Chaos, Solitons & Fractals 45 (2012) 238–245

Contents lists available at SciVerse ScienceDirect

Chaos, Solitons & Fractals

Nonlinear Science, and Nonequilibrium and Complex Phenomena

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

Analyzing logistic map pseudorandom number generators for periodicity induced by finite precision floating-point representation

K.J. Persohn, R.J. Povinelli *

Electrical and Computer Engineering Department, Marquette University, Milwaukee, WI, USA

ARTICLE INFO

Article history:
Received 31 December 2010
Accepted 6 December 2011
Available online 20 January 2012

ABSTRACT

Because of the mixing and aperiodic properties of chaotic maps, such maps have been used as the basis for pseudorandom number generators (PRNGs). However, when implemented on a finite precision computer, chaotic maps have finite and periodic orbits. This manuscript explores the consequences finite precision has on the periodicity of a PRNG based on the logistic map. A comparison is made with conventional methods of generating pseudorandom numbers. The approach used to determine the number, delay, and period of the orbits of the logistic map at varying degrees of precision (3 to 23 bits) is described in detail, including the use of the Condor high-throughput computing environment to parallelize independent tasks of analyzing a large initial seed space. Results demonstrate that in terms of pathological seeds and effective bit length, a PRNG based on the logistic map performs exponentially worse than conventional PRNGs.

1. Introduction

Chaotic nonlinear dynamical systems are capable of imitating random noise. This property has sparked research interest leading to various proposals of applied chaos in communications, cryptography, and computer simulations. Misunderstanding the relationship between chaos and an applied field has resulted in numerous failures when practical applications attempt to implement chaos theory. Some examples include an insecure synchronized communication system [1], a weak and slow encryption algorithm [2], and a pseudorandom number generator (PRNG) that falls short of its claims [3].

In this manuscript, we explore the following logistic map as a PRNG.

$$x_{n+1} = 4x_n(1 - x_n), \quad x_n \in (0, 1). \tag{1}$$

Specifically, we examine empirically and exhaustively the cyclic behavior of (1) in the range of 3 to 23 bits of precision. While we analyze (1) in the context of cryptography to develop an understanding of why (1) performs poorly compared to conventional PRNGs, our results are also applicable to understanding the limitations of finite precision simulations of (1).

PRNGs are an important area of study because of their ubiquitous use in a variety of applications: decision-making, sampling, cryptography, and computer simulations. One can also use a PRNG to construct other cryptographic primitives such as block-ciphers and hashing functions [4–7]. Various generation methods have different tradeoffs in randomness and computational efficiency that lead to compromises in speed, security, and randomness. Quantifying these characteristics is essential for comparisons between chaos-based algorithms and their conventional counterparts. In the next section, we discuss ideal properties of PRNGs that influence randomness and metrics for benchmarking performance in this context.

In the remainder of this manuscript, we present necessary background information, which explains why the claims in [3] are impossible in practice. Next, we propose a method to quantify the periodicity of a chaos-based PRNG implementation. Finally, an example based on the logistic map demonstrates the differences between chaos-based and conventional PRNGs.

* Corresponding author.
E-mail addresses: kpersohn@ieee.org (K.J. Persohn), richard.povinelli@marquette.edu (R.J. Povinelli).

0960-0779/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.chaos.2011.12.006

2. Background

2.1. Pseudorandom number generators

Sequences of random numbers are useful in many computer applications. Generation of these sequences is said to be done pseudo-randomly. That is, although the output appears to be random, the output is actually generated deterministically based on a seed value. Seeds are useful when reproducible sequences are desired, for example, debugging a simulation or encrypting/decrypting a message. An ideal random number generator is infinite, aperiodic, uniform, uncorrelated, and computationally efficient [3]. In other words, an ideal generator produces an endless sequence of numbers without repeating itself, and each number in the sequence has an equal probability of being generated. Moreover, successive terms are not predictable without knowing the seed [8]. Repetitive or correlated generators lead to systematic errors in simulations and insecure cryptosystems.

In practice, a PRNG implementation cannot be infinite or aperiodic when implemented with a finite precision computer system. The bit depth allocated to each numerical representation inherently limits the quantity of unique numbers. Consequently, this finite set limits the seed space for any PRNG implementation as well. As a corollary, a PRNG implementation is periodic because the sequences naturally repeat when the finite space used to represent each term is exhausted. In the context of finite precision implementations, an ideal PRNG does not repeat itself until all elements of its seed space have been generated. Like the theoretically ideal PRNG, the ideal practical implementation is uncorrelated, uniformly distributed, and computationally efficient for the first iteration of the entire seed space.

Various statistical and empirical tests exist to measure the randomness of a sequence generated by a PRNG. Some popular metrics are the chi-square (χ^2), Kolmogorov–Smirnov, poker, and run-up tests [9]. Passing these tests is a good indication a PRNG produces uncorrelated terms. Other authors have obtained results from standard metrics that suggest chaos-based generators are capable of sufficient randomness [10,11]. However, it is always best to test a PRNG in a specific application before determining it is sufficiently random [12]. For this reason, diverse generation methods are desirable for different applications.

However, statistical analysis does not provide a complete characterization of a PRNG. Previous studies of chaotic PRNGs limited their statistical tests to single, relatively short sequences [3,10]. While these sequences pass various statistical tests, our results illustrate that these sequence lengths are inconsistent. As a baseline for comparison, consider the characteristics of conventional, integer-based generation methods. Specifically, properly configured conventional generators can guarantee 100% utilization of the bits allotted for representing an entire period without repetition. This comparison bridges the gap between chaotic and conventional PRNGs. As will be shown later in this manuscript, truncation effects can be detrimental to the performance of a chaos-based implementation using finite precision floating-point numbers. Analyzing floating-point chaotic generators in a class of their own does not put the limitations of finite precision in perspective.

As a review, consider the following conventional methods:

2.1.1. Linear congruential generator

The linear congruential generator (LCG) is a very common class of PRNG used by many C compilers. It is defined by the recurrence relation

$$x_{k+1} = (ax_k + c) \bmod m. \tag{2}$$

The maximum period is limited to m uncorrelated numbers [13]. Some implementations, for example the Java 2 SE Random class, elect to throw out lower order bits for enhanced randomness [14]. Consequently, they do not maximize periodicity with respect to the number of bits representing the seed. Nevertheless, the LCG utilizes 100% of the retained bits producing a full period for all seed values when a, c, and m meet certain conditions [9]. In addition, LCGs are relatively fast and easy to implement. Unfortunately, this class of generator is subject to a number of defects making it unsuitable for simulations or cryptography [8].

2.1.2. Mersenne Twister (MT)

The MT is the default choice for randomization in many popular software tools including MATLAB, Python, and Ruby. The MT recurrence relation takes the form

$$x_{k+n} = x_{k+m} \oplus (x_k^u | x_{k+1}^l)\mathbf{A}, \quad k = 0, 1, 2, \ldots \tag{3}$$

where $|$ denotes bitwise OR, $\oplus$ is bitwise XOR, and x^u, x^l represent bitmasks applied to x. The matrix $\mathbf{A}$ is the twist transformation as described in [15]. The MT period is based on a Mersenne prime, commonly $2^{19937} - 1$. This extremely long period is attractive for simulations; however, the MT becomes predictable after a relatively small number of iterations. For example, the MT19937 is predictable after only 624 iterations—far short of its entire period. Consequently, the MT is unsuitable for cryptographic applications without further modifications such as those in [16].

2.2. Chaos and cryptography

The motivation to study chaos-based PRNGs comes from many parallels between chaos and cryptography. Chaotic systems are highly sensitive to changes in initial conditions. As a result, the mixing property of chaotic systems achieves desirable cryptographic properties of diffusion and confusion. This ensures that influence of key and plaintext bits are spread over the ciphertext, where the key is a secret, the plaintext is the message, and the ciphertext is an encrypted combination of the key and the plaintext. Moreover, successive iterations of a chaotic system reduce the statistical dependency of the ciphertext on the plaintext. These iterations closely parallel rounds of a cryptosystem [17]. All of these relationships allude to applied chaos being useful for cryptography. Likewise, chaos is also applicable to other situations that require randomness, such as computer simulations.

There is one significant difference between chaotic systems and cryptosystems that makes successful implementation challenging. Cryptosystems are defined on a discrete set of numbers (often a range of integers) that can be implemented on a computer with finite precision. On the other hand, chaotic systems rely on the set of real numbers to produce many of the desirable properties that are applicable to PRNGs. Truncation of finite precision real numbers causes sequences to repeat with very small periods relative to the corresponding cycles in purely theoretical infinite precision representations. Overlooking this crucial detail leads to implementations that fall short of expectations [3]. In terms of cryptanalysis, short periods lead to predictability after a relatively small number of iterations. Obviously, this trait is undesirable in a secure system and is further explained in [18].

2.3. The logistic map

Our example chaotic PRNG is based on the logistic map. The recurrence relation

$$x_{n+1} = kx_n(1 - x_n), \quad x_n \in (0,1) \tag{4}$$

defines the logistic map, where x_n is the nth value of the map and k is a parameter. Since chaotic behavior is of interest, $k = 4$ for this example. For a detailed discussion of the logistic map and its many applications, see [19]. Note that (4) is defined on the open set (0,1) because 0 is a known fixed point and 1 maps to 0. The logistic map is an interesting chaotic system to study because it uses simple operators that should be computationally fast in implementation. Furthermore, it is defined on a range of real numbers so it exemplifies the problem of interest when implemented in finite precision.

2.4. Floating-point representation

Computers use a special representation to store floating-point numbers as binary digits in memory. This demonstration uses the format that virtually all modern computers conform to, IEEE 754-2008 [12]. Specifically, this study uses single precision (binary32) numbers to keep computations manageable. In binary32, 4 bytes represent each floating-point number. The left-most bit designates the sign followed by 8 exponent bits and finally 23 fraction bits (significand). An additional implied leading 1 on the fractional part gives 24 total bits of significand precision. A bias representation allows signed exponents. For example, Fig. 1 depicts the bitwise representation of decimal 0.123456 in memory.

For the map defined in (4), interest focuses on the fractional portion of the IEEE representation since (4) is evaluated between 0 and 1. Henceforth, references to the bits of significand precision will refer to the number of bits explicitly represented in memory.

The significand bits represent all possible seed values x_0 for the map defined in (4). Likewise, the same set represents all possible outcomes. While (4) with $k = 4$ and an appropriately chosen seed (x_0) is aperiodic on (0,1), truncation of x_n to the floating-point representation on a finite precision system limits x_n to the set identified by the significand bits.

sign	exponent	fraction (significand)
0	01111011	11111001101011010000000

Fig. 1. IEEE 754-2008 binary32 representation of 0.123456_{10}.

2.5. Related work

The idea of using the logistic map to build a PRNG has been previously discussed. LOGMAP has been shown to pass the standard statistical tests by subsampling the logistic map and transforming the output to a uniformly distributed, uncorrelated sequence [10]. However, this study did not rigorously test the periodicity; only random seeds were considered resulting in unclear conclusions. Furthermore, the specific approach to testing for periodicity was omitted; consequently, the test in [10] is neither reproducible nor applicable to other recurrence relations.

Andrecut proposes a different transformation of the logistic map to produce uniform, uncorrelated series [3]. The result also passes various statistical tests and appears to be computationally efficient. Nevertheless, the author concludes that these series are aperiodic and infinite, which disregards the adverse effects of an implementation in finite precision. The generator proposed in [3] when implemented on a finite computer is finite and periodic and therefore far less impressive than the original claim of an endless generator.

Our approach to analyzing chaos-based PRNGs, which we name finite precision period calculation (FPPC), aims to provide a universal method for analyzing the period lengths of recurrence relations implemented in finite precision. Our FPPC approach helps evaluate chaos-based PRNGs against their conventional counterparts. Jiang and Wu present a method to efficiently convert series produced by the logistic map into uncorrelated uniform sequences [11]. FPPC complements their study by addressing the periodicity of the logistic map.

3. Finite precision period calculation

A finite precision implementation limits a theoretically aperiodic, infinite series produced by chaotic PRNGs to a periodic, finite series. In this section, we describe an approach, which we call finite precision period calculation (FPPC), to determine the periodic behavior of a map implemented on a finite computer. The FPPC algorithm exhaustively explores a maps periodic behavior across a range of precisions.

3.1. Example

To demonstrate how a map's periods may be calculated, consider an example based on the logistic map. Substituting $k = 4$ into (4) yields the chaotic relation of interest,

$$x_{n+1} = 4x_n(1 - x_n), \quad x_n \in (0,1). \tag{5}$$

Let 4-bit binary fractions represent the set of x_0 (seeds). This demonstrates the effect of a 4-bit floating-point significand. Obviously, this is an extreme simplification that would never be implemented in practice; nevertheless, this example illustrates the calculations that a computer performs for greater bit precisions. There are $2^4 - 1$ or 15 possible seeds (0 and 1 are excluded). Each produces a successive floating-point number, which is then truncated to its closest 4-bit representation. Table 1 summarizes the results of each seed including the sequence up to the first duplicate, the length of the periodic cycle, how many numbers are generated before the cycle (delay), and the total length of the sequence before the generator starts to repeat terms.

Given a finite seed space, the ideal sequence has a period equal to the cardinality of the space. Clearly, the effects of truncation are detrimental to achieving the ideal sequence period. For the 4-bit example, the maximum period of 5 is less than the ideal of 15. Moreover, many of the periods are even shorter, suggesting an extremely inefficient use of the available seed space. This simple case exemplifies the challenges introduced by implementing a PRNG with truncated real numbers. As a result, the same bit depth produces a far less random sequence than an integer-based generator achieves [9]. It is critical to understand the effects of truncation in order to evaluate the amount of randomness a chaotic PRNG is capable of producing.

3.2. Algorithm

The FPPC algorithm performs the afore mentioned calculations for a given recurrence relation. Our reference implementation, written in ANSI C, performs calculations for bit depths up to single precision (binary32). Given a seed range and bit precision, FPPC calculates lengths and delays for each seed (Algorithm 1). Using a range of seeds as input allows multiple copies of the same executable to run subsets of the entire seed space in parallel. As the recurrence relation generates each term, FPPC tracks the output checking for duplicates. We initialize the array of seeds with a symbol not in the seed space as a way to indicate whether or not a term repeats. The seeds array, indexed by each seed's binary fraction representation, stores the time step when each number is generated.

Algorithm 1: Finite precision period calculation

```
Require: min, max, bitsPrecision
  for ($x_n \leftarrow min; x_n \leqslant max; x_n \leftarrow x_n + \epsilon$) do
    seeds[ ] ← {SYMBOL}
    while (!periodic) do
      bf ← float2BinFrac($x_n$)
      seeds[bf] ← n++
      $x_{n+1} \leftarrow 4x_n(1 - x_n)$
      $x_{n+1}$ ← float2BinFrac($x_{n+1}$)
      bf ← trunc($x_{n+1}$, bitsPrecision)
      $x_{n+1}$ ← binFrac2Float(bf)
      if (seeds[bf] == SYMBOL) then
        $x_n = x_{n+1}$
      else
        periodic ← true
      end if
    end while
    calculateCycles(seeds)
  end for
```

Certain design decisions balance trade-offs in flexibility and performance. For example, restricting this algorithm to single precision greatly reduces the data structure complexity. Likewise, calculations are much faster than if designed to accommodate double precision. Contiguous blocks of memory make use of array indexing to efficiently determine when the relation enters a periodic cycle. This is not possible for double precision numbers given the memory available on most computer systems. With some modifications to the data structures, this same approach applies to increased bit depths beyond those presented in this study.

In order to efficiently track previously generated x_n, the binary fraction representation of each number replaces IEEE 754-2008 binary32. The 23-bit representation of each number requires significantly less memory and allows for easy truncation. This is an efficient solution for the logistic map since the seed space is defined as $(0,1)$. An additional sign bit may be optionally added to accommodate other

Table 1
Logistic map series for a 4 bit significand.

Binary fraction	x_0	x_1	$\mathtt{trunc}(x_1)$	Sequence $(x_0, x_1, \ldots, x_{\text{duplicate}})$	Length	Delay	Total
0001	0.0625	0.234375	0.1875	0.0625, 0.1875, 0.5625, 0.9375, 0.1875	3	1	4
0010	0.1250	0.437500	0.4375	0.1250, 0.4375, 0.9375, 0.1875, 0.5625, 0.9375	3	2	5
0011	0.1875	0.609375	0.5625	0.1875, 0.5625, 0.9375, 0.1875	3	0	3
0100	0.2500	0.750000	0.7500	0.2500, 0.7500, 0.7500	1	1	2
0101	0.3125	0.859375	0.8125	0.3125, 0.8125, 0.5625, 0.9375, 0.1875, 0.5625	3	2	5
0110	0.3750	0.937500	0.9375	0.3750, 0.9375, 0.1875, 0.5625, 0.9375	3	1	4
0111	0.4375	0.984375	0.9375	0.4375, 0.9375, 0.1875, 0.5625, 0.9375	3	1	4
1000	0.5000	1.000000	1.0000	0.5000, 1.0000, 1.0000	1	1	2
1001	0.5625	0.984375	0.9375	0.5625, 0.9375, 0.1875, 0.5625	3	0	3
1010	0.6250	0.937500	0.9375	0.6250, 0.9375, 0.1875, 0.5625, 0.9375	3	1	4
1011	0.6875	0.859375	0.8125	0.6875, 0.8125, 0.5625, 0.9375, 0.1875, 0.5625	3	2	5
1100	0.7500	0.750000	0.7500	0.7500, 0.7500	1	0	1
1101	0.8125	0.609375	0.5625	0.8125, 0.5625, 0.9375, 0.1875, 0.5625	3	1	4
1110	0.8750	0.437500	0.4375	0.8750, 0.4375, 0.9375, 0.1875, 0.5625, 0.9375	3	2	5
1111	0.9375	0.234375	0.1875	0.9375, 0.1875, 0.5625, 0.9375	3	0	3

recurrence relations seeded with $(-1, 1)$ at the cost of doubling the memory required to detect duplicates.

3.3. Restricting precision

Another key feature of FPPC is its ability to restrict bit precision lower than single precision. This feature serves two purposes: results verification and trend analysis of period length vs. precision. The implementation of this feature again makes use of the binary fraction number representation. The normalized nature of IEEE 754-2008 floating-point numbers makes it difficult to truncate directly. Conversely, a binary fraction can easily be truncated using a bitmask corresponding to the desired bits.

To illustrate the entire process, consider an example from Table 1, row 4. The decimal 0.1875 is used to seed (5), which is represented in memory as

$$\texttt{0}\quad\texttt{01111100}\quad\texttt{10000000000000000000000.} \tag{6}$$

The resulting term, decimal 0.609375, is represented by

$$\texttt{0}\quad\texttt{01111110}\quad\texttt{00111000000000000000000.} \tag{7}$$

In order to truncate (7) to a 4-bit representation, FPPC converts the binary32 format to a denormalized binary fraction. The float type requires conversion to an unsigned integer containing bitwise representation. Next, right-shift until the exponent bits become isolated and store this result to another variable. At this point, the exponent and signed bit from the original representation are discarded, isolating the mantissa. Now bitwise OR the fraction bits with a 1 in position 23 (little endian) introducing the implicit 1 from the normalized representation, which produces

$$\texttt{0}\quad\texttt{00000001}\quad\texttt{00111000000000000000000.} \tag{8}$$

Subtract the stored exponent bits from the bias (127_{10}) to determine the denormalization shift. Finally, right-shift (8) by this result to produce the resulting binary fraction,

$$\texttt{0}\quad\texttt{00000000}\quad\texttt{10011100000000000000000.} \tag{9}$$

Now, envision the mantissa bits as if they had a leading decimal point. As a sanity check, add the significant fraction bits,

$$\frac{1}{2^1}+\frac{0}{2^2}+\frac{0}{2^3}+\frac{1}{2^4}+\frac{1}{2^5}+\frac{1}{2^6} \tag{10}$$

which of course equals 0.609375, as one would expect.

Truncating the binary fraction representation is quite simple. First, generate a bitmask corresponding to the desired precision. For n bits, left-shift 1 n times and subtract 1. This result gets left-shifted to the most significant bit of the mantissa. The 4-bit truncation mask is

$$\texttt{0}\quad\texttt{00000000}\quad\texttt{11110000000000000000000.} \tag{11}$$

Next, combine (9) and (11) with the bitwise AND operation, producing the truncated binary fraction

$$\texttt{0}\quad\texttt{00000000}\quad\texttt{10010000000000000000000.} \tag{12}$$

The decimal equivalent of (12) is 0.5625, which corresponds to the expected value shown in Table 1.

Lastly, the truncated result requires conversion back to binary32 representation before it can update the recurrence relation. Start by left-shifting the binary fraction until the "implicit" 1 exits the mantissa region. The number of shifts is subtracted from the exponent bias to determine the value of the exponent bits. After discarding the implicit 1, combine the leftover mantissa bits and the generated exponent via bitwise OR. Applying this process to (12) yields,

$$\texttt{0}\quad\texttt{01111110}\quad\texttt{00100000000000000000000.} \tag{13}$$

which is the normalized binary32 representation. Finally, the unsigned integer bits are restored to a float type, thus completing the cycle.

As demonstrated previously, the effects of truncation can be studied manually for reasonable lesser degrees of precision, such as the 4-bit case presented in Table 1. We have verified FPPC results for three, four, and five bits of precision, which correspond, respectively, to base 10 precisions of 0.125 (1/8), 0.0625 (1/16), and 0.03125 (1/32). Using this software truncation technique one can explore the relationship between sequence period length and depth of precision.

3.4. Distributed computing implementation

As FPPC utilizes more bits of precision, the number of possible seeds and outcomes drastically increase. For single precision and beyond the seed space is so large it becomes unreasonable to calculate on a single computer. Single precision calculations performed on a single Intel Core 2 Duo workstation clocked at 2.26 GHz were very time consuming. As the cardinality of the random number space approaches the order of 10^6, calculating all possible outcomes approaches days of computing time instead of hours or minutes. Fortunately, each period calculation is independent of one another for a given seed. Under this condition, seed ranges are easily distributed to multiple nodes in a distributed computing environment without the need for communication between nodes. This enables many nodes to simultaneously determine the period of different subsets of the entire random number space. After FPPC calculates the metrics for each seed (in parallel), a post-processing job aggregates the output files into a unified result. Using this technique, the computation time for higher degrees of precision becomes reasonable.

FPPC is designed to run in a distributed high-throughput environment using Condor middleware [20]. The logistic map example runs on Père, a homogeneous subset of the Marquette University distributed computing grid [16]. Père is comprised of 128 compute nodes each with 8 Intel Nehalem 2.67 GHz cores. Even with the overhead introduced by Condor's job management, single precision FPPC calculations are reduced from several days on a single machine to about 20 min on the cluster. High-throughput computing enables the possibility to test precision beyond the binary32 format. In a parallel environment, random access memory (RAM) becomes a greater limiting factor than processing power. FPPC requires enough memory to represent all possible outcomes in order to detect when repetition occurs. The binary64 standard for floating-point precision representation uses 8 bytes of memory for each number [21]. Given m bytes of available heap memory, FPPC can calculate periods for up to

$$\left\lfloor \log_2\left(\frac{\text{available memory}}{\mathit{sizeof}(\textbf{double})}\right)\right\rfloor = \left\lfloor \log_2\left(\frac{m}{8}\right)\right\rfloor \tag{14}$$

bits of precision. Eq. (14) comes from dividing the available system memory by the size of a double precision number. Flooring the base-2 logarithm of this result converts the actual seed space size into the integer number of bits possible for binary representation without exceeding the system memory limitation. For example, each core on Père, has 3 GiB of physical RAM. After accounting for the operating system kernel, job management overhead, etc. there are approximately 2.75 GiB available for FPPC. Substituting into (14) yields,

$$\left\lfloor \log_2\left(\frac{2.75 * 2^{30}}{8}\right)\right\rfloor = 28\ \text{bits}. \tag{15}$$

As it turns out, 23 bits successfully demonstrates the effects of truncation on the logistic map so it is not necessary to exhaust these limits for this particular example. Of course, (15) assumes memory is allocated contiguously, as it is in this implementation of FPPC. Alternatively, any bookkeeping overhead needs consideration if a noncontiguous data structure replaces the array to improve memory allocation efficiency.

4. Results

FPPC reveals poor periodicity characteristics for the logistic map represented in (5) when implemented as a PRNG. The truncation routine previously described simulates precision varying from 3-bits to 23-bits. Fig. 2 shows box and whiskers plots of the total period lengths with respect to bit depth. The "whiskers" illustrate minimum and maximum lengths and the box outlines the interquartile range (25th to 75th percentile). Moreover, the center line depicts the median length. As indicated by the lower whisker, all depths of precision tested have pathological seeds that result in minimum cycles of length 1 or 2.

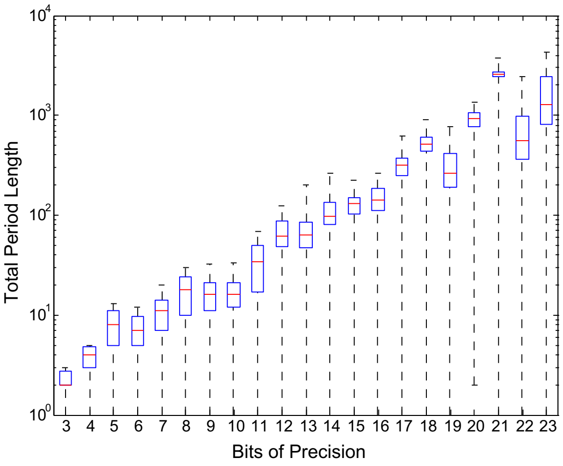

Fig. 2. Logistic map: periodicity vs. bits of precision.

As one might expect, the total period length generally increases with more bits of precision. On some occasions, however, the length actually decreases for a 1-bit increase in precision. This demonstrates that due to truncation effects, an increase in precision does not guarantee an increase in period length. Interestingly, the maximum period length for single precision (23 bit significand) is several orders of magnitude smaller than that of the logistic map's conventional counter parts. Moreover, the linear methods guarantee the maximum length (with correct parameter selection), while the logistic map only produces that sequence for a few select seeds. Fig. 2 also confirms that the logistic map performs much worse than the LCG [9] and the Mersenne Twister [15]. On top of these poor performance characteristics, the logistic map has numerous pathological seeds that should deter anyone from using it as a PRNG where the application requires long sequences of non-repetitive numbers.

For each bit depth, FPPC calculates the delay, length, and total statistics introduced in the four bit example. These arrays represent the number of times each period occurs. For example, the logistic map has seven unique cycles (Table 2) when implemented in single precision (23 bit significand).

Although there are only a small number of periodic cycles, the delay before landing on one of these orbits greatly varies. Combining the various delays with each periodic cycle yields numerous, but a finite number of, total series elements before the generator repeats itself. Fig. 3 shows

Table 2
Finite cycles of the logistic map in single precision.

Period length	Occurrences	Frequency (%)
1	238,675	2.8452
2	502	0.0060
4	204	0.0024
115	49,998	0.5960
123	211,896	2.5260
400	1,677,912	20.0023
487	6,209,420	74.0221

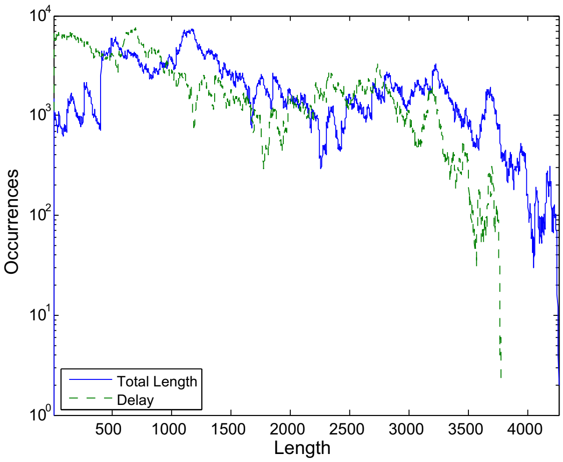

Fig. 3. Finite cycles of the logistic map (single precision).

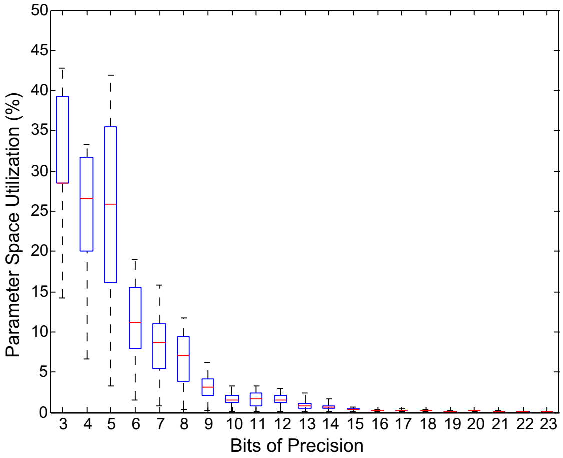

Fig. 4. Parameter space utilization shows the total sequence lengths normalized to the maximum length possible with a certain number of bits.

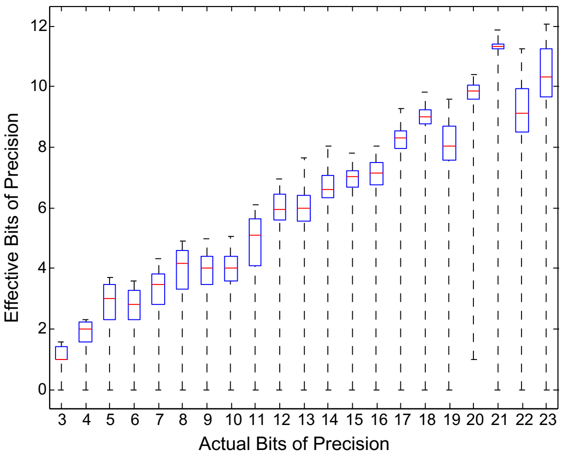

Fig. 5. Effective precision shows the number of bits utilized by the logistic map PRNG with respect to the bit depth of the seed space.

the occurrences of each delay and total cycle lengths for single precision. Although the distribution of lengths is not perfectly uniform, the general trend suggests undesirable short lengths are similarly likely as the most robust lengths. A good PRNG should have rare pathological seeds, which are excluded from use if necessary. The distribution in Fig. 3 suggests the logistic map does not meet this condition.

Phatak and Rao determined the logistic map has six unique periods using 105 seeds chosen by another PRNG [10]. The extra period FPPC finds is likely due to an exhaustive search. Regardless, the similarity between both results is good validation that FPPC produces correct results. However, there is a discrepancy between the exact cycle lengths reported in [10] and those determined by FPPC. Phatak and Rao did not elaborate on their technique so it is difficult to determine the cause of this difference. It is possible that their periodicity detection was based on when the logistic map hits fixed points, $\{0, 0.75, 1\}$. We suspect the study in [10] allowed the use of escalated precision for intermediate calculations. This could explain why Phatak and Rao observed periodicity after approximately 5000 iterations, when truncation to zero occurs. In contrast, FPPC, which finds cycles based on any repeated value, does not detect any periods larger than 4261.

An ideal PRNG never repeats itself; however, an actual implementation cannot meet this characteristic with a finite amount of memory. Nevertheless, the best realistic implementation maximizes the usage of its finite memory allotment. In other words, the sequence length is equal to the size of the seed space for the best-performing realistic generator.

Fig. 4 demonstrates how poorly the logistic map utilizes its available seed space. From Fig. 2 alone it seems reasonable to keep increasing the bit of precision until a desired length results. However, Fig. 4 demonstrates how inefficient that approach is for the logistic map. Due to real number truncation, the utilization of the available space decreases severely. In single precision, the logistic map only uses a tiny fraction of a percent of the maximum possible sequence length. In contrast, integer based generators are capable of guaranteeing 100% utilization before repeating [9,15]. Simply providing the real number-based generator more memory does not increase its efficiency.

Effective precision is another interesting metric for analyzing the PRNG performance. For a good generator, the bits required to represent the sequence length should equal the bits required to represent all possible outcomes. Fig. 5 shows the effective precision produced by the logistic map.

The effective precision is computed by taking the base-2 logarithm of the total period lengths. For almost every bit depth, the bits required to represent the sequence length is less than half the bits available in memory. Again, this trend does not suggest that the efficiency of the logistic map will improve simply by increasing the available memory.

All of these metrics suggest the logistic map performs poorly as a PRNG. It is clear that the truncation of the real number floating-point representation is detrimental to the performance of this generator. Conversely, numerous studies [3,10,11] conclude the logistic map performs adequately based on the results of many standardized statistical tests. This discrepancy is simply due to the fact that the statistical tests pass when applied to a sequence that has not yet entered its periodic cycle. Phatak and Rao explicitly describe rejecting terms beyond the region they determined to be aperiodic [10]. Jiang and Wu use 100,000 iterations of the logistic map implemented in double precision [11]. As a result, the statistical tests pass because the sequences are not periodic unless the length is another order of magnitude larger [10]. In conclusion, the logistic map produces sufficiently uniform and uncorrelated series; nonetheless, the length of these series is inferior relative to what a conventional generator produces given the same amount of memory.

5. Future work

FPPC is primarily limited by the amount of memory required to store sequence history. In order to enable efficient exploration of double precision and beyond, a creative coding scheme is required to represent this data. As computing resources expand it will be necessary to study the effects of finite precision with larger and more complicated number representations. Extending FPPC with modular data structures would enable it to grow alongside the data storage industry. The logistic map has been thoroughly analyzed with respect to each of the ideal PRNG characteristics except for computational speed. Although some authors [3,10,11] have made loose claims about the efficiency of the logistic map, it has yet to be quantitatively benchmarked against other algorithms. Understanding the tradeoffs between computational performance and randomness is key to determining if the logistic map can make up for what it lacks in periodicity with generation speed.

6. Conclusion

Real number implementations in finite precision are detrimental to the periodicity of chaotic PRNGs. Ignoring this reality makes chaos-based PRNGs deceptively appealing for random applications. FPPC algorithm can comprehensively analyze the periodicity of truncated real number series generated by a recurrence relation. Using these results one can make informed decisions about the appropriate use of a chaotic PRNG with respect to its conventional counter-parts. The results revealed about the logistic map do not appear competitive with conventional PRNGs.

Acknowledgment

This research was funded in part by National Science Foundation awards OCI-0923037 “MRI: Acquisition of a Parallel Computing Cluster and Storage for the Marquette University Grid (MUGrid)” and CBET-0521602 “Acquisition of a Linux Cluster to Support College-Wide Research & Teaching Activities.” K. J. Persohn is funded by the Greater Milwaukee Foundation’s Frank Rogers Bacon Fellowship.

References

[1] Alvarez G, Li S, Montoya F, Pastor G, Romera M. Breaking projective chaos synchronization secure communication using filtering and generalized synchronization. Chaos Solitons Fract 2005;24(3):775–83.

[2] Baptista MS. Cryptography with chaos. Phys Lett A 1998;240(1–2):50–4.

[3] Andrecut M. Logistic map as a random number generator. Int J Modern Phys B 1998;12(9):921–30.

[4] Dachselt F, Schwarz W. Chaos and cryptography. IEEE Trans Circ Syst I: Fundam Theory Appl 2001;48(12):1498–509.

[5] Wang F, Zhang Y, Cao T. Research of chaotic block cipher algorithm based on logistic map. In: Second international conference on intelligent computation technology and automation, 2009, ICICTA’09; 2009. p. 678 –81.

[6] Jakimoski G, Kocarev L. Chaos and cryptography: block encryption ciphers based on chaotic maps. IEEE Trans Circ Syst I: Fundam Theory Appl 2001;48(2):163–9.

[7] Xiao D, Liao X, Deng S. One-way hash function construction based on the chaotic map with changeable-parameter. Chaos Solitons Fract 2005;24(1):65–71.

[8] Park SK, Miller KW. Random number generators: good ones are hard to find. Commun ACM 1988;31(10):1192–201.

[9] Knuth DE. The Art of Computer Programming, 2nd ed., vol. 2. Addison Wesley; 1981.

[10] Phatak SC, Rao SS. Logistic map: a possible random-number generator. Phys Rev E 1995;51(4):3670–8.

[11] Jiang C, Wu S. A valid algorithm of converting chaos sequences to uniformity pseudo-random ones. In: International symposium on information engineering and electronic commerce, 2009, IEEC’09; 2009. p. 295–8.

[12] Ferrenberg AM, Landau DP, Wong YJ. Monte Carlo simulations: hidden errors from “good” random number generators. Phys Rev Lett 1992;69(23):3382–4.

[13] Lehmer DH. Mathematical methods in large-scale computing units. Ann Comput Lab Harvard Univ 1951;26:141–6.

[14] Oracle. Java 2 platform se random class. <http://download.oracle.com/javase/1.4.2/docs/api/java/util/Random.html>; 2010.

[15] Matsumoto M, Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. Trans Model Comput Simul 1998;8(1):3–30.

[16] Matsumoto M, Nishimura T, Hagita M, Saito M. Cryptographic Mersenne twister and Fubaki stream/block cipher, cryptology ePrint archive, Report 2005/165; 2005.

[17] Kocarev L. Chaos-based cryptography: a brief overview. IEEE Circ Syst Mag 2001;1(3):6–21.

[18] Alvarez G, Li S. Some basic cryptographic requirements for chaos-based cryptosystems. Int J Bifurcat Chaos 2006;16:2129–51.

[19] Feigenbaum MJ. Quantitative universality for a class of nonlinear transformations. J Stat Phys 1978;19(1):25–52.

[20] Condor high throughput computing. <http://www.cs.wisc.edu/condor/>.

[21] Standard for floating-point arithmetic, Tech. Rep. 754-2008, IEEE; 2008.