microsoft to make changes in Excel 2007.

Modern empirical testing of RNGs requires that the RNG be programmed into the testing software; it is not practical to make a file of RNs and read them into the software. Since Microsoft has not made available the algorithm for MENTWH2, we cannot subject it to empirical testing. Hence, whether MENTWH2 passes any modern battery of randomness tests is unknown.

It will be interesting to see what Microsoft chooses to do. Will Excel users have to wait until Excel 2010 to get good random numbers, or will Microsoft offer a patch for Excel 2007? Will Microsoft once again attempt to implement the Wichmann–Hill RNG and, if so, will the third time be a charm? Or will Microsoft offer an RNG more suitable to the 21st century and, if so, will Microsoft program it correctly? Whatever Microsoft chooses to do, it should ensure that users have the ability to verify the RNG, since Microsoft has demonstrated that users cannot rely on Microsoft's claims to have done it correctly.

In the meantime, Excel users will have to content themselves with an unknown RNG of unknown period that is not known to pass any standard battery of tests for randomness. At least now, users will not mistakenly believe that they are using WH RNs. Finally, Microsoft should do Messrs. Wichmann and Hill the courtesy of taking their names off the RNG in Excel.

Acknowledgements

Thanks to two referees for the extremely useful reports, to Ian Smith and Jerry Lewis for comments, and to various colleagues for assistance.

Appendix. The Wichmann–Hill code

```
      real function random()
c
c     Algorithm AS 183 Appl. Statist. (1982) vol.31, no.2
c
c     Returns a pseudo-random number rectangularly
c     distributed between 0 and 1.
c
c     IX, IY and IZ should be set to integer values
c     between 1 and 30000 before the first entry.
c
c     Integer arithmetic up to 30323 is required.
c
      integer ix, iy, iz
      common /randc/ ix, iy, iz
c
      ix = 171 * mod(ix, 177) - 2 * (ix / 177)
      iy = 172 * mod(iy, 176) - 35 * (iy / 176)
      iz = 170 * mod(iz, 178) - 63 * (iz / 178)
c
      if (ix .lt. 0) ix = ix + 30269
      if (iy .lt. 0) iy = iy + 30307
      if (iz .lt. 0) iz = iz + 30323
c
c     If integer arithmetic up to 5212632 is available, the
c      the preceding 6 statements may be replaced by:
c
c     ix = mod(171 * ix, 30269)
c     iy = mod(172 * iy, 30307)
c     iz = mod(170 * iz, 30323)
c
c        on some machines, this may slightly increase
c        the speed.  the results will be identical.
      random = amod(float(ix) / 30269.0 + float(iy) / 30307.0 +
     *              float(iz) / 30323.0, 1.0)
      return
      end
```

References

Gentle, J.E., 2003. Random Number Generation and Monte Carlo Methods, 2nd ed. Springer-Verlag, New York.
Kahan, W., 2006. How futile are mindless assessments of roundoff in floating-point computation? http://www.cs.berkeley.edu/~wkahan/Mind1ess.pdf.
L'Ecuyer, P., 1992. Testing random number generators. In: Swain, J.J., Crain, R.C., Wilson, J.R. (Eds.), Proceedings of the 1992 Winter Simulation Conference. IEEE Press, NY, pp. 305–313.
L'Ecuyer, P., 1994. Uniform random number generation. Annals of Operations Research 53, 77–120.
L'Ecuyer, P., Simard, R., 2007. TESTU01: A C library for empirical testing of random number generators. ACM TOMS 33 (4), Article 22.
Marsaglia, G., 1996. DIEHARD: A battery of tests of randomness. http://stat.fsu.edu/pub/diehard.

McCullough, B.D., 2006. A review of TESTU01. Journal of Applied Econometrics 21 (5), 677–682.
McCullough, B.D., Wilson, B., 1999. On the accuracy of statistical procedures in microsoft excel 97. Computational Statistics & Data Analysis 31 (1), 27–37.
McCullough, B.D., Wilson, B., 2002. On the accuracy of statistical procedures in microsoft excel 97. Computational Statistics & Data Analysis 40 (4), 713–721.
McCullough, B.D., Wilson, B., 2005. On the accuracy of statistical procedures in microsoft excel 2003. Computational Statistics & Data Analysis 49 (4), 1244–1252.
Ripley, B.D., 1990. Thoughts on pseudorandom number generators. Journal of Computational and Applied Mathematics 31, 153–163.
Wichmann, B.A., Hill, I.D., 1982. Algorithm AS 183: An efficient and portable pseudo-random number generator. Applied Statistics 31 (2), 188–190. Reprinted, with a correction. in Hill I.D., Griffiths P. (Eds.), Applied Statistics Algorithms, Ellis Horwood, Chichester, 1985.
Wichmann, B.A., Hill, I.D., 2006. Generating good pseudo-random numbers. Computational Statistics & Data Analysis 51 (3), 1614–1622.
Zeisel, H., 1986. Remark on algorithm AS 183. Applied Statistics 35, 89.