Middle Square Weyl Sequence RNG

Bernard Widynski[1]

May 20, 2020

Abstract

In this article, we propose a new implementation of John von Neumann's middle square random number generator (RNG). A Weyl sequence keeps the generator running through a long period.

1 Introduction

Early in the field of computer science, John von Neumann proposed the middle square method for generating random numbers [10]. A number is squared and the middle digits are returned as the next random number. This method was used as a source of data in the early work on Monte Carlo. Good data was produced but the middle square had a known problem which was referred to as the "zero mechanism". Once the middle digits became zero the generator would continue to produce zero outputs. Repeating cycles of non-zero data could also be produced. In this article we propose the use of a Weyl sequence to run the middle square. The Weyl sequence overcomes the "zero mechanism". It keeps the generator running through a long period.

2 Weyl Sequence

We use a Weyl sequence similar to the Weyl sequence in Brent's xorgens RNG [1]. This is actually an integer stepping sequence of period 2^{64}. In C, it has the following form: w += s; where w is an unsigned 64-bit integer and s is an odd constant. A middle square RNG run by this sequence is shown below:

```c
uint64_t x = 0, w = 0, s = 0xb5ad4eceda1ce2a9;
inline static uint32_t msws() {
   x *= x; x += (w += s); return x = (x>>32) | (x<<32);
}
```

The Weyl sequence is added to the square of x. The middle is extracted by shifting right 32 bits.

3   Middle Square Implementation

One might ask "Where is the middle?" in the above program. In C, arithmetic on unsigned 64-bit integers is modulo 2^{64}. After squaring, the x value will have only the lower 64 bits of the result. These 64 bits represent the lower half of the square. Shifting right 32 bits will return the upper part of this lower half. This is the 32-bit middle that is returned.

Perhaps an illustration might be useful. We will consider only the square and shift operations. Let's begin by showing how John von Neumann's original middle square would obtain a middle from a square. Let's square a 16-digit hexadecimal number, say E3296D171EC4A36F. The square is

$$C9927E2B2075471D31C2914AAE4E8A21$$

The middle is
$$2075471D31C2914A$$

Our computer, however, is only able to store 64-bit numbers. That is enough for 16 hexadecimal digits. So, after squaring x will store only the lower 16 digits of the 32 produced:

$$31C2914AAE4E8A21$$

The circular shift operation will transform this to

$$AE4E8A2131C2914A$$

The return statement will provide the lower 32 bits or 8 digits:

$$31C2914A$$

Notice that this is the lower half of the middle from the original middle square. We are in effect returning middle digits from a square. The modulo 2^{64} arithemetic implicit in C allows us to work with larger squares than would otherwise be possible. It is not necessary to store all 32 digits of the result. It is sufficient to use the lower 16 that are computed. This half square implementation is quite efficient and allows for an easy extraction of the middle by a simple shift operation. The RNG compiles into only four machine instructions: imulq, iaddq, iaddq, and rorq. The rorq is the rotate right instruction which corresponds to a circular shift. Being so compact, the RNG is suitable for inline which increases efficiency by removing the subroutine linkage.

4 Overcoming the Zero Mechanism

The primary defect with the middle square is that it eventually produces repeating cycles of data, often a cycle of zero outputs. In this section we demonstrate that the Weyl sequence w prevents repeating cycles in x. We will show this in two steps. First we demonstrate that no two of the first 2^{64} elements of the Weyl sequence w can be equal. It follows from this that there can be no repeating cycles in x.

Theorem A: For the Weyl sequence that is generated by the formula w += s where w is an unsigned 64-bit integer and s is an odd constant, no two of the first 2^{64} generated elements can be equal.

Method: This can be shown with a proof by contradiction. The assumption that any two are equal leads to the contradiction that odd is equal to even.

Proof: The mathematical equivalent of w += s in C is shown below:

For i = 0 to $2^{64} - 1$, and some odd constant s $< 2^{64}$

$$w(i) = (i * s) \ mod \ 2^{64}$$

Now, assume that there exists an m and n with m < n < 2^{64} such that

$$w(m) = w(n)$$

This implies that

$$n * s - m * s = k * 2^{64} \qquad \text{for some integer k}$$

which implies that

$$(n - m) * s = k * 2^{64}$$

There are two cases to be considered.

Case 1: (n - m) is odd

Here we have odd times odd equal to some k times even, which implies that odd is equal to even.

Case 2: (n - m) is even

Let $2^j * p$ be the factorization of (n - m) for some integer j and odd in-

teger p. This will give

$$2^j * p * s = k * 2^{64}$$

which is equivalent to

$$p * s = k * 2^{64-j}$$

Note that $0 < j < 64$ because (n - m) is even and (n - m) is $< 2^{64}$

which implies that odd equals to even.

Q.E.D.

We conclude that no two of the first 2^{64} numbers can be equal.

From Theorem A it can be determined that the period of the Weyl sequence is 2^{64}. If no two of the first 2^{64} numbers are equal then all must be different. The only way this could happen is if each number between 0 and $2^{64} - 1$ occurs exactly one time. Also, from $w(i) = (i * s) \bmod 2^{64}$ we see that the next 2^{64} numbers generated will be exactly the same sequence as the first 2^{64} numbers generated and for all 2^{64} subsequently generated. It follows that the period of the Weyl sequence is 2^{64} and that the output of the Weyl sequence will be uniform.

Theorem B: For the first 2^{64} iterations of the RNG, there will be no repeating cycles in x.

Method: This can be shown quite easily by noting that the following Weyl sequence value for any two equal x values will be different (Theorem A), thus producing a different x value.

Proof: Consider two separate x values in the first 2^{64} iterations that are equal. The RNG performs three operations: square, add Weyl sequence, and shift. The first operation is the square and this will yield the same result. The next operation is to add the Weyl sequence value w which we know from Theorem A is different. So the result will be different. The third operation is the circular shift. The numbers will still be different after shifting. This implies that the next x value after each of the equal x values will be different. We conclude that there can be no repeating cycles in x for the first 2^{64} iterations of the RNG.

Q.E.D.

That there are no repeating cycles in the the first 2^{64} interations implies that the period of x will be at least 2^{64}.

5 Uniformity

In addition to providing a long period, the Weyl sequence also provides a basis for uniformity in the output. The Weyl sequence is uniform. We demonstrate that by simply adding it to the square we will obtain a uniform output.

Theorem C: Let x be a random but not necessarily uniform bit stream. Let w be a uniform but not necessarily random bit stream. Then x + w will be a uniform bit stream.

Method: We will work with 32-bit quantities because the output of the RNG is 32 bits. We can demonstrate uniformity by showing that the probability of any given output is $1/2^{32}$.

Proof: Given an arbitrary element y in [0, 0xffffffff], let's compute the probability of y where y = x + w.

$$prob(y) = \sum_{x=0}^{2^{32}-1} (prob(x) * prob(w))$$

Now because w is uniform, prob(w) will be $1/2^{32}$. So,

$$prob(y) = \sum_{x=0}^{2^{32}-1} (prob(x) * 1/2^{32})$$

which is equvalent to

$$prob(y) = 1/2^{32} \sum_{x=0}^{2^{32}-1} (prob(x))$$

From basic probability, we know that the sum of the probabilities of all possible outcomes is 1. So,

$$prob(y) = 1/2^{32}$$

Q.E.D.

There are 2^{32} possible outputs and each has the probability of $1/2^{32}$. We conclude that the RNG will produce a discrete uniform distribution.

6   Initialization

The state vector consists of three 64-bit words: x, w, and s. Two of these are computed and one is a constant. The constant s is the element that provides for a distinct initialization. If x or w are used to provide a different initialization, there is some danger that overlapping data will be produced. For example, let's say that one initialized the state vector as follows:

$$x = 0; w = 0; s = 0xb5ad4eceda1ce2a9;$$

for the first run and then set the state

$$x = 2^{32}; w = 0; s = 0xb5ad4eceda1ce2a9;$$

for the next run. In this case, exactly the same data would be generated even though the initial x values are different. 0 squared is the same as 2^{32} squared when taken mod 2^{64}. If one only changes w, there is a possibility of overlap unless these are chosen with adequate space between them. So, it is recommended to use different s values when initializing the RNG. This will produce a different stream of data with no overlap.

The constant s should be an irregular bit pattern with roughly half of the bits set to one and half of the bits set to zero. The best statistics have been obtained with this type of pattern. Unduly sparse or dense values do not produce good results. If one initializes the RNG, for example, as follows: x=0; w=0; s = 0x0000000100000001; the first set of outputs appear non-random:

```
00000001
00000004
0000001b
00000406
00170a61
f765b52a
68d57352
0aafc03f
f461cd1e
fbe33cc0
808d47e0
230dc324
93202f86
```

The numbers eventually square out to random data, but it takes about five iterations before this happens. This type of effect can be avoided by choosing s constants that are neither too sparse nor too dense.

A method that has been shown to work well is to choose the constant s so that the hexadecimal digits or nibbles are different. This results in change in each 4-bit section of the Weyl sequence on each iteration of the RNG. The following constants are of this type: 0x9f32e1cbc5e1374b, 0x278c5a4d8419fe6b, 0x38ea2514b48de29f, 0x91c43526df517a8b. The digits are chosen so that the upper 8 are different and the lower 8 are different. Only non-zero digits are used and the least significant digit is chosen to be odd. Such constants have been tested extensively and produce good data. A utility called init_rand_digits which creates this type of constant is provided in the software download[1]. There are two methods for use. One may simply call init_rand_digits from the program to create the constant. Or one may use init_rand_digits offline to create an include file of constants that could be compiled into the program. The compiled in constants would make for a faster initialization. This routine has been verified to produce unique constants for input values ranging from 0 to 3 billion. Example usage is shown below:

```
#include ''init.h''
x = w = s = init_rand_digits(n);
```

or if using pre-computed constants

```
uint64_t seed[] = {
#include ''seed.h''
};
x = w = s = seed[n];
```

Note: x and w have been initialized non-zero so that a randomization occurs on the first iteration. This assures that as a set the first outputs from all streams should be random data.

The number of available constants from init_rand_digits is on the order of 33 quadrillion. Since the stream length is 2^{64}, this provides about 2^{118} random numbers in total. We have a suggestion regarding usage that may help make all the numbers more easily available. One can on initialization jump ahead to a later portion in the Weyl sequence to use that data instead of the data at the beginning of the Weyl sequence. For example, on the first run one might set the x, w, s as above. On the second run after setting x, w, and s, one could multiply w by say a trillion, w *= 1000000000000; and use that portion of the Weyl sequence. On the third run one could multiply w by 2 trillion, on the fourth run by 3 trillion and so on. Using this method one could use all 2^{64} numbers available from a given Weyl sequence.

---

[1]Software download available at http://mswsrng.wixsite.com/rand

7   Parallel Usage

This RNG is well suited to parallel processing. Creating a separate stream of data is simply a matter of choosing a unique integer. Assume we have a super-computer with N processing elements and n identifies a given processing element ($0 \leq n < N$). One could initialize the state for each instance of the RNG as shown in the example above.

8   Statistical and Timing Results

The msws generator was subjected to the BigCrush [6] and PractRand [5] tests. 25000 random seed values were tested with no significant failures. A square is nonlinear and for this reason the data is likely to be quite superior to data produced by any linear based RNG. This generator passes the tests of Linear Complexity and Matrix Rank.

The time to generate one billion random numbers was computed using an Intel Core i7-9700 3.0 GHz processor running Cygwin64. To provide a basis of comparison, we also timed Marsaglia's xorwow [7], one of the fastest RNGs in current usage. The time for msws was 1.17 seconds. The time for xorwow was 1.32 seconds.

9   Floating-Point Output

In random number generation, a floating-point number in the range of 0 to 1 is often required. In this section, we address the generation of floating-point numbers.

For the msws generator, one can simply divide by 2^{32} as shown below.

```
#define two32 4294967296.
double r;
r = msws() / two32;
```

This will produce a 32-bit resolution floating-point number in [0,1). A good compiler will convert a division by a constant to a multiplication[2]. If your compiler does not do this then you would multiply by 2.32830643653 86962890625e-10 instead of dividing by 2^{32}.

We will now consider generating a 53-bit resolution floating-point number. A IEEE 754 floating-point double has 53 bits of resolution. Such a number can be generated by dividing a 53-bit integer by 2^{53}. To obtain a 53-bit output, one could call msws twice and place the two 32-bit outputs into a 64-bit word and then shift right 11 bits as shown below.

---

[2]For the gcc compiler, this occurs with optimization -O1 or higher and can be verified by setting -save-temps and looking at the .s output file.

```
inline static uint64_t msws53() {
   uint64_t t;
   t = msws(); t = (t<<32) | msws(); return t>>11;
}
```

The above generator can produce one billion random numbers in about 2.29 seconds on an Intel Core i7-9700. A method which is about 40% faster is shown here.

```
uint64_t x1 = 0, w1 = 0, s1 = 0xb5ad4eceda1ce2a9;
uint64_t x2 = 0, w2 = 0, s2 = 0xb5ad4eceda1ce2ab;
inline static uint64_t msws53() {
   uint64_t xx;
   x1 *= x1; xx = x1 += (w1 += s1); x1 = (x1 >> 32) | (x1 << 32);
   x2 *= x2;      x2 += (w2 += s2); x2 = (x2 >> 32) | (x2 << 32);
   return (xx ^ x2)>>11;
}
```

This generator can produce one billion random numbers in about 1.29 seconds on an Intel Core i7-9700.

One may obtain a floating-point number from msws53 as follows.

```
#define two53 9007199254740992.
double r;
r = msws53() / two53;
```

This will produce a 53-bit resolution number in the range of [0,1). If your compiler does not convert a division by a constant to a multiplication, you would multiply by 1.1102230246251565404236316680908203125e-16 instead of dividing by 2^{53}.

10 Cryptography

Here we briefly discuss the subject of cryptography. The fundamental requirement for a cryptographically secure RNG is that it be difficult to determine the internal state by examining the outputs. This RNG is nonlinear and hence the only attack would probably be brute force. The state vector has three 64-bit elements: x, w, and s. Using sheer brute force, one could crack the state with $(2^{64})^3$ or 2^{192} combinations of x, w, and s. A more efficient attack which uses known available information might be as follows: The available information we have is a sequence of known outputs and also the operations performed in the RNG (square, add Weyl sequence, and shift). Now, let's consider three subsequent x values. This would allow

us to determine two subsequent w values and then an s value. This would provide a potential x, w, s combination which could be verified by checking if the generated outputs matched the known outputs. A sequence of known outputs would reduce the number of bits to examine in x from 64 to 32. So, if we look at three x values at time, it would require $(2^{32})^3$ or 2^{96} values. This is smaller than 2^{192} but is still quite a large number. It would still be impractical to crack the state in reasonable time even with this more efficient method. We conclude that the RNG is suitable for cryptography. One could significantly increase the complexity by simply xoring the output from two different msws generators. This would increase the complexity to $(2^{64})^3 * (2^{32})^3$ or 2^{288}.

11   Running the Middle Square with an RNG

In this paper we have proposed using a Weyl sequence to run the middle square. The Weyl sequence is very fast and produces satisfactory results. It may be of interest, however, to note that any reasonably good RNG with uniform output could be used instead of the Weyl sequence. Assume that rand64() is a 64-bit RNG. The following shows how to create a middle square version of this RNG:

```
uint64_t x = 0;
inline static uint32_t msrand() {
    x *= x; x += rand64(); return x = (x>>32) | (x<<32);
}
```

This will produce a 32-bit output that would most likely pass the tests of Linear Complexity and Matrix Rank.

12   Conclusion

The middle square was invented at the very beginning of computer science. Modern 64-bit computing architecture has made it possible to create a usable version which has a sufficiently long period (2^{64} per stream). Processing speed is comparable with the fastest RNGs. An easy to use stream capability makes this generator quite suitable for parallel processing. A square is nonlinear and gives this generator an advantage in quality of data over linear based generators.

A free software package with example programs is available at
`http://mswsrng.wixsite.com/rand`

References

[1]   R. P. Brent. "Some Long-Period Random Number Generators using Shifts and Xors". In: *ANZIAM Journal* 48 (2006), pp. C188–C202. URL: `http://maths-people.anu.edu.au/~brent/pub/pub224.html` (cit. on pp. 1, 11).

[2]   Wikipedia Contributors. "Integer Overflow". In: *Wikipedia, The Free Encyclopedia* (2018). URL: `https://en.wikipedia.org/w/index.php?title=Integer_overflow&oldid=874320318` (cit. on p. 11).

[3]   Wikipedia Contributors. "Mersenne Twister". In: *Wikipedia, The Free Encyclopedia* (2018). URL: `https://en.wikipedia.org/w/index.php?title=Mersenne_Twister&oldid=864897141` (cit. on p. 11).

[4]   Wikipedia Contributors. "Weyl Sequence". In: *Wikipedia, The Free Encyclopedia* (2018). URL: `https://en.wikipedia.org/w/index.php?title=Weyl_sequence&oldid=878733044` (cit. on p. 12).

[5]   C. Doty-Humphrey. *Practically Random: C++ library of statistical tests for RNGs*. version 0.94. 2018. URL: `http://pracrand.sourceforge.net` (cit. on pp. 8, 12).

[6]   P. L'Ecuyer and R. Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software* 33 (2007). URL: `http://simul.iro.umontreal.ca/testu01/tu01.html` (cit. on pp. 8, 11).

[7]   George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (2003). URL: `www.jstatsoft.org/v08/i14/paper` (cit. on pp. 8, 11).

[8]   M. Matsumoto and T. Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), pp. 3–30. URL: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html` (cit. on p. 11).

[9]   N. Metropolis and S. Ulam. "The Monte Carlo Method". In: *Journal of the American Statistical Association* 44.247 (1949), pp. 335–341 (cit. on p. 12).

[10]   John von Neumann. "Various Techniques Used in Connection with Random Digits". In: *A.S. Householder, G.E. Forsythe, and H.H. Germond, eds., Monte Carlo Method, National Bureau of Standards Applied Mathematics Series* 12 (1951), pp. 36–38 (cit. on pp. 1, 12).

[11] M. Saito and M. Matsumoto. *XSadd*. 2014. URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/XSADD/ (cit. on p. 11).

[12] J.K. Salmon et al. "Parallel Random Numbers: As Easy As 1, 2, 3". In: *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, New York*. 2011, 16:1–16:12 (cit. on p. 12).

[13] S. Vigna and D. Blackman. *xoroshiro128+*. 2018. URL: http://xoshiro.di.unimi.it/xoroshiro128plus.c (cit. on p. 11).

[14] H. Weyl. "Ueber die Gleichverteilung von Zahlen mod. Eins". In: *Math. Ann.* 77.3 (1916), pp. 313–352 (cit. on p. 12).

Author address

1. **Bernard Widynski**, mailto:mswsrng@gmail.com