Ergodic Pseudo-Random Number Generators

Tatsuki IKEDA*

11th January 2022

Abstract

Mathematical pseudo-random number generators in general have a period, and the length of the period is a measure of the spec of the pseudo-random number generator. The accuracy of Monte Carlo methods depends on the period of the pseudo-random numbers[3], but it is common to use physical random numbers to obtain non-periodic random numbers.

On the other hand, physical random numbers have a high threshold for introduction because they requires the preparation of dedicated equipment or the use of functions that depend on the OS or CPU architecture.

In the previous work by Hiroshi Sugita[8], a random number generator using irrational rotation can generate an infinite period pseudo-random number sequence. However, there is a problem that this random number generator fails to pass some randomness tests. In addition, this random number generator is operated by rational numbers which strictly approximate irrational numbers, and its period is finite in practice. Therefore, by using the ergodic property and adding the bounce operation to the sequence generation, the number of tests that can be passed is increased and the randomness is improved. At the same time, by defining an irrational number in the program, we succeeded in generating a pseudo-random sequence with a truly infinite period.

Keywords— random number generator, non-periodic, ergodic theory, Haskell

1 Introduction

An **Ergodic** pseudo-random number generator(Ergodic PRNG) produces a non-periodic sequence of numbers.

Most of mathematical pseudo-random numbers have a period. However, it is possible to create a sequence of pseudo-random numbers without a period, which requires an infinite number of internal states. The way in which the internal state grows in terms of data could easily make the period infinite, but this would depend on the upper limit of the RAM of machine, which is effectively finite and impractical.

Therefore, it could be possible to solve this problem by making the internal structure mathematically infinite within a finite range. In order to make the internal state mathematically infinite, we have an irrational number as a state in a finite range and use its ergodic property.

Since the Ergodic PRNG has no period, it transcends the long period of the Mersenne Twister[7](period: $2^{19937} - 1$).

*Fourth Year Student, Course of Information, Faculty of Business Administration, Matsuyama University

2 Theory

Definition 1. Let l_x be a maximum number of x-axis, l_y be a maximum number of y-axis, the range of *seed* is the semi-closed interval $[0, l_y)$, then define a pseudo-random number sequence E_n as follows:

$$E_1 = seed \tag{2.0.1}$$
$$E_{n+1} = (E_n + l_x) \bmod l_y \ for \ n = 1, 2, 3, ... \tag{2.0.2}$$

2.1 Proof

We prove in this subsection that the pseudo-random number sequence E_n(2.0.1) has no period. Let $m, n \in \mathbb{N}$ with $m \neq 0$ and $n \neq 0$. Let $\varphi \in \mathbb{R} \setminus \mathbb{Q}$.

Theorem 1. *E_n(2.0.1) has no period, if l_x / l_y is irrational.*

To prove this theorem, prove follows lemma when $l_x = \varphi$ and $l_y = 1$ without loss of generality.

Lemma 2. *There are no $m \in \mathbb{N}$ and $n \in \mathbb{N}$ such that*

$$\exists (m\varphi, n) \in \{(x, x) \in \mathbb{R} \times \mathbb{R}\}, \ exept \ for \ (m, n) = (0, 0) \tag{2.1.1}$$

Proof. To prove it, assume that there exist $(m, n) \in \mathbb{N} \times \mathbb{N}$ such that $(m\varphi, n) \in \{(x, x) \in \mathbb{R} \times \mathbb{R}\}$

Substitute $(m\varphi, n)$ into (x, x).

$$n = m\varphi$$
$$\frac{n}{m} = \varphi, \ sincem \neq 0$$

On the other hand, we know $\frac{n}{m} \in \mathbb{Q}$ in spite of $\varphi \in \mathbb{R} \setminus \mathbb{Q}$.

Our argument caused, a contradiction.

Therefore, the initial assumption (2.1.1) must be false, which is our desired conclusion. □

2.2 Geometrical Bouncing

To make Ergodic PRNGs more randomise, we use value bouncing process. Geometrically, when a point starts moving at an angle of $\frac{\pi}{4}$ radian and reaches $x = 0, l_x$ or $y = 0, l_y$, it makes a billiard-like reversal. In other words, this pseudo-random sequence starts from the origin in a rectangle with side lengths l_x and l_y, respectively, at an angle of $\frac{\pi}{4}$ radian, and plots the coordinates (x, y) of the point at which y moves a distance of $l_y\sqrt{2}$ while continuously bouncing like a billiard ball. This bounce process is equivalent to the torus construction of the graph, and the infinite period is preserved.

The gradual formula of the pseudo-random number sequence E_n with the bounce process added is as follows:

$$
E_{n+1} = \begin{cases}
\begin{array}{l} \mathit{seed}, \\ \quad x \text{ shall not bounce} \end{array} & \text{for } n = 0 \qquad (2.2.1) \\[2ex]
\begin{array}{l} (E_n + l_x) \bmod l_y, \\ \quad x \text{ shall not bounce} \end{array} & \begin{cases} \text{for } x \text{ does not bounced,} \\ \text{and } \left\lfloor \dfrac{E_n + l_x}{l_y} \right\rfloor \text{ is even.} \end{cases} \qquad (2.2.2) \\[2ex]
\begin{array}{r} l_y - \{(E_n + l_x) \bmod l_y\}, \\ x \text{ shall bounce} \end{array} & \begin{cases} \text{for } x \text{ does not bounced,} \\ \text{and } \left\lfloor \dfrac{E_n + l_x}{l_y} \right\rfloor \text{ is odd.} \end{cases} \qquad (2.2.3) \\[2ex]
\begin{array}{l} (-E_n + l_x) \bmod l_y, \\ \quad x \text{ shall not bounce} \end{array} & \begin{cases} \text{for } x \text{ bounced,} \\ \text{and } \left\lfloor \dfrac{-E_n + l_x}{l_y} \right\rfloor \text{ is even.} \end{cases} \qquad (2.2.4) \\[2ex]
\begin{array}{r} l_y - \{(E_n + l_x)\{\mathrm{mod} l_y, \\ x \text{ shall bounce} \end{array} & \begin{cases} \text{for } x \text{ bounced,} \\ \text{and } \left\lfloor \dfrac{-E_n + l_x}{l_y} \right\rfloor \text{ is odd.} \end{cases} \qquad (2.2.5)
\end{cases}
$$

3 Application to Ergodic PRNG

From now on, based on the above Theorem 1, we will actually apply it to the algorithm the pseudo-random number generator. The rest of the definition is based on Haskell[5] and Backus-Naur form[1].

3.1 Why do we do in Haskell

Haskell is a general-purpose, statically typed, purely functional programming language with type inference and lazy evaluation[5].

3.2 Irrational Expression Type

In Haskell, it is easy to build new data structure. In order to implement Ergodic PRNG, we first implement an **Irrational Expression** type in our code. Let an Irrational Expression be:

$$a + b\varphi. \tag{3.2.1}$$

Where $a, b \in \mathbb{Q}$ and $\varphi \in \mathbb{R} \setminus \mathbb{Q}$.
First, we define a new data type in Backus-Naur form.

$$
\begin{array}{rcl}
< \mathit{digit} > & ::= & (0|1|2|3|4|5|6|7|8|9) \\
< \mathit{digits} > & ::= & < \mathit{digit} > \\
 & | & < \mathit{digit} >< \mathit{digit} > \\
< \mathit{rational} > & ::= & < \mathit{digits} > \% < \mathit{digits} > \\
< \mathit{exp} > & ::= & < \mathit{rational} > + < \mathit{rational} > \varphi
\end{array}
$$

Implementation in Haskell is showed as below:

```
data Irrational = Irrational Rational -- ^ Rational term of a
                             Rational -- ^ Rational term of b
```

```haskell
                             deriving ( Eq )

instance Show Irrational where
    ... -- Complete Code is contained in Appendix A

instance Num Irrational where
    ... -- Complete Code is contained in Appendix A

toFloatingIr :: Floating a
             => Irrational
             -> a
toFloatingIr =  fromRational . toRationalIr

toRationalIr                     :: Irrational
                                 -> Rational
toRationalIr (Irrational a b) =  a + (toRational b) * phi

instance Ord Irrational where
    ... -- Complete Code is contained in Appendix A
```

Note that in Haskell, the type **Integer** means arbitrary-precision integers and the type **Rational** means rational numbers contains a pair of **Integers.** It is important to realize truly non-periodic pseudo-random number generators.

3.3 Generator State Type

Next, we define a type to represent the state of the pseudo-random number generator. Since Ergodic PRNGs perform bounce, **Generator State** type must contain a boolean type representing the state of the bounce in addition to the Irrational Expression type. The definition in Backus-Naur form is expressed as follows:

$$\begin{array}{rcl} < bool > & ::= & true \\ & | & false \\ < gen > & ::= & < exp >, < bool > \end{array}$$

Implementation using Haskell is shown as below:

```haskell
data Ergodic = Ergodic Irrational
                       Bool
                       deriving ( Eq )

instance Show Ergodic where
  show (Ergodic i True ) = show i ++ ", Not bounced"
  show (Ergodic i False) = show i ++ ", Bounced"

instance RandomGen Ergodic where
    genWord32 gen@(Ergodic seed _) = ( mapIntIr False
                                                 32
                                                 seed
                                     , next      gen )

mapIntIr        :: Integral a
```

```haskell
                    => Bool         -- ^ Is signed
                    -> Int          -- ^ Numbers of bits
                    -> Irrational
                    -> a
mapIntIr s i r =    floor ((toFloatingIr r) * (mb s i))

mb             :: Floating a
               => Bool -- ^ Is signed
               -> Int  -- ^ Numbers of bits
               -> a
mb True   8    =  fromIntegral (maxBound :: Int8)
mb True   16   =  fromIntegral (maxBound :: Int16)
mb True   32   =  fromIntegral (maxBound :: Int32)
mb True   64   =  fromIntegral (maxBound :: Int64)
mb False  8    =  fromIntegral (maxBound :: Word8)
mb False  16   =  fromIntegral (maxBound :: Word16)
mb False  32   =  fromIntegral (maxBound :: Word32)
mb False  64   =  fromIntegral (maxBound :: Word64)
mb False  256  =  fromIntegral (maxBound :: Word256)
```

To get the next value of E_n, define a function as follows:

```haskell
divIr     :: Irrational -> Irrational -> Irrational
divIr a b -- Complete Code is contained in Appendix A

modIr     :: Irrational -> Irrational -> Irrational
modIr a b -- Complete Code is contained in Appendix A

evenIr   :: Irrational -> Bool
evenIr r =  -- Complete Code is contained in Appendix A

oddIr   :: Irrational -> Bool
oddIr r =  -- Complete Code is contained in Appendix A

next     :: Ergodic -> Ergodic
next gen =  if bounce
               then Ergodic ns        True
               else Ergodic (ly - ns) False
             where Ergodic s b = gen
                   ln = lx
                   (ns, bounce)
                      =  if b
                           then ( modIr  (s + ln)
                                         ly
                                , evenIr (divIr (s + ln)
                                                ly) )
                           else ( modIr (ly - s + ln)
                                        ly
                                , oddIr (divIr (ly - s + ln)
                                                ly) )
```

3.4 Initializing generator

Next, we consider the initialization of the pseudo-random number generator. In the definition (2.2.1), the seed is directly divided by the maximum value of 64bit signed integer, but there is a problem with this. That is, it generates almost the same sequence of random numbers when the seed values are close. For this reason, Ergodic PRNGs use Xorshift[6] to randomise the seed as follows:

$$E_1 = \frac{Xorshift(seed)}{2^{63} - 1} \tag{3.4.1}$$

In Haskell:

```haskell
mkErgoGen       :: Int          -- ^ Seed
                -> Irrational   -- ^ Initialised generator
mkErgoGen seed  =  Irrational (toRational ((xorshift seed)
                                          % maxBound))
                              0
```

4 Choose the optimal combination of parameters

So far, we have considered the part of the Ergodic PRNG algorithm that is related to the computational process. We now consider the actual parameters used in the calculation.

In Ergodic PRNG, three parameters, l_x that the maximum value of x-axis, l_y that the maximum value of y-axis and φ are required. These parameters are defined as constants in advance.

4.1 The maximum value of y-axis

First of all, consider the maximum value of y-axis l_y. where l_y is the maximum possible value of the y-axis in the graph. It must be $l_y \in \mathbb{R}$ and $l_y > 0$. For simplicity, we assume $l_y = 1$.

4.2 φ and the maximum value of x-axis

For more randomness, the choice of φ and the maximum value of x-axis l_x is crucial.

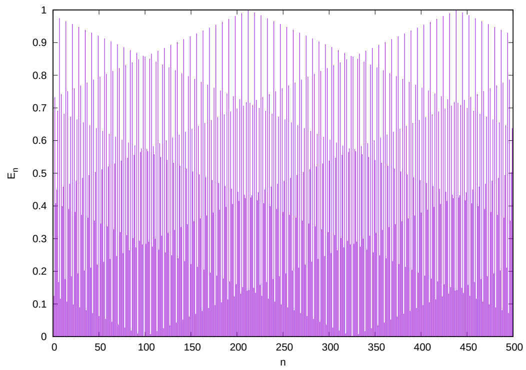

Figure 1: $l_x = \sqrt{2}$

Figure 1 plots the output results for the case $l_x = \sqrt{2}$. As can be seen from this graph, the outer period is easily seen. The more the graph looks like a gradient, the higher the randomness of the pseudo-random number sequence. So, E_n with $l_x = \sqrt{2}$ has low randomness.

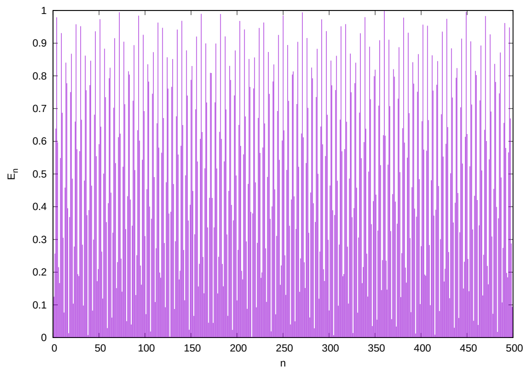

Figure 2: $l_x = \frac{1+\sqrt{5}}{2}$

Figure 2 plots the output of l_x as the golden ratio ($\frac{1+\sqrt{5}}{2}$). This is more random than the previous Figure 1.

4.3 Pick an Irrational Number

From Fermat's Last Theorem[9], the number case $n = 3$ is expressed as below:

$$\nexists x, y \in \mathbb{Q} \text{ s.t. } x^3 + y^3 = 1 \tag{4.3.1}$$

except for the trivial case $(x, y) = (0, 1), (1, 0)$

Therefore, the graph of $x^3 + y^3 = 1$ has no rational solutions except $(1, 0)$ and $(0, 1)$, and all points except $(1, 0)$ and $(0, 1)$ through which the graph passes are irrational numbers. Moreover, the cubic roots do not circulate in the continuous fraction expansion. In view of the above, it is thought that the cubic root of φ would give a better result.

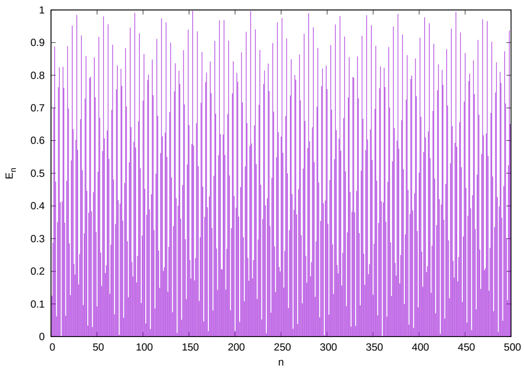

Figure 3: $l_x = \sqrt[3]{4}$

Figure 3 shows a plot of the output as $l_x = \sqrt[3]{4}$. Although the pattern is finer than in Figure 1, it is more linear than that in Figure 2. This suggests that the rational term should also be a non-zero number.

The outer period is determined by the denominator of the approximation by rational numbers after the decimal point. Therefore, it is necessary to find a value which is as difficult to approximate by rational numbers as possible. To find the best value, we plot $0 \leq \frac{m}{n} \leq 1$ with $m \in \{x \in \mathbb{N} | x < n\}$ and $n \in \mathbb{N}$.

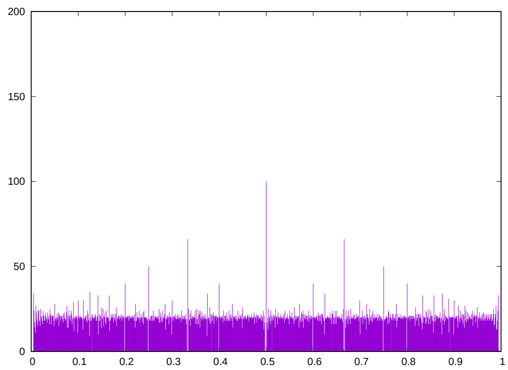

Figure 4: $0 \leq \frac{m}{n} \leq 1$ $(n = 200)$

Figure 4 is a graph showing the frequency of occurrence for the case $n = 200$. From the graph, we can see that there is a hole around 0.64.

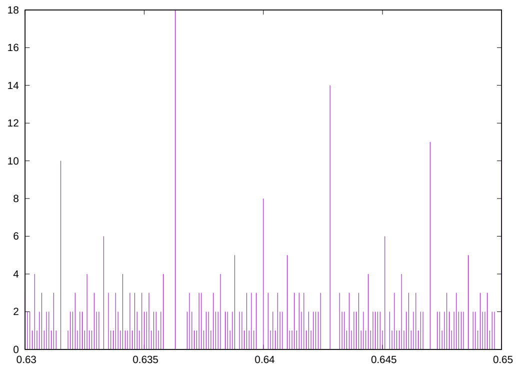

Figure 5: $0.63 \leq \frac{m}{n} \leq 0.65$ $(n = 200)$

Figure 5 is an enlargement of Figure 4 in the range $0.63 \leq \frac{m}{n} \leq 0.65$ for better clarity. From the above result, we generate a sequence of pseudo-random numbers with

$$l_x = \frac{1 + \sqrt[3]{12}}{2} \approx 1.644714. \tag{4.3.2}$$

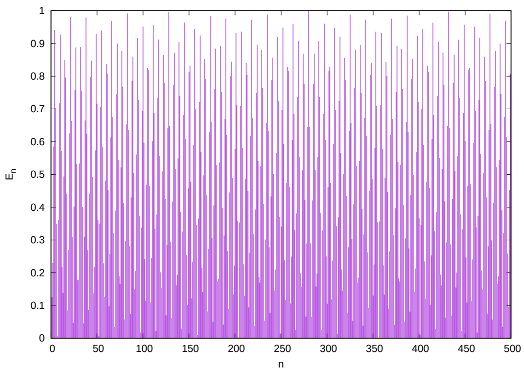

Figure 6: $l_x = \frac{1+\sqrt[3]{12}}{2}$

Figure 6 shows a plot of the output as $l_x = \frac{1+\sqrt[3]{12}}{2}$. This is an optimal result because the graph looks the most gradated among the ones we tried.

Based on the above results, we set $\varphi = \sqrt[3]{12}$ and $l_x = \frac{1+\sqrt[3]{12}}{2}$. In order to obtain fast and accurate results, we approximate φ by using the continued fraction expansion.

$$\sqrt[3]{12} = 2+\cfrac{1}{3+\cfrac{1}{2+\cfrac{1}{5+\cfrac{1}{15+\cfrac{1}{7+\cfrac{1}{3+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{96+\cfrac{1}{\ddots}}}}}}}}}}}}} \tag{4.3.3}$$

$$\approx \frac{53415281}{23331273} \tag{4.3.4}$$

4.4 Definition in Haskell

The parameters used in Ergodic PRNG to generate random numbers are as follows:

$$l_y = 1 \tag{4.4.1}$$

$$\varphi \approx \frac{53415281}{23331273} \tag{4.4.2}$$

$$l_x = \frac{1+\varphi}{2} \tag{4.4.3}$$

We now define these parameters as constants in Haskell as follows:

```haskell
ly :: Irrational
ly =  Irrational 1 0 -- 1 + 0φ

phi :: Rational
phi =  53415281 % 23331273 -- ∛12 ≈ 53415281/23331273

lx :: Irrational
lx =  Irrational (1 % 2) (1 % 2) -- 1/2 + φ/2
```

This completes the implementation of the Ergodic PRNG. The actual sequence of pseudo-random numbers generated according to this algorithm is shown in Appendix B.

5 Benchmarks

We now estimate benchmark tests on the performance of Ergodic PRNG. The benchmark tests are performed on two aspects: statistical randomness and its generation speed.

5.1 Randomness test

For the tests, we used the Python implementation of NIST 800-22 test suite[2][4]. The following Table 1 shows the results of testing a pseudo-random number sequence generated by Ergodic PRNG using NIST 800-22.

Table 1: Test result of NIST 800-22

Test name	Value	Result
monobit test	0.7286253077289306	PASS
frequency within block test	0.12149948119993939	PASS
runs test	0.9819908995076131	PASS
longest run ones in a block test	0.11532499498977329	PASS
binary matrix rank test	0	FAIL
dft test	0	FAIL
non overlapping template matching test	-0.8446845109937807	FAIL
overlapping template matching test	0.05142128621666958	PASS
maurers universal test	1.3011979903245765e-70	FAIL
linear complexity test	0.006549021060733038	FAIL
serial test	0.9017749891453991	PASS
approximate entropy test	0.9968778600697635	PASS
cumulative sums test	0.9821000522967793	PASS
random excursion test	0.2778791285352132	PASS
random excursion variant test	0.2001052592727655	PASS

The above results show that its randomness is not perfect although the Ergodic PRNG passes more than half of the tests.

5.2 Generation speed

The computational environment used for the speed measurements is shown in Table 2 below.

Table 2: Environment

OS	Red Hat Enterprise Linux release 8.5 (Ootpa)
CPU	Intel Core i7 4790K
RAM	32GB
Haskell Stack	Version 2.7.3
Stackage	LTS Haskell 18.21
Compiler	GHC 8.10.7

As a measure of speed evaluation, we compare the generation speed of Xorshift and RDRAND with that of Ergodic PRNG. RDRAND is an instruction for returning random numbers from the Intel on-chip hardware random number generator which has been seeded by an on-chip entropy source.

Table 3 shows the generation speed results of the three algorithms for the case $n = 1000000$. The standard time command of Red Hat Enterprise Linux was used for the speed measurement.

Table 3: Time and speed

Algorithm	Time	Speed(bps)
Ergodic PRNG	1:09.75	458,781.3620
Xorshift	0:02.26	14,159,292
RDRAND	0:03.38	9,467,455.621

As can be seen from the above results, the speed of Ergodic PRNG is more than 20 times faster than RDRAND and more than 30 times faster than Xorshift. This is probably due to the large data structure and the large number of arithmetic operations including division. Speeding up the generation process is one of the remaining challenges of Ergodic PRNG.

6 Summary

Ergodic PRNG is a periodless mathematical pseudo-random number generator. While it has the property of no period, it is inferior to RDRAND and Xorshift in terms of randomness and generation speed. On the other hand, it has a merit that it is possible to obtain a sequence of random numbers of arbitrary precision from the same algorithm because it is internally an irrational number.

Although the generation speed and the randomness are major issues to be solved in the future, there is a way to solve the randomness problem by using the Ergodic PRNG as a seed only, instead of using it as a pseudo-random sequence. By using the output of Ergodic PRNG as a seed, it is possible to generate a pseudo-random number sequence by another pseudo-random number generator, which eliminates the period of the existing pseudo-random number generator.

References

[1] John W. Backus, Friedrich L. Bauer, Julien Green, Charles Katz, John McCarthy, Peter Naur, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. "Report on the Algorithmic Language ALGOL 60". *Numerische Mathematik*, 2(1):106–136, 1960.

[2] Lawrence E. Bassham, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. "SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications". Technical report, Gaithersburg, MD, USA, 2010.

[3] Yuya DAN. "A Theoretical and Practical Study for Monte Carlo Simulations with Quantum Random Numbers," in Japanese. *MATSUYAMA UNIVERSITY REVIEW*, 27(4-2):121–152, 2015.

[4] dj-on-github. sp800_22_tests. https://github.com/dj-on-github/sp800_22_tests.

[5] Simon Peyton Jones. *"Haskell 98 Language and Libraries: The Revised Report"*. Cambridge University Press, 2003.

[6] George Marsaglia et al. "Xorshift RNGs". *Journal of Statistical Software*, 8(14):1–6, 2003.

[7] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator". *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[8] Hiroshi Sugita. "Pseudo-random number generation by irrational number rotation". 915:146–156, 1995.

[9] Andrew Wiles. "Modular elliptic curves and Fermat's Last Theorem". *Annals of mathematics*, 141(3):443–551, 1995.

Appendix A Complete Code in Haskell

In this part, specific operations that are not mathematically relevant in nature are omitted from the source code. The complete Haskell code is written as follows:

```haskell
module ErgodicPRNG where

import System.Random ( RandomGen ()
                     , genWord32 )
import Data.Int      ( Int8
                     , Int16
                     , Int32
                     , Int64 )
import Data.Word     ( Word8
                     , Word16
                     , Word32
                     , Word64 )
import Data.WideWord ( Word256 )
import Data.Ratio    ( (%) )
import GHC.Real      ( Ratio ( (:%) ) )

import Data.Function ( (&) )
import Data.Bits     ( shiftL
                     , shiftR
                     , xor )

data Irrational = Irrational Rational -- ^ Rational term of a
                             Rational -- ^ Rational term of b
                             deriving ( Eq )

instance Show Irrational where
    show (Irrational 0 0) = "0"
    show (Irrational 0 b) = show b ++ " * phi"
    show (Irrational a 0) = show a
    show (Irrational a b) = show a ++ " + " ++
                            show b ++ " * phi"

instance Num Irrational where
    (+) (Irrational a b) -- a + bφ + c + dφ
        (Irrational c d) = Irrational (a + c)
                                      (b + d)

    signum r | r' > 0     =  1
             | r' < 0     = -1
             | otherwise  =  0
             where r'     = toRationalIr r

    negate (Irrational a b) = Irrational (-a) (-b)

    abs r = r * signum r

    fromInteger a = Irrational (a % 1) 0
```

```haskell
toFloatingIr :: Floating a
             => Irrational
             -> a
toFloatingIr =  fromRational . toRationalIr

toRationalIr                   :: Irrational
                               -> Rational
toRationalIr (Irrational a b) =  a + (toRational b) * phi

instance Ord Irrational where
    compare (Irrational a b)
            (Irrational c d) | n == 0    = EQ
                             | n >  0    = GT
                             | otherwise = LT
                               where n   = (a - c) +
                                           ((b - d) * phi)

divIr     :: Irrational -> Irrational -> Irrational
divIr a b | a >= b = 1 + divIr (a - b) b
          | a <  b = 0

modIr     :: Irrational -> Irrational -> Irrational
modIr a b | a == b = Irrational 0 0
          | a >  b = modIr (a - b) b
          | a <  b = a

evenIr   :: Irrational -> Bool
evenIr r =  even (n `div` d)
            where n :% d = toRationalIr r

oddIr   :: Irrational -> Bool
oddIr r =  not (evenIr r)

phi :: Rational
phi =  53415281 % 23331273 -- ∛12 ≈ 53415281/23331273

ly :: Irrational
ly =  Irrational 1 0 -- 1 + 0φ

lx :: Irrational
lx =  Irrational (1 % 2) (1 % 2) -- 1/2 + φ/2

data Ergodic = Ergodic Irrational
                       Bool
                       deriving ( Eq )

instance Show Ergodic where
  show (Ergodic i True ) = show i ++ ", Not bounced"
  show (Ergodic i False) = show i ++ ", Bounced"

instance RandomGen Ergodic where
    genWord32 gen@(Ergodic seed _) = ( mapIntIr False
                                                32
```

```haskell
                                                seed
                                     , next     gen )

mapIntIr       :: Integral a
               => Bool          -- ^ Is signed
               -> Int           -- ^ Numbers of bits
               -> Irrational
               -> a
mapIntIr s i r =  floor ((toFloatingIr r) * (mb s i))

mb           :: Floating a
             => Bool -- ^ Is signed
             -> Int  -- ^ Numbers of bits
             -> a
mb True  8   =  fromIntegral (maxBound :: Int8)
mb True  16  =  fromIntegral (maxBound :: Int16)
mb True  32  =  fromIntegral (maxBound :: Int32)
mb True  64  =  fromIntegral (maxBound :: Int64)
mb False 8   =  fromIntegral (maxBound :: Word8)
mb False 16  =  fromIntegral (maxBound :: Word16)
mb False 32  =  fromIntegral (maxBound :: Word32)
mb False 64  =  fromIntegral (maxBound :: Word64)
mb False 256 =  fromIntegral (maxBound :: Word256)

xorshift   :: Int -> Int
xorshift s =  s & (\v -> (v `shiftL` 23) `xor` v)
                & (\v -> (v `shiftR` 13) `xor` v)
                & (\v -> (v `shiftL` 58) `xor` v)

next     :: Ergodic -> Ergodic
next gen =  if bounce
               then Ergodic ns        True
               else Ergodic (ly - ns) False
             where Ergodic s b = gen
                   ln = lx
                   (ns, bounce)
                      =  if b
                         then ( modIr  (s + ln)
                                       ly
                              , evenIr (divIr (s + ln)
                                              ly) )
                         else ( modIr (ly - s + ln)
                                      ly
                              , oddIr (divIr (ly - s + ln)
                                              ly) )

mkErgoGen      :: Int           -- ^ Seed
               -> Irrational    -- ^ Initialised generator
mkErgoGen seed =  Irrational (toRational ((xorshift seed)
                                      % maxBound))
                             0
```

Appendix B A List of Generated Random Numbers

The table of random numbers generated by the Ergodic PRNG is shown below. Note that $seed = 4$ and the range is the semi-closed interval [0,1).

$n \leq 50$	Value	$n \leq 100$	Value	$n \leq 150$	Value
1	0.12500000	51	0.36071213	101	0.59642426
2	0.23028576	52	0.00543	102	0.24113850
3	0.58557151	53	0.34985939	103	0.11414726
4	0.94085727	54	0.70514514	104	0.46943302
5	0.70385697	55	0.93956910	105	0.82471877
6	0.34857121	56	0.58428334	106	0.81999547
7	0.00671	57	0.22899758	107	0.46470971
8	0.36200030	58	0.12628817	108	0.10942395
9	0.71728606	59	0.48157393	109	0.24586180
10	0.92742818	60	0.83685969	110	0.60114756
11	0.57214243	61	0.80785455	111	0.95643332
12	0.21685667	62	0.45256880	112	0.68828092
13	0.13842909	63	0.0973	113	0.33299517
14	0.49371485	64	0.25800272	114	0.0223
15	0.84900060	65	0.61328848	115	0.37757635
16	0.79571364	66	0.96857423	116	0.73286211
17	0.44042788	67	0.67614001	117	0.91185214
18	0.0851	68	0.32085425	118	0.55656638
19	0.27014363	69	0.0344	119	0.20128062
20	0.62542939	70	0.38971726	120	0.15400514
21	0.98071515	71	0.74500302	121	0.50929089
22	0.66399909	72	0.89971122	122	0.86457665
23	0.30871334	73	0.54442546	123	0.78013759
24	0.0466	74	0.18913971	124	0.42485183
25	0.40185818	75	0.16614605	125	0.0696
26	0.75714394	76	0.52143181	126	0.28571968
27	0.88757031	77	0.87671757	127	0.64100544
28	0.53228455	78	0.76799668	128	0.99629120
29	0.17699879	79	0.41271092	129	0.64842305
30	0.17828697	80	0.0574	130	0.29313729
31	0.53357272	81	0.29786060	131	0.0621
32	0.88885848	82	0.65314635	132	0.41743423
33	0.75585576	83	0.99156789	133	0.77271998
34	0.400570	84	0.63628213	134	0.87199426
35	0.0453	85	0.28099637	135	0.51670850
36	0.31000151	86	0.0743	136	0.16142274
37	0.66528727	87	0.42957514	137	0.19386301
38	0.97942697	88	0.78486090	138	0.54914877
39	0.62414122	89	0.85985334	139	0.90443453
40	0.26885546	90	0.50456759	140	0.74027971
41	0.0864	91	0.14928183	141	0.38499396
42	0.44171606	92	0.20600393	142	0.0297
43	0.79700181	93	0.56128969	143	0.32557756
44	0.84771243	94	0.91657544	144	0.68086331
45	0.49242667	95	0.72813880	145	0.96385093
46	0.13714091	96	0.37285304	146	0.60856517
47	0.21814484	97	0.0176	147	0.25327941
48	0.57343060	98	0.33771847	148	0.10200634
49	0.92871636	99	0.69300423	149	0.45729210
50	0.71599789	100	0.95171001	150	0.81257786

$n \leq 200$	Value	$n \leq 250$	Value	$n \leq 300$	Value
151	0.83213638	201	0.93215149	251	0.69643936
152	0.47685063	202	0.71256275	252	0.94827488
153	0.12156487	203	0.3572770	253	0.59298912
154	0.23372089	204	0.00199	254	0.23770337
155	0.58900665	205	0.35329452	255	0.11758239
156	0.94429240	206	0.70858028	256	0.47286815
157	0.70042184	207	0.93613397	257	0.82815391
158	0.34513608	208	0.58084821	258	0.81656034
159	0.0101	209	0.22556245	259	0.46127458
160	0.36543543	210	0.12972331	260	0.10598882
161	0.72072119	211	0.48500906	261	0.24929694
162	0.92399305	212	0.84029482	262	0.60458269
163	0.56870729	213	0.80441942	263	0.95986845
164	0.21342154	214	0.44913366	264	0.68484579
165	0.14186422	215	0.0938	265	0.32956003
166	0.49714998	216	0.26143785	266	0.0257
167	0.85243574	217	0.61672361	267	0.38101148
168	0.79227851	218	0.97200937	268	0.73629724
169	0.43699275	219	0.67270488	269	0.9084170
170	0.0817	220	0.31741912	270	0.55313125
171	0.27357877	221	0.0379	271	0.19784549
172	0.62886452	222	0.39315240	272	0.15744027
173	0.98415028	223	0.74843815	273	0.51272603
174	0.66056396	224	0.89627609	274	0.86801178
175	0.30527820	225	0.54099033	275	0.77670246
176	0.05	226	0.18570457	276	0.42141670
177	0.40529331	227	0.16958118	277	0.0661
178	0.76057907	228	0.52486694	278	0.28915481
179	0.88413517	229	0.88015270	279	0.64444057
180	0.52884942	230	0.76456154	280	0.99972633
181	0.17356366	231	0.40927579	281	0.64498791
182	0.18172210	232	0.054	282	0.28970216
183	0.53700786	233	0.30129573	283	0.0656
184	0.89229361	234	0.65658149	284	0.42086936
185	0.75242063	235	0.98813276	285	0.77615511
186	0.39713487	236	0.6328470	286	0.86855913
187	0.0418	237	0.27756124	287	0.51327337
188	0.31343664	238	0.0777	288	0.15798761
189	0.66872240	239	0.43301027	289	0.19729814
190	0.97599184	240	0.78829603	290	0.55258390
191	0.62070609	241	0.85641821	291	0.90786966
192	0.26542033	242	0.50113246	292	0.73684458
193	0.0899	243	0.14584670	293	0.38155883
194	0.44515119	244	0.20943906	294	0.0263
195	0.80043694	245	0.56472482	295	0.32901269
196	0.84427730	246	0.92001057	296	0.68429845
197	0.48899154	247	0.72470367	297	0.96041580
198	0.13370578	248	0.36941791	298	0.60513004
199	0.22157997	249	0.0141	299	0.24984428
200	0.57686573	250	0.34115360	300	0.10544148