PROIECT DE DIPLOMĂ

EDUARD-TIBERIU GANEA

COORDONATOR ȘTIINȚIFIC

SEF LUCRARI DOCTOR INGINER MARIUS MARIAN

IULIE 2021

IMPLEMENTATION OF A PSEUODORANDOM NUMBER GENERATOR (USING LCG, LFG, LFSR TECHNIQUES)

EDUARD-TIBERIU GANEA

COORDONATOR ȘTIINȚIFIC

SEF LUCRARI DOCTOR INGINER MARIUS MARIAN

IULIE 2021

Subsemnatul EDUARD-TIBERIU GANEA, student la specializarea CALCULATOARE IN LIMBA ENGLEZA din cadrul Facultății de Automatică, Calculatoare și Electronică a Universității din Craiova, certific prin prezenta că am luat la cunoştinţă de cele prezentate mai jos şi că îmi asum, în acest context, originalitatea proiectului meu de licenţă:

- cu titlul IMPLEMENTATION OF A PSEUDORANDOM NUMBER GENERATOR (USING LCG, LFG, LFSR TECHNIQUES),
- coordonată de SEF LUCRARI DOCTOR INGINER MARIUS MARIAN,
- prezentată în sesiunea IULIE 2021.

La elaborarea proiectului de licenţă, se consideră plagiat una dintre următoarele acţiuni:

- reproducerea exactă a cuvintelor unui alt autor, dintr-o altă lucrare, în limba română sau prin traducere dintr-o altă limbă, dacă se omit ghilimele şi referinţa precisă,
- redarea cu alte cuvinte, reformularea prin cuvinte proprii sau rezumarea ideilor din alte lucrări, dacă nu se indică sursa bibliografică,
- prezentarea unor date experimentale obţinute sau a unor aplicaţii realizate de alţi autori fără menţionarea corectă a acestor surse,
- însuşirea totală sau parţială a unei lucrări în care regulile de mai sus sunt respectate, dar care are alt autor.

Pentru evitarea acestor situaţii neplăcute se recomandă:

- plasarea între ghilimele a citatelor directe şi indicarea referinţei într-o listă corespunzătoare la sfărşitul lucrării,
- indicarea în text a reformulării unei idei, opinii sau teorii şi corespunzător în lista de referinţe a sursei originale de la care s-a făcut preluarea,
- precizarea sursei de la care s-au preluat date experimentale, descrieri tehnice, figuri, imagini, statistici, tabele et caetera,
- precizarea referinţelor poate fi omisă dacă se folosesc informaţii sau teorii arhicunoscute, a căror paternitate este unanim cunoscută și acceptată.

Data, Semnătura candidatului,

07.06.2021

PROIECTUL DE DIPLOMĂ

Numele și prenumele studentului/-ei:	Eduard-Tiberiu Ganea
Enunțul temei:	Implementation of a pseudorandom number generator (using LCG, LFG, LFSR Techniques)
Datele de pornire:	Definitia de baza a unui generator de numbere random si ce inseamna acesta in sine
Conținutul proiectului:	Proiectul o sa aiba 4 mari capitole, 3 dintre acestea fiind impartite in subcapitole: LCG-1. Definitie de baza 2. Metode de implementare 3. Pseduocod 4. Avantaje/dezavantaje LFG LFSR Comparatie
Material grafic obligatoriu:	
Consultații:	Periodice
Conducătorul științific (titlul, nume și prenume, semnătura):	Șef lucrări dr. ing. Marius MARIAN
Data eliberării temei:	01.12.2011
Termenul estimat de predare a proiectului:	01.06.2012
Data predării proiectului de către student și semnătura acestuia:	

REZUMATUL PROIECTULUI

Lucrarea prezinta implementarea a 3 generatoare de numbere pseudorandom. Pentru cele 3 generatoare in cauza sunt folosite 3 metode diferite:LCG, LFG, LFSR. Fiecare metoda este prezentata avand si codul in cauza. In functie de metoda am folosit un limbaj de programare adecvat precum si o descriere in partea documentatiei pe masura. Toate folosind o relatie diferita pe baza careia termenii sunt creati si utilizarea lor fiind impartita in domenii diferite au facut ca aceasta experienta sa fie mai interesanta.

Termenii cheie: Random, Generator, Metoda, Algoritm, Relation

Linear Congruential Generator

Basic Introduction

A linear congruential generator represents an algorithm that produces a sequence of pseudo-randomized numbers determined with a discontinuous piecewise linear equation. The method is well known for being one of the oldest and most spreaded pseudorandom number generators algorithm. The logic behind it is very easy to be understood as well as easy and fast to be implemented, especially if the computer hardware can provide modular arithmetic by storage-bit truncation.

The generator uses as its main core the recurrence relation:

$$X_{n+1} = (a * X_n + c) \% m$$

Where X represents the sequence of pseudorandom values, and

m, $0 < m$ – the modulus
a, $0 < a < m$ – the multiplier
c, $0 <= c < m$ – the increment
X_0, $0 <= X_0 < m$ – the seed or start value

Are integer constants that describe the generator. For example, if c=0, the generator would be called a multiplicative congruential generator(MCG), or Lehmer RNG. On the other hard, if c != 0, the method is called a mixed congruential generator.

In terms of math, when c != 0 the recurrence relation would represent rather an affine transformation than a linear one, but computer science has established better the inaccurate name.

Properties

A benefical thing about LCGs is that depending on the choice of its parameters' values, the period is known and also more important long. Although it is not the only criterion that affects a pseudorandom number generator being flawed, a period being too short can be clearly fatal.

Generally, LCGs can produce pseudorandom numbers that can pass the formal tests for randomness, but the quality of the output it generates is very responsive to the choice of parameters m and a. Let's take as an example, a = 1 and c = 1 that produce a simple modulo-m counter, which has a long period, but clearly isn't as random as we would want it to be.

Mainly, there exists the three common groups of parameter choice for LCGs :

1. **M prime, c = 0**

This is also known as the Lehmer RNG construction. The period is m-1 only if the multiplier a is picked as a primitive element of the integers modulo m. In the initial state the first value must be chosen between 1 and m-1.

One disavantage of a prime modulus is the fact that the modular lowering needs a double-width product and an explicit reduction

step. Usually a prime that is less than a power of 2 is prefered(popular choices are $2^{31}-1$ and $2^{61}-1$), so that we have the formula for the reduction modulo $m = 2^e - d$ calculated in such a way as $(ax \bmod 2^e) + d \lfloor ax/2^e \rfloor$. Although, this needs a follow-up with a conditional substraction of m if the result is too big, but the number of substractions is limited by the rule ad/m, which can be easily transformed into one if d is small.

As a fact we have the situation in which if a double-width product is unreasonable, and the multiplier is chosen precisely, Schrage's method may be used. To do this, use the factor m = q * a + r, i.e. q =[m/a] and r = m mod a. Then compute a * x mod m = a * (x mod q) - r * [x / q]. Since we have x mod q < q <= m/a, the first term is strictly less than a *m / a =m. If a is chosen in order to let r <= q, then the second term is also less than m: r * [x / q] <= r * x / q = x * (r / q) <= x < m. Thus, both products can be calculated in the end with a single-width product and the difference between them lies in the sequence [1-m, m-1], so it can be transformed to [0, m-1] with a single condition add.

Another disadvantage is that is uncommon to convert the value 1 <= x < m to uniform random bits. If a prime that is less than a power of 2 have been used, the missing values should be simply ignored.

2. M a power of 2, c = 0

M being a power of 2, usually $m = 2^{32}$ or $m = 2^{64}$, results in the creation of an efficient LCG, because this allows the modulus operation to be computed using the simple shortening of the binary representation. Actually, the most significant bits aren't computed at all. However, disadvantages are still bound to appear.

This form has the maximum period of m / 4, achieved if a ≡ 3 or a ≡ 5. The initial state X_0 must be an odd number, and the lowest three bits

of X be alternated between two states and they are not useful. It can be shown that this form is as good as a generator with a modulus a quarter the size and c != 0. A more important problem that comes with the usage of a power-of-two modulus is the fact that the low bits have a shorter period than the high bits. The lowest-order bit of X never changes(X is always odd), and the next two bits oscilate between two states. The rule follows like this: if $a \equiv 5 \pmod 8$, then bit 1 never changes and bit 2 alternates. If $a \equiv 3 \pmod 8$, then bit 2 never changes and bit 1 alternates. Bit 3 repeats with a period of 4, bit 4 has a period of 8, and so on. Only the most significant bit attains the full period.

3. c != 0

If c != 0, chosen parameters can influence a period equal to m to happen no matter the seed values. Although this will occur only if these conditions are respected:

- m and c are prime,
- a - 1 is divisible by all prime factors of m,
- a - 1 is divisible by 4 if m is divisible by 4

This form may work with any m, but it is bound to succeed only if m has many repeated prime factors such as powers of 2. If m was a square-free integer, this would allow $a \equiv 1$, which makes a very bad PRNG; a selection of full-period multipliers is available when m has repeated prime factors.

Even tho, this theorem produces a maximum period, it is not enough to guarantee a good generator. As an example, it is desirable for a – 1 to not be any more divisible by prime factors of m than necessary. Thus, if m is a power of 2, then a – 1 should be divisible by 4 but not divisible by 8 to respect the rule, i.e. a ≡ 5.

Note that a power-of-2 modulus has the same problem as described above for c = 0: the low k bits form a generator with modulus 2^k and

thus repeat with a period of 2^k; only the most significant bit achiving the full period.

The generator doesn't care about the choice of c, as long as it is relatively prime to the modulus(the simplest example would be m a power of 2 resulting in c being odd), so the value c = 1 is commonly chosen.

The series created by other picks of c can be illustrated as a simple function of series when c = 1. Getting deeper into it, if Y is the test series defined by $Y_0 = 0$ and $Y_{n+1} = a * Y_n + 1 \bmod m$, then a general series $X_{n+1} = a * X_n + c \bmod m$ can be written as an affine function of Y:

$X_n = (X_0 * (a - 1) + c) * Y_n + X_0 = (X_1 - X_0) * Y_n + X_0 \pmod m$.
A more usual illustration would be that any two series X and Z with the same multiplier and modulus are related according to the relation:

$(X_n - X_0) / (X_1 - X_0) = Y_n = (a^n - 1) / (a - 1) = (Z_n - Z_0) / (Z_1 - Z_0)$

PseudoCode

```
Begin
  Declare class mRND
  Create a function Seed(number)
       Assign a variable _seed=number
      Create a constructor mRND
 Declare _seed(0), a(0), c(0), m(2147483648)
 Create a function rnd()
    Return
      _seed = (a * _seed + c) mod m
```

Declare a, c, m, _seed

Done

Declare an another subclass MS_RND inheriting from base class mRND

Create a constructor

Read the variables a, c

Create a function rnd()

return mRND::rnd() right shift 16

Done

Declare an another subclass SS_RND inheriting from base class mRND

Create a constructor

Read the variables a, c

Create a function rnd()

return mRND::rnd()

Done

For x=0 to 6

Print MS_RAND

For x=0 to 6

Print SS_RAND

Done

End

The Good And Bad Parts of using LCGs

LCGs are fast and use minimal memory in order to keep their composure. This makes them worth of simulating multiple independent streams.

Mainly, there are some specific disavantages of using LCGs, the most problematic one would be the fact that the state is too small. People who used them for so many years can prove the fact that the technique is so good in spite of the issues. A LCG with big enough state can even pass stringent statistical tests.

For a specific example, an ideal random number generator with 32 bits of output is most likely to begin reproducing earlier values after $\sqrt{m} \approx 2^{16}$ results. Any pseudo random number generator which has its output its full, untruncated state will not create duplicates untill the full period happens. For the fact, PRNGs should have a period longer than the number of outputs required. According to the speed of modern computers, a period of 2^{64} would be good for all but a lower one for the least demanding applications, and longer ones for the simulations.

A specific flaw to LCGs is that if the dimesional space is of n-dimension, the points will lie on $\sqrt[n]{n!} \cdot m$. This is happenning because of the connection between the successive values in the sequence Xn. If a(the multiplier) is chosen carelessly, the planes will be fewer and wider and most likely will conduct problems. The spectral test, which is a simple test of an LCG's quality, measures this spacing and can lead to a good multiplier being chosen.

Another flaw mainly found in LCGs is the short period of the low-order bits when is m is a power of 2. This can be avoided by using a modulus larger than the required output, and also using the most significant bits of the state.

Although, LCGs can be a good option in some situations. As an example, in an embedded system, the memory is severly capped. Another situation can be found in a video game console taking a small number of high-order bits of an LCG may well be enough. The low order bits go through very short cycles.

LCGs should be evaluated with a lot of attention for fitting in non-cryptographic applications where highly advanced randomness is critical. In Monte Carlo simulations, a LCG must use a modulus greater than the cube of the number of random samples which are needed. This means, a good 32-bit LCG can be suited for obtaining one thousand random numbers; a 64-bit LCG can be good for approximatively 2 milion random numbers. Taking this in consideration, LCGs are not the best for Monte Carlo simulations.

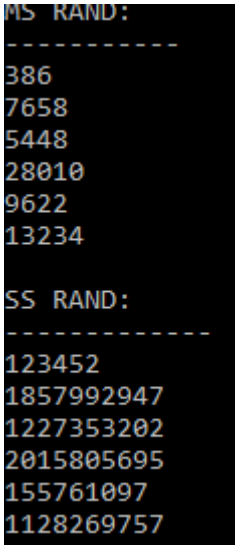
Output based on different inputs

At LCG we have 4 main inputs that correlate in changing the output:

The seed, a, c and m

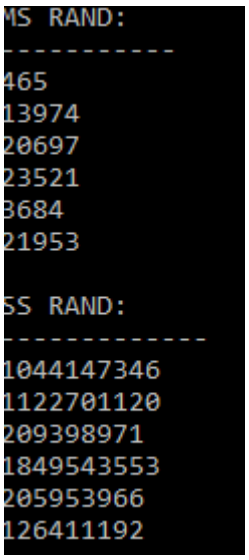
Seed = 0, a = 2140, c = 25310133, m =2147483649

a = 1016404594, c = 123452



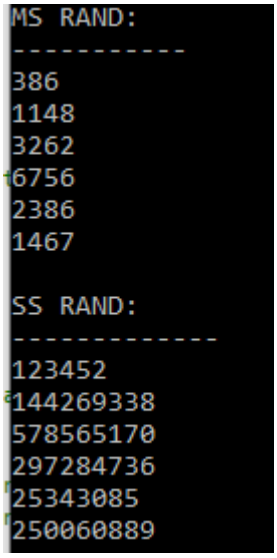
Seed = 2435, a = 2140, c = 25310133, m =2147483649

a = 1016404594, c = 123452



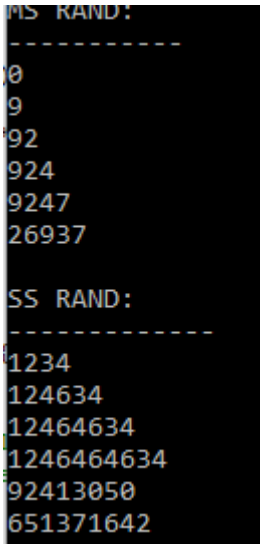
Seed = 0, a = 2140, c = 25310133, m = 643534543

a = 1016404594, c = 123452



Seed = 0, a = 10, c = 54543, m = 2147483649

a = 100, c = 1234



Lagged Fibonacci generator

A lagged Fibonacci generator is an implementation of a pseudorandom number generator. This type of random number generator has the goal to be an improvement to the classic 'linear congruential generator'. These generators have their base in the generalisation of the Fibonacci Sequence.

The reccurence relation that describes Fibonacci sequence mainly is:

$S_n = S_{n-1} + S_{n-2}$

So, as the item can be computed as the sum of the previous 2. The relation can be generalised like this:

$S_n \equiv S_{n-j} * S_{n-k} \pmod m$, $0 < j < k$

In this case the new term is some sort of combination between the previuos two. m is usually a power of 2(most used are 2^{32} or 2^{64}). The * operator doesn't denote multiplication here but a general binary operation. This operation may be addition, substraction, multiplication or the bit-wise exclusive-or operator(XOR). The main idea for this type of generators isn't that simple as it seems, choosing random values for j and k being not enough to guarantee good randomness. Also the initialisation may lead to instant errors if done wrong. Moreover, generators labbeled like this need k words of state(they remember the last k values).

If the operation used for creating the generator is addition, then the generator is called Additive Lagged Fibonacci Generator(ALFG), if multiplication is the operation utilized, it results in a Multiplicative Lagged Fibonacci Generator, and as an instance if XOR is used, the

generator is a Two-tap generalised feedback shift register(GFSR). As a fact, the Mersenne Twister algorithm is a variation on GFSR.

Properties

Lagged Fibonacci Generators that use addition or substraction as their operation have a maximum period of $(2^k - 1)*2^{M-1}$, while the exclusive-or operator has the period $(2^k - 1) \times k$. Another example would be the multiplication period that is $(2^k - 1)*2^{M-3}$.

A rule so that the generator has the maximum period possible is that the polynomial:

$y = x^k + x^j + 1$,

Must be primitive over the integers mod 2. Values for j and k have been discovered and published in different literatures. Some of the popular pairs are:

{j = 7, k = 10}, {j = 5, k = 17}, {j = 24, k = 55}, {j = 65, k = 71},

{j = 128, k = 159}, {j = 6, k = 31}, {j = 31, k = 63}, {j = 97, k = 127},

{j = 353, k = 521}, {j = 168, k = 521}, {j = 334, k = 607},

{j = 273, k = 607}, {j = 418, k = 1279}.

A remark would be that smaller numbers have shorter periods(meaning that only a few random numbers are therefore created after the first ‘random’ number is repeated so the sequence is resetted).

A rule regarding the choice of addition is that at least one of the first k values chosen to initialise the generator must be odd. If multiplication is chosen, it is required that all first k values to be odd.

Another suggestion made is that good ratios between j and k are close to the golden ratio.

Pseudocode

```
Begin
    Declare j = 3
    Declare k = 7
    Declare m = 10


    Declare val = 8675309
    Create function conv(val)
        Declare array arr[]
        For i=0 to len(val)
                arr.append(int(val[i]))
        Return arr
    Done
    Create function showvals(val,j,k)
        For i=0 to len(val)
                if i = j-1 then
                        print val[i]
                else if i = k-1 then
                        print val[i]
                else if then
                        print val[i]
  Done
```

```
Declare s = conv(val)
Print j, k
Print “seed:” s
If len(s) < k then
        Print “value needs to be larger than 7”
        Exit()
Showvals(s,j,k)
For  n=0 to 20
        out = (s[j-1] + s[k-1]) % m
        For i=0 to len(s)-1
                s[i]=s[i+1]
                s[len(s)-1] = out
                Print out
        Showvals(s,j,k)
         Print out
End
```

The Good and Bad parts of using LFGs

In a paper about four-tap shift registers, Robert M. Ziff, reffering to LFGs that use XOR operator, says that "It is now widely known that such generators, in particular with the two-tap rules such as R(103, 250), have serious deficiencies. Marsaglia observed very poor behavior with R(24, 55) and smaller generators, and advised against using generators of this type altogether. ... The basic problem of two-tap generators R(a, b) is that they have a built-in three-point

correlation between x_n, x_{n-a} and x_{n-b}, simply given by the generator itself ... While these correlations are spread over the size p=max(a,b,c,...) of the generator itself, they can evidently still lead to significant errors.". This statement is made based on classic LFGs where each new item depends on the previous two. A three-tap LFG has been shown to eliminate some statistical problems such as failing the Birthday Spacings and Generalized Triple tests.

As mentioned before, the initiliasation of LFGs presents a very big issue regarding the correctness in terms of quality for the generator. The output of LFGs is highly dependent on the condtions(inital terms), and statistical defects may appear instantly but also periodically. Another problem that is met in the implementation of LFGs is that the mathematical theory behind them is not yet refined, making it necessary to rely on statistical tests rather than theoretical performance.

Output based on different inputs

So the 4 main inputs in this implementation of the LFG are j, k, m and val.

J = 3 k = 7 m = 100 val = 86753445409

```
j= 3  k= 7
Seed:    86753445409
  8  6[  7]  5  3  4[  4]  5  4  0  9--> 11
  6  7[  5]  3  4  4[  5]  4  0  9 11--> 10
  7  5[  3]  4  4  5[  4]  0  9 11 10--> 7
  5  3[  4]  4  5  4[  0]  9 11 10  7--> 4
  3  4[  4]  5  4  0[  9] 11 10  7  4--> 13
  4  4[  5]  4  0  9[ 11] 10  7  4 13--> 16
  4  5[  4]  0  9 11[ 10]  7  4 13 16--> 14
  5  4[  0]  9 11 10[  7]  4 13 16 14--> 7
  4  0[  9] 11 10  7[  4] 13 16 14  7--> 13
  0  9[ 11] 10  7  4[ 13] 16 14  7 13--> 24
  9 11[ 10]  7  4 13[ 16] 14  7 13 24--> 26
 11 10[  7]  4 13 16[ 14]  7 13 24 26--> 21
 10  7[  4] 13 16 14[  7] 13 24 26 21--> 11
  7  4[ 13] 16 14  7[ 13] 24 26 21 11--> 26
  4 13[ 16] 14  7 13[ 24] 26 21 11 26--> 40
 13 16[ 14]  7 13 24[ 26] 21 11 26 40--> 40
 16 14[  7] 13 24 26[ 21] 11 26 40 40--> 28
 14  7[ 13] 24 26 21[ 11] 26 40 40 28--> 24
  7 13[ 24] 26 21 11[ 26] 40 40 28 24--> 50
 13 24[ 26] 21 11 26[ 40] 40 28 24 50--> 66
 24 26[ 21] 11 26 40[ 40] 28 24 50 66--> 66
```

What is displayed is the choice of 2 ‘random numbers from the seed value that are therefore used to create the final random number based on the addition method we used for the LFG.

As stated above there may appear issues if the inputs aren’t chosen correctly

J = 55 k =7 m = 100 val = 86753445409

```
j= 55  k= 7
Seed:    86753445409
  8  6  7  5  3  4[  4]  5  4  0  9Traceback (most recent call last):
  File "main.py", line 42, in <module>
    out = (s[j-1] + s[k-1]) % m     #the method used here is addition so it is ALFG
IndexError: list index out of range
```

As seen here j is too big.

Now we took the tests up to a larger scale

J = 65 k = 71 m = 255 val = 867534454093243243253264236426427427427548654874687547364354253252329898932324325335325136316347548568456945696494096409064

5 4 0 9 3 2 4 3 2 4 3 2 5 3 2 6 4 2 3 6 4 2 6 4 2 7 4 2 7 4

Linear-feedback shift register random number generator

In order to be able to talk about the generator itself we need to do some research in what the linear-feedback shift register is.

A linear-feedback shift regsiter is a shift register whose input bit is a linear function of its previous state.

As a matter of interest for our topic the most commonly used linear function is the exclusive-or. Thus, a LFSR is a shift register whose input bit is conducted by the XOR of some bits of the overall shift register value.

The initial value of the LFSR is called seed, and because the nature of the register operation is deterministic, the sequence of values produced by the register is determined by its current state. Obviously, the register by limited to having a finite number of states, it leads to a repeating cycle at some point. However, if the well-choosen feedback function is good, it can create a sequence of bits that appear to be very long and random.

The common usage of LFSRs is in generating pseudo-random numbers, pseudo-noise sequences, fast digital counters etc.

Properties

Types of LFSRs:

- Fibonacci LFSR – the bit positions that affect the next state are called the taps. The rightmost bit of the LFSR is called the output bit. The taps are XOR’d in pairs with the output bit and

then fed back into the leftmost bit. The bits produced in the rightmost positions are called the output stream.

A maximum-length for LFSR produces an m-sequence(it cycles through all $2^m - 1$ states within the shift register expect the state where all bits are zero), unless the sequence is only zeros, in which case it doesn't ever change.

XNOR can be used as an alternative in a LFSR. This function is an affine map, not strictly linear map, but the result is an equivalent polynomial counter whose state is the complement of the state of an LFSR. A state with all ones is unacceptable when using an XNOR feedback, based on the same rule as zeroes being impossible to use when using XOR. The state is unefficient to use as the counter would remain locked-up in this state.

The LFSR is maximal-length if the corresponding feedback polynomial is primitive. Meaning that the following conditions are necessary(but not enough): 1. number of taps even 2. the set of taps is setwise co-prime(there must be no common divisor other than 1 between the taps).

- Galois LFSR – it is known also as the modular, internal XORs, or one-to-many LFSR, the Galois LFSR is an alternate structure that can create the same output sequence as a normal LFSR. In the Galois model, when the system is clocked, bits that are not taps are shifted one position to the right unchanged. The taps are XORed with the output bit before they are stored in the next position. The new output bit is the next input bit. The outcome of this is that when the output bit is zero, all the bits in the register shift to the right unchanged, and the input bit becomes zero. On the other hand, when the output bit is one, the bits in the tap positions all flip(if they are 0, they become 1, and if

they are 1, they become 0), and then the entire register is shifted to the right and the input bit becomes 1.

Galois LFSRs do not link together every tap to create the new input(the XORing is done within the LFSR, no XOR gates are run in serial, therefore propagation times are lowered to just one XOR instead of a whole chain),letting each tap to be calculated in parallel, and therefore increase the speed of execution.

When taking about software implementation of an LFSR, the Galois form is more advantageous, as the XOR operations can be constructed a word at a time; only the output bit must be examined individually.

- Xorshift LFSR – As shown in the past, linear feedback shift registers can be constructed using XOR and Shift operations. This approach has a good amount of speed software-wise because these operations map efficiently into modern processor instructions.

Pseudocode

Begin

Declare POLYNOM_1 = 0x1 POLYNOM_2 = 0xAB

Declare lfsr32, lfsr31

Create function shift_lfsr(lfsr, polynomial)

Declare feedback

feedback = *lfsr & 1

*lfsr >>= 1

```
if feedback = 1 then
*lfsr ^= polynomial
return lfsr
Done
Create function init_lfsrs()
lfsr_1 = 0x1EA
lfsr_2 = 0xAEE
Done
Create function get_random()
shift_lfsr(&lfsr_1, POLYNOM_1)
return(shift_lfsr(&lfsr_1, POLYNOM_1) ^
shift_lfsr(&lfsr_2, POLYNOM_2))
Done
    Declare random_value[30]
    Init_lfsrs()
    For i =0 to 30
    random_value[i] = get_random()
    print random_value[i]
    Done
End
```

Good and Bad parts of using LFSR

The repetition of the sequence of states of an LFSR lets it to be used as a clock divider or as a counter when a non-binary sequence is

possible to be used. LFSR counters have simpler feedback logic than natural binary counters or Gray-code counters, and therefore their clock rate is higher. Also, a problem that may occur and can be prevented is that LSFR enters an all-zeros state, by setting it as a state similar to any other state in the sequence.

LFSRs have been used for a long time as pseudo-random number generators for the work usage in stream ciphers, because of the ease of creation from simple electromechanical or electronic circuits, long periods, and very uniformly distributed output streams. Also, a remark would be that a LFSR is a linear system, leading to fairly easy cryptanalysis.

Three methods are introduced in order to reduce the problem of LFSR-based stream ciphers:

1. Non-linear combination of several bits from the LFSR state
2. Non-linear combination of the output bits of two or more LFSRs
3. Irregular clocking of the LFSR

In scramblers, the data bit sequence is connected with the output of a linear-feedback register before modulation and transimission. The scrambling is removed at the received after demodulation. When the LFSR runs at the same speed as the transmitted symbol stream, this way of things to be is called scrambling. When it runs faster, it is called chipping code. The chipping code is connected with the data using exclusive OR before transmitting using binary phase-shift keying. The resulting signal has a bigger bandwith than the data, so it is called spread-spectrum communication.

Now going back to LFSR in generating pseudo-random numbers, a prime thing to say is that a basic LFSR does not create very good random numbers. A larger LFSR can improve this and also using the lower bits for the random number. Let’s take this example, you have a 10-bit LFSR and want an 8-bit number, you can take in this case the bottom 8 bits of the register for your number. Using this method in

this case you will get 8-bit number 4 times and 0, three times, before the LFSR finsihes one period and repeats. This solves the problem stated above of getting zeros, but it is still not good enough for achieving most statistical properties. Also, a subset of LFSR can be used to increase the number of permutations and therefore improve the randomness of the LFSR output.

Shifting the LFSR more than once before getting a random number can also improve the statistical properties.

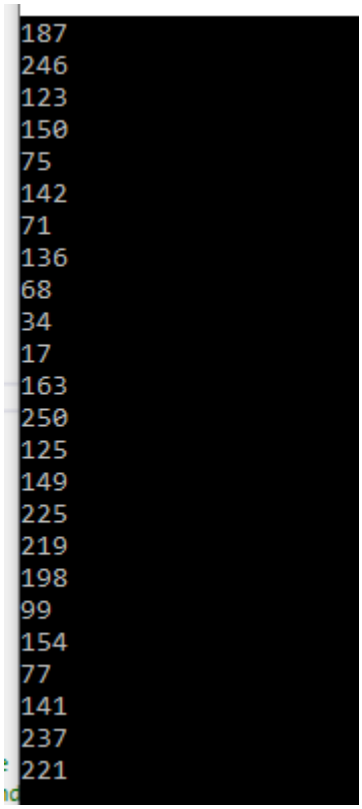
The issue of short periods in LFSRs can be solved by XORing the values of two or more different sized LFSRs together. The new period of XORed LFSRS will be the least common multiple of the periods.

The randomness of LFSRs can be increased as well by XORing a bit of measuring with the feedback term. However, you need to be cautious when doing this as there is a chance that the LFSR will go to all zeros with the addition of the bit. The zeroing will be remediated by itself if the measuring bit will be added periodically.

Output based on different inputs

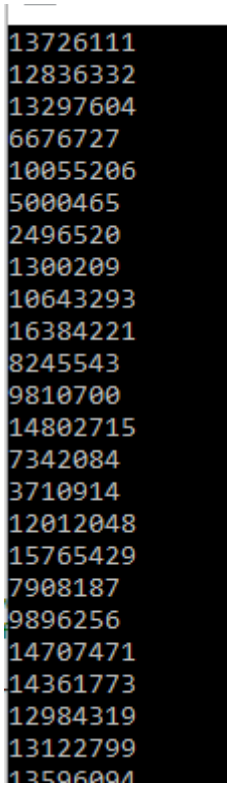
The relevant inputs for the output are: POLYNOM_1 , POLYNOM_2, lfsr32 and lfsr31.

POLYNOM_1 = 0x1, POLYNOM_2 = 0xAB, lfsr_1 = 0x1EA, lfsr_2 = 0xAEE



Now we will modify only the polynoms

POLYNOM_1 = 0xEEEE POLYNOM_2 = 0xABCDEF, lfsr_1 = 0x1EA, lfsr_2 = 0xAEE



Now the lfsr seed values

POLYNOM_1 = 0x1 POLYNOM_2 = 0xAB, lfsr_1 = 0xEA4634C, lfsr_2 = 0xAAABBB554543D