Remarks on "The Logistic Lattice in Random Number Generation"

Neal R. Wagner

1. Introduction

Please refer to the quoted article before reading these "remarks." I have always been fond of this particular random number generator (RNG) because of its unusual character, as explained in the main article. These remarks just present a few additional experiments that I've carried out.

2. The Logistic Equation

The logistic equation is given by $f(x) = 4x(1-x), 0 \leq x \leq 1$. I experimented on cycle lengths with various hardware, but not with the IEEE standard for `double` calculations.

Table 1 shows cycle lengths obtained from 100000 random starting values. Notice that 60% of the time it headed into the cycle (0) of length 1. (The software checked that this was not the other cycle of length 1: $(3/4)$.) This cycle structure is similar in form to that given in the earlier article for other hardware

Hard-ware	Preci-sion	Cycle Structure			Number of starting values
		Cycle length	Percent occurrence	Average initial run	
IEEE standard (Pent II)	**double**	**1**	**59.956%**	**49 878 403**	**100000**
		15 784 521	**40.007%**	**24 970 938**	
		1 122 211	**0.033%**	**555 186**	
		173 139	**0.003%**	**196 763**	
		28 970	**0.001%**	**484 337**	

Table 1. Cycles of the logistic equation.

I also carried out experiments on the interval of numbers around 0.5 that is mapped into 1.0 and then into the cycle (0) by f. The following is a small piece of C code that checks for this interval. Below `getdouble()` fetches the next `double` value, and `putdouble(x)` prints the value of `x` in exponent and hex (internal) form.

```
x = getdouble();
y = 4.*x*(1.0 - x);
putdouble(y);
z = 4.*y*(1.0 - y);
putdouble(z);
```

Here are results of a run. Notice that `3ff00000 00000000` is an exact one, and all zero bits is an exact zero. The number `3feffffff fffffffff` is as close to 1 as one can get from below. User input is in **bold red**, while the blue comments inside square brackets are not part of the output.

```
0.500000003727
 y: Dec:  9.9999999999999989e-01, Hex: 3feffffff fffffff [almost one]
 z: Dec:  4.4408920985006257e-16, Hex: 3cbffffff fffffff
0.500000003726
 y: Dec:  1.0000000000000000e+00, Hex: 3ff00000 00000000 [exactly one]
 z: Dec:  0.0000000000000000e+00, Hex: 00000000 00000000

0.499999996273
 y: Dec:  9.9999999999999989e-01, Hex: 3feffffff fffffff [almost one]
 z: Dec:  4.4408920985006257e-16, Hex: 3cbffffff fffffff
0.499999996274
 y: Dec:  1.0000000000000000e+00, Hex: 3ff00000 00000000 [exactly one]
 z: Dec:  0.0000000000000000e+00, Hex: 00000000 00000000
```

The above shows that the entire interval $(0.499999996274, 0.500000003726)$ of length 0.000000007452 is mapped to 1.0, and then on into the cycle (0). The hex values show that no larger interval will map this way. This might not sound like so much, but two smaller intervals map into the above interval, and so on with double the number of intervals at each stage, each smaller than those at the previous stage. To get the two intervals that map into the interval around 0.5, one can solve for x in the formula $y = 4x(1-x)$, to get $x = (1 \pm \sqrt{1-y})/2$. Now plug in the endpoints of the first interval above to get the endpoints of the two intervals that map into it. The following results, using the same C code as before, gives values for the two intervals that map into the interval around 0.5. Again the hex values show that no larger interval will map this way. (Below, "almost one" means as close to 1 as you can get with a `double` value.)

```
0.853553389275
 y: Dec:  5.0000000372864117e-01, Hex: 3fe00000 020075e6
 z: Dec:  9.9999999999999989e-01, Hex: 3feffffff fffffff [almost one]
0.853553389276
 y: Dec:  5.0000000372581288e-01, Hex: 3fe00000 02001263
 z: Dec:  1.0000000000000000e+00, Hex: 3ff00000 00000000 [exactly one]
0.853553391911
 y: Dec:  4.9999999627290737e-01, Hex: 3fdffffff fbff812c
 z: Dec:  9.9999999999999989e-01, Hex: 3feffffff fffffff [almost one]
0.853553391910
 y: Dec:  4.9999999627573577e-01, Hex: 3fdffffff fc004834
 z: Dec:  1.0000000000000000e+00, Hex: 3ff00000 00000000 [exactly one]

0.146446610725
 y: Dec:  5.0000000372864128e-01, Hex: 3fe00000 020075e7
 z: Dec:  9.9999999999999989e-01, Hex: 3feffffff fffffff [almost one]
0.146446610723
 y: Dec:  5.0000000372298437e-01, Hex: 3fe00000 01ffaede
 z: Dec:  1.0000000000000000e+00, Hex: 3ff00000 00000000 [exactly one]
0.146446608089
 y: Dec:  4.9999999627290737e-01, Hex: 3fdffffff fbff812c
 z: Dec:  9.9999999999999989e-01, Hex: 3feffffff fffffff [almost one]
0.146446608090
 y: Dec:  4.9999999627573583e-01, Hex: 3fdffffff fc004835
 z: Dec:  1.0000000000000000e+00, Hex: 3ff00000 00000000 [exactly one]
```

Table 2 below shows the first interval and the two intervals mapping into it.

Interval I	Length	What maps where
(0.499 999 996 274, 0.500 000 003 726)	0.000 000 007 452	$f(x) = 1$, for x in I
(0.853 553 389 275, 0.853 553 391 911)	0.000 000 002 636	$f(f(x)) = 1$, for x in I
(0.146 446 608 089, 0.146 446 610 724)	0.000 000 002 635	$f(f(x)) = 1$, for x in I

Table 2. Intervals mapped to 1.

3. The Re-mapped Logistic Equation

While I was studying the logistic equation, I had an intuitive feeling that the equation could be re-structured in a way that would improve its performance. I didn't know exactly what I was looking for. I spent an entire afternoon one day scratching around, trying all sorts of ideas, drawing endless pictures. The result was the equation in this section. It very much out-performs the ordinary logistic equation of the previous section in terms of the length of its cycles and the average initial run up to a cycle. The re-structured or re-mapped logistic equation is defined by:

$$F_\beta = \begin{cases} 2|x|(2 - |x|), \text{ for } |x| \le \beta, \\ -2(1 - |x|)^2, \text{ for } \beta < |x| \le 1, \end{cases} \tag{1}$$

where $\beta = 1 - (1/\sqrt{2})$. The earlier article gives a graph of the original (expanded by a factor of two) and re-mapped versions (Figure 1 of the earlier article).

Table 3 below gives the cycles obtained from 50000 random starting values of the re-mapped equation using the IEEE standard. (In the earlier article, I only used 475 random starting values.)

Hard-ware	Preci-sion	Cycle Structure			Number of starting values
		Cycle length	Percent occurrence	Average initial run	
IEEE standard (Pent II)	**double**	**73 573 097**	**80.018%**	**122 671 507**	**50000**
		38 326 216	**18.172%**	**135 923 736**	
		6 006 146	**1.44%**	**24 288 536**	
		5 195 797	**0.368%**	**10 974 795**	
		322 931	**0.002%**	**358 102**	

Table 3. Cycles of the re-mapped logistic equation.

In the earlier article, I stated: "In infinite precision, this re-mapped equation behaves exactly like the original, but with floating point numbers there is no longer any convergence to the cycle (0) of length 1." This was my intuition all along, but I still don't have a proof of it. However, numerical experiments similar to those above show that the conjecture must be true.

So here again I carried out experiments in the re-mapped setting. The numbers must be near but less than 1 and near but greater than -1. So what corresponds to an interval $(0.5-\epsilon, 0.5+\epsilon)$ is replaced in this re-mapped setting by the intervals $[-1, -1+\epsilon)$ and $(1-\epsilon, 1]$. I used the following C code to evaluate the behavior under f using different values of ϵ.

```
#define BETA 0.292893218813452476
x = getdouble();
putdouble(x);
x = fabs(x);
```

```
if ( x <= BETA ) y = (2.0*x*(2.0-x));
else  y = (-2.0*(1.0-x)*(1.0-x));
putdouble(y);
y = fabs(y);
if ( y <= BETA ) z = (2.0*y*(2.0-y));
else  z = (-2.0*(1.0-y)*(1.0-y));
putdouble(z);
```

Here are results of the second run. Notice that `3feffffff ffffffff` is the largest number less than 1, while `bfeffffff ffffffff` is the smallest number greater than -1. The number `80000000 00000000` is usually referred to as "minus zero", that is, -0, but in fact the hardware treats it just like the usual value of zero. (Again, "almost one" means as close to 1 as you can get with a `double` value.)

```
0.99999999999999994
x: Dec:  9.9999999999999989e-01, Hex: 3feffffff ffffffff [almost one]
y: Dec: -2.4651903288156619e-32, Hex: b9600000 00000000
z: Dec:  9.8607613152626476e-32, Hex: 39800000 00000000
0.99999999999999995
x: Dec:  1.0000000000000000e+00, Hex: 3ff00000 00000000 [exactly one]
y: Dec: -0.0000000000000000e+00, Hex: 80000000 00000000 [exactly -zero]
z: Dec:  0.0000000000000000e+00, Hex: 00000000 00000000 [exactly zero]

-0.99999999999999994
x: Dec: -9.9999999999999989e-01, Hex: bfeffffff ffffffff [almost -one]
y: Dec: -2.4651903288156619e-32, Hex: b9600000 00000000
z: Dec:  9.8607613152626476e-32, Hex: 39800000 00000000
-0.99999999999999995
x: Dec: -1.0000000000000000e+00, Hex: bff00000 00000000 [exactly one]
y: Dec: -0.0000000000000000e+00, Hex: 80000000 00000000 [exactly -zero]
z: Dec:  0.0000000000000000e+00, Hex: 00000000 00000000 [exactly zero]
```

The above results show that no proper intervals with left end at -1 or with right end at 1 will map into 0, but only the single values -1 and 1.

4. Runs Into the Cycles (0.75) and (-0.5)

The ordinary logistic equation has the obvious cycle (0) of length 1 and one other cycle of length 1, namely (0.75). In the re-mapped logistic equation, the corresponding cycles are (0) and (-0.5).

With the ordinary logistic equation in single precision (`float` type), the original article showed that a large number of initial values (93%) end up in (0), while there are no runs at all into (0.75). Using the re-mapped logistic equation, there were very few runs into (0), and again none at all into (-0.5).

In the article, I stated without proof: "The cycle search in single precision is exhaustive because every cycle will eventually enter the range from $3/4$ to 1." I meant to say "inclusive" at the end, that is, $[0.75, 1]$. I also meant to include, "except for the cycle (0)." Here is a proof, with a tighter statement of the assertion. (Below, (x) means a cycle containing just x, and $(0.75, 1]$ means those x for which $0.75 < x \leq 1$.)

Here are cases based on the value of $x \in [0, 1]$:

- If $x = 0$, then it is in the cycle (0), the special case.
- If $x = 0.75$ then it is in the cycle (0.75), so it is certainly in the range $[0.75, 1]$.
- If $x \in (0.75, 1]$ then $f(x) < 0.75$, so no cycle can be entirely inside $(0.75, 1]$.

- If $x \in (0.25, 0.75)$ then $f(x) \in (0.75, 1]$ so no cycle can be entirely inside $(0.25, 0.75)$, and in fact in this case, $f(x) \in (0.75, 1]$.
- If $x \in (0, 0.25]$ then $f'(x) = 4 - 8x$, so $f'(x) \geq 2$. Using the Mean Value Theorem, there is an $x' < x$ such that $(f(x) - 0)/(x - 0) = f'(x')$, so $f(x)/x \geq 2$ or $f(x) \geq 2x$.

  If we take iterates $f(x), f(f(x)) = f^{(2)}(x)$, etc., then eventually either $f^{(n)}(x) = 0.25$ so that $f^{(n+1)}(x) = 0.75$, or $f^{(n)}(x) > 0.25$ so that $f^{(n+1)}(x) > 0.75$, so that iterates go into $(0.75, 1]$. Thus no cycle can be entirely inside $(0, 0.25]$.

This completes the proof. This shows that every cycle except (0) must eventually be inside $[0.75, 1]$ (including the cycle (0.75).) It also shows that no cycle can be entirely inside $(0, 0.25]$, or entirely inside $(0.25, 0.75)$, or entirely inside $(0.75, 1]$. However, much to my surprise, there is a run of length 512, ending in the cycle (0.75), that is entirely inside $(0, 0.75]$. The same situation occurs with the re-mapped logistic equation.

So now I look into possible runs into (0.75) (ordinary logistic equation) and (-0.5) (re-mapped logistic equation) in double precision. First, focus on the ordinary logistic equation, and work backwards from 0.75. In infinite precision, there are in general two values that map by f into any single value. We can use the equation $x = (1 \pm \sqrt{1 - y})/2$ to get approximate values, although to qualify, the value must be exact. Calculating, both 0.25 and 0.75 map into 0.75. There are two values that map approximately into 0.25: 0.93301270189221930 and 0.066987298107780702. However, as the calculation below shows, only the second of these values actually works. (Below, the input consists of exact hex values, rather than approximate decimal values.)

```
3feddb3d 742c2655
   x : Dec:  9.3301270189221930e-01, Hex: 3feddb3d 742c2655 --+
f1(x): Dec:  2.5000000000000011e-01, Hex: 3fd00000 00000002   | succes
f2(x): Dec:  7.5000000000000022e-01, Hex: 3fe80000 00000002   | -sive
                                                              | values
3feddb3d 742c2656                                             |
   x : Dec:  9.3301270189221941e-01, Hex: 3feddb3d 742c2656 --+
f1(x): Dec:  2.4999999999999969e-01, Hex: 3fcfffff fffffff5
f2(x): Dec:  7.4999999999999933e-01, Hex: 3fe7ffff fffffffa

3fb12614 5e9ecd56
   x : Dec:  6.6987298107780674e-02, Hex: 3fb12614 5e9ecd56
f1(x): Dec:  2.5000000000000000e-01, Hex: 3fd00000 00000000
f2(x): Dec:  7.5000000000000000e-01, Hex: 3fe80000 00000000
```

For the number that starts with 0.9330127, you can see two *successive* hex values, one too small and one too large, so that no value can work exactly. The number that starts with 0.066987298 has a close hex value that works exactly. So only one "precursor" value from 0.25 works. Continuing to move backwards, at each stage the value near 0 works, while the value near 1 does not. One gets a *single run* into 0.75 of length 512:

```
00118bc4 418cafe1
     x : Dec:  2.4400665274079501e-308,  Hex: 00118bc4 418cafe1
f  1(x): Dec:  9.7602661096318003e-308,  Hex: 00318bc4 418cafe1
f  2(x): Dec:  3.9041064438527201e-307,  Hex: 00518bc4 418cafe1
f  3(x): Dec:  1.5616425775410881e-306,  Hex: 00718bc4 418cafe1
 ... [480 lines omitted, all ending with ``18bc4 418cafe1'']
f484(x): Dec:  6.0874789166839900e-17,   Hex: 3c918bc4 418cafe1
f485(x): Dec:  2.4349915666735960e-16,   Hex: 3cb18bc4 418cafe1
```

```
f486(x): Dec:  9.7399662666943820e-16,   Hex: 3cd18bc4 418cafe0
f487(x): Dec:  3.8959865066777488e-15,   Hex: 3cf18bc4 418cafdb
f488(x): Dec:  1.5583946026710935e-14,   Hex: 3d118bc4 418cafc8
f489(x): Dec:  6.2335784106842770e-14,   Hex: 3d318bc4 418caf7b
f490(x): Dec:  2.4934313642735553e-13,   Hex: 3d518bc4 418cae47
f491(x): Dec:  9.9737254570917352e-13,   Hex: 3d718bc4 418ca978
f492(x): Dec:  3.9894901828327149e-12,   Hex: 3d918bc4 418c963a
f493(x): Dec:  1.5957960731267196e-11,   Hex: 3db18bc4 418c4943
f494(x): Dec:  6.3831842924050162e-11,   Hex: 3dd18bc4 418b1567
f495(x): Dec:  2.5532737167990266e-10,   Hex: 3df18bc4 418645f6
f496(x): Dec:  1.0213094864588423e-09,   Hex: 3e118bc4 41730830
f497(x): Dec:  4.0852379416630770e-09,   Hex: 3e318bc4 41261118
f498(x): Dec:  1.6340951699895633e-08,   Hex: 3e518bc4 3ff234b9
f499(x): Dec:  6.5363805731475719e-08,   Hex: 3e718bc4 3b22c33c
f500(x): Dec:  2.6145520583619446e-07,   Hex: 3e918bc4 27e4fd53
f501(x): Dec:  1.0458205499094791e-06,   Hex: 3eb18bc3 daede659
f502(x): Dec:  4.1832778246754260e-06,   Hex: 3ed18bc2 a7119500
f503(x): Dec:  1.6733041299448271e-05,   Hex: 3ef18bbd d7a0f869
f504(x): Dec:  6.6931045219108568e-05,   Hex: 3f118baa 99e912e4
f505(x): Dec:  2.6770626181717779e-04,   Hex: 3f318b5d a3b24972
f506(x): Dec:  1.0705383806982466e-03,   Hex: 3f518a29 d563b9c0
f507(x): Dec:  4.2775693130947942e-03,   Hex: 3f71855b 44e5d92f
f508(x): Dec:  1.7037086855465854e-02,   Hex: 3f91722b 8b740eb3
f509(x): Dec:  6.6987298107780674e-02,   Hex: 3fb12614 5e9ecd56
f510(x): Dec:  2.5000000000000000e-01,   Hex: 3fd00000 00000000
f511(x): Dec:  7.5000000000000000e-01,   Hex: 3fe80000 00000000
```

The notation `f 3(x)` stands for `f(f(f(x)))`, and so on for values other than 3. The largest possible IEEE `double` less than 1 is `3feffffff ffffffff`. Apply f to this value to get $4.4408920985006257e$–16. This is the smallest value, other than 0, that can occur in a run using the logistic equation, although one could choose a smaller initial value. Thus everything in the list above preceding `f486(x)` could only occur as an initial value. All numbers before `f485(x)` end with `18bc4 418cafe1`, and in this range each successive hex value is obtained by adding `200000 00000000` to the previous value, that is, by multiplying by 4, since for numbers x very close to 0, $f(x)$ is exactly multiplication by 4. The smallest possible IEEE `double` is $2.2250738585072014e$–308, so that the run cannot be made one longer. Of course all subsequent values in the run will be the cycle value 0.75.

With the re-mapped there is an exact corresponding run of length 512 into the cycle (-0.5):

```
00218bc4 418cafe1
    x :  Dec:  4.8801330548159002e-308,  Hex: 00218bc4 418cafe1
f 1(x):  Dec:  1.9520532219263601e-307,  Hex: 00418bc4 418cafe1
f 2(x):  Dec:  7.8082128877054403e-307,  Hex: 00618bc4 418cafe1
f 3(x):  Dec:  3.1232851550821761e-306,  Hex: 00818bc4 418cafe1
 ... [504 lines omitted]
f508(x): Dec:  3.4074173710931709e-02,   Hex: 3fa1722b 8b740eb3
f509(x): Dec:  1.3397459621556135e-01,   Hex: 3fc12614 5e9ecd56
f510(x): Dec:  5.0000000000000000e-01,   Hex: 3fe00000 00000000
f511(x): Dec: -5.0000000000000000e-01,   Hex: bfe00000 00000000
```

There is one type of addition to this run. In the case of the re-mapped logistic equation, at any stage of the run, you can start with the negative of an entry. There is no precursor to this value, but the subsequent values are the same positive ones as above. Thus, for example, the numbers $-.034074173710931709$, 0.13397459621556135, 0.5, -0.5 form an initial run up to -0.5. Otherwise the same remarks apply to these calculations as to the ones for the ordinary logistic equation.

The same behavior occurs with IEEE `float` calculations, except that the run into (0.75) or (-0.5) is only 64 long. Here the smallest possible IEEE `float` is $1.17549435e$–38. The list of values below shows both the ordinary and re-mapped logistic equations side-by-side. Here before the `f52(x)` terms, each term is obtained from the previous one by adding `1000000`, that is, multiplying by 4. Any run for the re-mapped logistic equation can start with a negated entry, as above.

In the original article, the exhaustive search of the ordinary equation for `float` precision only covered the range in the interval $[0.75, 1]$, so it found the cycle (0.75), but missed the run below altogether, and similarly for the re-mapped equation.

```
         Ordinary Logistic Equation   |  Re-mapped Logistic Equation
         008c5e22                     |  010c5e22
    x : 1.2890738e-38, Hex: 008c5e22 |  2.5781476e-38, Hex: 010c5e22
f 1(x): 5.1562952e-38, Hex: 018c5e22 |  1.0312590e-37, Hex: 020c5e22
f 2(x): 2.0625181e-37, Hex: 028c5e22 |  4.1250361e-37, Hex: 030c5e22
f 3(x): 8.2500723e-37, Hex: 038c5e22 |  1.6500145e-36, Hex: 040c5e22
    ...  [46 lines omitted, all ending with ``c5e22'']
f50(x): 1.6340952e-08, Hex: 328c5e22 |  3.2681903e-08, Hex: 330c5e22
f51(x): 6.5363807e-08, Hex: 338c5e22 |  1.3072761e-07, Hex: 340c5e22
f52(x): 2.6145520e-07, Hex: 348c5e21 |  5.2291040e-07, Hex: 350c5e21
f53(x): 1.0458206e-06, Hex: 358c5e1f |  2.0916411e-06, Hex: 360c5e1f
f54(x): 4.1832777e-06, Hex: 368c5e15 |  8.3665554e-06, Hex: 370c5e15
f55(x): 1.6733042e-05, Hex: 378c5def |  3.3466084e-05, Hex: 380c5def
f56(x): 6.6931047e-05, Hex: 388c5d55 |  1.3386209e-04, Hex: 390c5d55
f57(x): 2.6770626e-04, Hex: 398c5aed |  5.3541252e-04, Hex: 3a0c5aed
f58(x): 1.0705384e-03, Hex: 3a8c514f |  2.1410768e-03, Hex: 3b0c514f
f59(x): 4.2775692e-03, Hex: 3b8c2ada |  8.5551385e-03, Hex: 3c0c2ada
f60(x): 1.7037086e-02, Hex: 3c8b915c |  3.4074172e-02, Hex: 3d0b915c
f61(x): 6.6987298e-02, Hex: 3d8930a3 |  1.3397460e-01, Hex: 3e0930a3
f62(x): 2.5000000e-01, Hex: 3e800000 |  5.0000000e-01, Hex: 3f000000
f63(x): 7.5000000e-01, Hex: 3f400000 | -5.0000000e-01, Hex: bf000000
```

5. Discussion

Suppose one tried to use the ordinary logistic equation directly as a RNG. Start with a seed (some initial double value). Then 99.93% of the time, you'll get some 40-50 million values before cycling (40%) or ending in 0 (60%). This is compared to a single cycle of about 2 billion numbers with an old-fashioned 32-bit generator. (Of course you have to iterate maybe 100 times between sampling for a random number.) Anyway, this isn't so terrible, perhaps, except for the remaining 0.07% of the time.

The main article provides three ways to improve the basic logistic equation based RNG:

1. Iterate the generator to fill with random noise.
2. Use the "re-mapped" logistic equation that has much longer cycles and initial runs, and has no convergence to 0.
3. Use a coupled lattice of logistic equation generators that will perturb one another.

Of the above, item 1 is essential, but fairly easy to think of. Item 2 is the one I'm most proud of, because it's unexpected and original. But item 2 is not essential as long as one uses item 3, which really is essential. The basic idea of item 3 was suggested to me by Kay Robbins. (Coupled lattices of logistic equations are a standard simple model of chaos, but I didn't know that initially.)