Journal of Computer Sciences

Original Research Paper

Parametrical Tuning of Twisting Generators

[1]Aleksei F. Deon and [2]Yulian A. Menyaev

[1]Department of Computer Science, Department of Mathematical Sciences,
N.E. Bauman Moscow State Technical University, Moscow, Russia
[2]Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Sciences, Little Rock, AR, USA

Article history
Received: 05-04-2016
Revised: 04-10-2016
Accepted: 05-10-2016

Corresponding Author:
Yulian A. Menyaev
Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Sciences, Little Rock, AR, USA
Email: yamenyaev@uams.edu

Abstract: Generators of uniformly distributed random numbers are broadly applied in simulations of stochastic processes that rely on normal and other distributions. In a point of fact, the uniform random numbers are actively used for applications that range from, modeling different phenomena such as theoretical mathematics and technical designing, to evidence-based medicine. This paper proposes a novel approach which consists of a combination of global twister with circular technique and initial congruential generation with complete stochastic sequences. It has been experimentally confirmed that for complete sequences this type of generation provides uniformity in distribution of random numbers. The offered program codes include the tuning methods for the generation technique where random numbers may take any bit length. Moreover, the automatic switching of generator parameters such as initial congruential constants depending on intervals for generated numbers is considered as well. Demonstrated results of testing confirm the uniformity of distribution without any repeated or skipped generated elements.

Keywords: Pseudorandom Number Generator, Stochastic Sequences, Congruential Number, Twister Generator

1. Introduction

1.1. Related Works

A Pseudorandom Number Generators (PRNG) are well-known techniques with broad applications in such areas as cryptography (Tusnoo *et al*., 2003; Ozturk *et al*., 2004; Panneton *et al*., 2006), simulation of stochastic processes (Entacher, 1998), comprehensive testing of technical systems (Leeb and Wegenkittl, 1997; Park and Miller, 1998), medical (Menyaev and Zharov, 2006a; 2006b; Menyaev and Zharova, 2006; Menyaev *et al*., 2006; 2013; 2016; Sarimollaoglu *et al*., 2014; Cai *et al*., 2016a; 2016b) and biological research (Wiese *et al*., 2005; Leonard and Jackson, 2015; Juratly *et al*., 2015; 2016) and others (Rababbah 2004; 2007; Politano *et al*., 2014; 2016; Riguzzi, 2016). In these publications, the concept of uniform random numbers in PRNG actively uses the operations of bit logic. Great success has been found in directions such as linear congruential generators (Niederreiter, 1995; Entacher, 1999) and in twisting algorithm generators where Mersenne numbers are usually used (Matsumoto and Kurita, 1992; 1994; Matsumoto and Nishimura, 1998; Nishimura, 2000). Important results were received in the use of approaches such as Fibonacci numbers (Makino, 1994; Aluru, 1997), Blum-Blum-Shub algorithm (Blum *et al*., 1986) and others. However, the issue of the repeatability of elements within a given time period and question of completeness of sets with elements are important and remain to be discussed.

1.2. Congruential Generators

According to the concept of congruence or similarity, the next following random number x_{i+1} is created based on current one x_i in accordance with:

$$x_{i+1} = f\left(x_i\right) mod\ m \tag{1}$$

where, *mod m* defines the interval $\left[\overline{0, m-1}\right]$ of generated numbers.

Historically the function $f(x_i)$ was chosen as linear algebraic transformation:

$$f\left(x_i\right) = ax_i + c \tag{2}$$

The constant coefficients a and c are selected in accordance with the properties of required Linear

Congruential Generator (LCG). Let's point out on example of a generating technique for several random numbers $N = 8$ within small interval $\left[\overline{0,7}\right]$. Choosing a short length interval doesn't break the concept of the current study, but it presents the results more obviously and in simple visual form. Let's take the starting point as $x_0 \in \left[\overline{1,7}\right] \subset \left[\overline{0,7}\right]$. Also, let's take into consideration all stochastic sequences with $a \in A = \left[\overline{1,7}\right]$ and $c \in C = \left[\overline{0,7}\right]$. So, the total volume of possible combinations among *a*, *c* and *x* looks as:

$$NN = card(A) * card(C) * (N-1) = 7 * 8 * 7 = 392 \quad (3)$$

Now let's add to the program code the function *Repeating()* which returns the meaning *true* if any element *x* from 0 to 7 has the repeating in the stochastic sequence having 8 numbers. This approach helps to define the congruential uniformity of completeness, i.e., sort out all the possible values what could be pointed here as:

$$CC = r / NN \quad (4)$$

where, *r* is the number of uniform stochastic sequence and *NN* is the total amount of sequences.

Below is the program code where the programming media C# (Schildt, 2010) is used from Microsoft Visual Studio 2013; although the same principles may be used for the classic programming language *C* (dialect Win32), or for C++ (dialect CLR). It doesn't matter, the result is similar:

```
static void Main ( string[] args )
{ int N = 8;                                  // sequence length
  Console.WriteLine ( "N = {0}", N );
  int NN = 0;                    // total amount of sequences
  int m = N;                        // modulus of congruence
  int r = 0;                     // uniform sequence number
  int[] x = new int[N];                 // stochastic sequence
  for ( int a = 1; a < N; a++ )
    for ( int c = 0; c < N; c++ )
      for ( int x0 = 1; x0 < N; x0++ )
      {  x[0] = x0;                // beginning of the sequence
        for ( int i = 1; i < N; i++ )
          x[i] = (a * x[i - 1] + c) % m;
        NN++;                    // total amount of sequences
        if ( Repeating(x, N) ) continue;          // repeating
        Console.WriteLine( "a = {0} c = {1}", a, c);
        r++;                                  // sequence number
        Console.Write( "r = {0,4}  ", r );
        qWrite( "x = ", x, N, true );
      }
  Console.WriteLine( "Finish" );
  Console.WriteLine( "NN = {0}", NN );
  double CC = (double)r / NN;        // cong-completeness
  Console.WriteLine( "CC = r / NN = {0:F4}", CC );
  Console.ReadKey();                          // result viewing
}
//---------------------------------------------------------------------
static bool Repeating( int[] x,  int N )
{ for ( int i = 1; i < N; i++ )
    for ( int j = 0; j < i; j++ )
      if ( x[i] == x[j] ) return true;                // repeating
  return false;                       // no repeating was found
}
//---------------------------------------------------------------------
static void qWrite( string text, uint[] x, uint N,
                    bool newstr )
{ Console.Write(text);
  for ( int i = 0; i < N; i++ )
    Console.Write( "{0,3}", x[i] );
  if (newstr) Console.WriteLine();
}
```

After executing this code the following listing appears. It is presented with abridging, for what dash line is used:

```
N = 8
a = 1  c = 1
r =   1  x =   1  2  3  4  5  6  7  0
r =   2  x =   2  3  4  5  6  7  0  1
- - - - -
a = 5  c = 1
r = 29  x =   1  6  7  4  5  2  3  0
r = 30  x =   2  3  0  1  6  7  4  5
- - - - -
a = 5  c = 7
r = 55   x =   6  5  0  7  2  1  4  3
a = 5  c = 7
r = 56   x =   7  2  1  4  3  6  5  0
Finish
NN = 392
CC = r / NN = 0.1429
```

This listing shows that only 56 uniform sequences are found from the total of 392 congruential ones in all ranges of possible combinations among *a*, *c* and *x.* In other words, we have no complete set of uniform sequences, only *CC* = 56/392 = 0.1429 from the total quantity of congruential list and that isn't much.

1.3. Modulus in Uniform Sequence

When generating congruential stochastic sequences the following technique is admitted: the operation *mod* in formula (1) above may be replaced by operation of bit conjunction. This is possible if generated binary number *x* having length of *w* bit belongs to interval $x \in \left[\overline{0, 2^w - 1}\right]$. As an example, below is the next program code which

uses congruential formula $5x+1$ to generate uniform sequences consisting of $2^w = 8$ numbers, where $w = 3$ is the bit length. The initial value $x_0 = 2$ has been chosen by chance. In this program code two arrays are created independently in accordance with the equations:

$$x_{i+1} = (ax_i + 1) mod\ 2^w = (5 * x_i + 1)\%\ 8 \quad (5)$$

$$q_{i+1} = (aq_i + 1)\ \&\ 111_2 = (5 * q_i + 1)\ \&\ 0 \times 7 \quad (6)$$

The function *qWrite()* shown previously is used to print out the result of task:

```
static void Main(string[] args)
{  int w = 3;                              // number bit length
   int N = 8;                              // sequence length
   Console.WriteLine( "w = {0}  N = {1}", w, N );
   int m = N;                          // modulus of congruence
   int[] x = new int[N];                // congruential numbers
   int[] q = new int[N];                // congruential numbers
   int a = 5, c = 1;                   // congruential constants
   x[0] = 2;         // beginning of congruential sequence x
   q[0] = 2;         // beginning of congruential sequence q
   for ( int i = 1; i < N; i++ )
   {  x[i] = ( a * x[i - 1] + c ) % m;
      q[i] = ( a * q[i - 1] + c ) & 0x7;
   }
   qWrite( "x = ", x, N, true );
   qWrite( "q = ", q, N, true );
   Console.ReadKey();                          // result viewing
}
```

After executing this code the following listing appears:

```
w = 3  N = 8
x =  2   3   0   1   6   7   4   5
q =  2   3   0   1   6   7   4   5
   010 011 000 001 110 111 100 101
```

In this example the interval length for 8 numbers is $x \in \left[\overline{0,7}\right]$, which appears in the binary scale as $x \in \left[\overline{000_2,111_2}\right]$ where each number x has the length of $w = 3$ bits. The last string has been added to emphasize the completeness of bit filling for the numbers in interval $\left[\overline{0,2^w-1}\right] = \left[\overline{0,7}\right] = \left[\overline{000_2,111_2}\right]$. As it's possible to see, the sequence x after applying the programming operation of modulus (%) is equal to sequence q after applying the programming operation of conjunction (&). However, the benefit is in the fact that conjunction (&) operation is running significantly faster than modulus (%) one.

1.4. Twisting Generators

As a base the technology of twisting generator uses a bit shifting of binary numbers in the stochastic sequence. Partially this approach was used in the classic research articles published by Japanese researches (Matsumoto and Nishimura, 1998; Nishimura, 2000). They have built several generators, including the well-known MT19937 (or MT19937-64 for the implementation that uses a 64-bit word length), which can reach a big value of repeatability as 2^{19937}-1 and that is excellent for some special cases.

The essence of circular shifting or global twister is in the following. If two numbers x_i and x_{i+1} having the same bit length w are taken, the next new values are derived according to the rule: the bit values taken from number x_{i+1} are successively moved to the left into number x_i; at the same time, the disengaged bits taken from the left of number x_i are joined circularly one-by-one to the right of number x_{i+1}. So, let's pay attention to those sequences which consist of several numbers having equal bit length. Next, we apply twisting algorithm to generate such sequences.

As an example, let's obtain the twisting shift in binary form for $w = 3$ and for those randomly taken numbers $x_2 = 5_{10} = 101_2$, $x_1 = 3_{10} = 011_2$ and $x_0 = 6_{10} = 110_2$. The structure of this approach may be presented as displayed in Fig. 1, where the first two strings are considered. The twister 0 is the initial sequence of congruential generation, while twister 1 is a result of global shifting to the left with a step of 1 bit. In turn, the elder bit of initial sequence realizes circular movement to the last position on the right in next sequence. Following this, the initial sequence 101 011 110_2 = 5 3 6_{10} is twisted into a twisting sequence 010 111 101_2 = 2 7 5_{10}. This algorithm is named as a twisting technique or a circular twister or a global circular twister due to it uses binary shift of all the numbers with no bit loss.

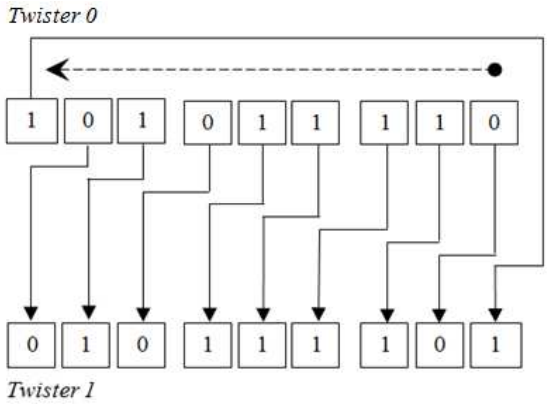

Fig. 1. Circular twister diagram

Program code shows the left twister together with one additional circular bit shifting for verification. As it's possible to see the last generated sequence is equal to the initial one that demonstrates the loop. The function *BitX()* brings out the numbers in binary view, which makes it easier to observe the twisting shift to the left by 1 bit. The function *Twist()* provides the twister algorithm itself, where elder bit of sequence is moved to the end, i.e., to the last position on the right. The function *qWrite()* is taken from previous program:

```
static void Main ( string[] args )
{ int w = 3;                                  // number bit length
  int N = 3;                                  // sequence length
  int maskW = 0x7;                            // number mask
  int maskU = 0x4;                            // elder bit mask
  Console.WriteLine( "w = {0}  N = {1}", w, N );
  int[] x = new int[] {5, 3, 6};              // 101 011 110
  int nwN = w* N + 1;  //amount of twisting iterations+1
  for ( int k = 1; k <= nwN; k++ )
  { Console.Write( "k = {0,2} |  ", k );
    BitX( x, w, N, maskW, false );  //bit view of numbers
    qWrite( " | ", x, N, true );              // decimal view
    Twist( x, w, N, maskW, maskU );           // global twister
  }
  Console.ReadKey();                          // result viewing
}
//---------------------------------------------------------------------
static void BitX ( int[] x, int w, int N, int maskW,
                   bool newstr)
{ for ( int i = 0; i < N; i++ )
  { int b = 1 << ( w – 1 ));                  // number beginning
    for  ( int j = 0; j < w; j++ )
    { Console.Write( (x[i] & b ) == 0 ? '0' : '1' );
      b >>= 1;                                // for the next bit
    }
    Console.Write( " " );
  }
  if ( newstr ) Console.WriteLine();
}
//---------------------------------------------------------------------
static void Twist ( int[] x, int w, int N,
                    int maskW, int maskU )
{ int z = ( x[0] & maskU ) >> ( w – 1 );      // left bit
  for ( int j = 0; j < N - 1; j++ )
  { int g = (x[j + 1] & maskU ) >> ( w – 1 );
    x[j] = ( (x[j] << 1 ) & maskW) | g;
  }
  x[N - 1] = ( ( x[N - 1] << 1 ) & maskW ) | z;   // loop
}
```

After executing this program code the following listing appears. The first string presents the initial sequence of bits which consists of the first three binary numbers. For visual convenience, at the last part of each string the decimal equivalents of binary numbers are pointed out. The next eight strings, from the 2nd to the 9th, are the result of step-by-step shifting to the left with a step size of 1 bit. The 10th string is the same as the 1st one what finishes a circular rotation:

```
w = 3  N = 3
k =  1 |  101  011  110  | 5 3 6
k =  2 |  010  111  101  | 2 7 5
k =  3 |  101  111  010  | 5 7 2
k =  4 |  011  110  101  | 3 6 5
k =  5 |  111  101  010  | 7 5 2
k =  6 |  111  010  101  | 7 2 5
k =  7 |  110  101  011  | 6 5 3
k =  8 |  101  010  111  | 5 2 7
k =  9 |  010  101  111  | 2 5 7
k =10 |  101  011  110  | 5 3 6
```

Now we may note here that number 5 is repeated 9 times, number 3 - 3 times, 6 - 3 times and 7 - 6 times. This result looks far from the ideal case in which all the generated numbers must be distributed uniformly.

1.5. Other Generators

In the theory of computational methods some other principles to generate the uniform random numbers are considered (Lewis and Payne, 1973; Chandrasekaran and Amira, 2008; Pellicer-Lostao and Lopez-Ruiz, 2008; Zhou *et al*., 2009; Bos *et al*., 2011). Let's have a short look at two general directions. In the first case, most techniques are exploring the complicated algebraic formulas, or multistep mathematical transformations, which is a time-consuming process and so it's discussible to be included in routine practice even for modern fast computers. In the second case, the simplest artificial mathematical solutions, like so-called Neumann's middle square method (Rahimov *et al*., 2011), have some severe weaknesses such as short loop and then the output sequence after a while maybe converted to zero. So, while simple and extremely fast to implement, their output is of poor quality (Park and Miller, 1998).

In general, both directions have no hundred-percent completeness of sets of non-repeated elements and moreover, they are worse than twisting random generators for the same tasks. So, in the following next sections of current work we propose additional solutions for twisting shift operations to improve the level of completeness in generation of uniform sequences.

2. Fundamentals

In the technology of global twister one of the positive results is the appearance of new values of random numbers which aren't presented in the initial sequence. As in section 1.4, the last example resulted in two new

numbers 010 and 111 after circular shifting of three initial numbers 101, 011 and 110. At the same time, the negative result is that in all received sequences the numbers are repeated non-uniformly. The situation may be improved if additional rule shown here in the following is applied.

Let's consider the issue of random number formation with a finite length of *w* bits in each number. From the informatics theory (Knuth, 1997) it is known that bit length defines the interval of numbers $x \in \left[\overline{0, 2^w - 1}\right]$. So, the sequence can be named completed if all the numbers in it are presented in interval $\left[\overline{0, 2^w - 1}\right]$. Thus, complete sequence consists of numbers having *w* bit length each and all the numbers in it may be found just once. Let's take this as a rule for the total of $N = 2^w$ numbers in initial sequence.

Now let's find out the answer to the question: would the unicity of global twister be saved if initially the uniform sequence is taken? The next program code helps in finding the answer. As an example let's take the numbers 5, 3, 6, 1, 7, 0, 4, 2 by chance; each of them has the length of 3 bits in binary scale. All eight numbers together organize a complete uniform sequence with a length of $N = 2^w = 2^3 = 8$ numbers. The use of functions *BitX()*, *qWrite()* and *Twist()* is the same as it has been shown in section 1.4:

```
static void Main( string[] args )
{ int w = 3;                              // number bit length
  int N = 8;                               // sequence length
  int maskW = 0x7;                          // number mask
  int maskU = 0x4;                         // elder bit mask
  Console.WriteLine( "w = {0}  N = {1}", w, N );
  int[] x = new int[] { 5, 3, 6, 1, 7, 0, 4, 2 };
  int nwN = w * N + 1; //amount of twisting iterations+1
  for ( int k = 1; k <= nwN; k++)
  { Console.Write( "k = {0,2} |  ", k );
    BitX( x, w, N, maskW, false );              // bit view
    qWrite( " | ", x, N, true );              // decimal view
    Twist( x, w, N, maskW, maskU );    // global twister
   }
  Console.ReadKey();                        // result viewing
}
```

After this code executing, a listing below appears:

```
w = 3  N = 8
k=  1 | 101 011 110 001 111 000 100 010 | 5 3 6 1 7 0 4 2
k=  2 | 010 111 100 011 110 001 000 101 | 2 7 4 3 6 1 0 5
k=  3 | 101 111 000 111 100 010 001 010 | 2 7 0 7 4 2 1 2
k=  4 | 011 110 001 111 000 100 010 101 | 3 6 1 7 0 4 2 5
k=  5 | 111 100 011 110 001 000 101 010 | 7 4 3 6 1 0 5 2
k=  6 | 111 000 111 100 010 001 010 101 | 7 0 7 4 2 1 2 5
k=  7 | 110 001 111 000 100 010 101 001 | 6 1 7 0 4 2 5 1
k=  8 | 100 011 110 001 000 101 010 111 | 4 3 6 1 0 5 2 7
k=  9 | 000 111 100 010 001 010 101 111 | 0 7 4 2 1 2 5 7
k=10 | 001 111 000 100 010 101 011 110 | 1 7 0 4 2 5 3 6
k=11 | 011 110 001 000 101 010 111 100 | 3 6 1 0 5 2 7 4
k=12 | 111 100 010 001 010 101 111 000 | 7 4 2 1 2 5 7 0
k=13 | 111 000 100 010 101 011 110 001 | 7 0 4 2 5 3 6 1
k=14 | 110 001 000 101 010 111 100 010 | 6 1 0 5 2 7 4 2
k=15 | 100 010 001 010 101 111 000 111 | 4 2 1 2 5 7 0 7
k=16 | 000 100 010 101 011 110 001 111 | 0 4 2 5 3 6 1 7
k=17 | 001 000 101 010 111 100 011 110 | 1 0 5 2 7 4 3 6
k=18 | 010 001 010 101 111 000 111 100 | 2 1 2 5 7 0 7 4
k=19 | 100 010 101 011 110 001 111 000 | 4 2 5 3 6 1 7 0
k=20 | 000 101 010 111 100 011 110 001 | 0 5 2 7 4 3 6 1
k=21 | 001 010 101 111 000 111 100 010 | 1 2 5 7 0 7 4 2
k=22 | 010 101 011 110 001 111 000 100 | 2 5 3 6 1 7 0 4
k=23 | 101 010 111 100 011 110 001 000 | 5 2 7 4 3 6 1 0
k=24 | 010 101 111 000 111 100 010 001 | 2 5 7 0 7 4 2 1
k=25 | 101 011 110 001 111 000 100 010 | 5 3 6 1 7 0 4 2
```

In this listing there are 24 non-repeatable sequences and last, the 25th one, confirms the circular properties of twister. However, only 14 of them are satisfied to uniformity, i.e., all of the numbers can be encountered once. The other 10 sequences have repeatable numbers and thus can't be named as uniform. So, this example confirms the fact that even if the random complete sequence is taken initially, there is no guarantee that global twister creates the complete set of all the unique sequences. At the same time, even 14 uniform sequences from the total of 24 of them might be considered as a good result.

To obtain a twister having no any repeats we apply the congruential generator $x_i+1 = (ax_{i+1}+c)\&\ w$ for the complete set of numbers. The next program code works with congruential and twisting techniques of generation, thus positive properties from both of them are combined. In section 1.2, the function *Repeating()* provides the complete uniform sequences. Each initial congruential sequence is created by function *Cong_Start()* for what the values $a = 5$ and $c = 1$ are chosen by chance. The functions *Twist()* and *qWrite()* are taken from section 1.4:

```
static void Main ( string[] args )
{ int w = 3;                              // number bit length
  int N = 8;                               // sequence length
  int maskW = 0x7;                          // number mask
  int maskU = 0x4;                         // elder bit mask
  Console.WriteLine ( "w = {0}  N = {1}", w, N );
  int[] x = new int[N];                  // stochastic sequence
  int k = 0;                        // complete sequence number
  int a = 5, c = 1;                  // congruential constants
  int x0 = 1;                     // beginning of the sequence
  Cong_Start ( x, N, a, c, x0, maskW );
  Console.WriteLine ( "a = {0}  c = {1}", a, c );
  k++;                                    // sequence number
```

```
  Console.Write ( "k = {0,4}  ", k );
  qWrite ( "x = ", x, N, true );
  for ( uint i = 1; i < w * N; i++ )
  { Twist (x, w, N, maskW, maskU );              // twister
    if ( Repeating ( x, N ) ) continue;
     k++;
    Console.Write ( "k = {0,4}  ", k );
    qWrite ("x = ", x, N, true );
  }
  Console.ReadKey();                          // result viewing
}
//---------------------------------------------------------------------
static void Cong_Start ( int[] x, int N, int a, int c,
                         int x0, int maskW )
{ x[0] = x0;
  for ( int i = 1; i < N; i++ )
    x[i] = ( a * x[i - 1] + c ) & maskW;
}
```

After this code executing, a listing below appears:

```
w = 3  N = 8
a = 5  c = 1
k =   1   x =  1 6 7 4 5 2 3 0
k =   2   x =  3 5 7 1 2 4 6 0
k =   3   x =  7 3 6 2 5 1 4 4
k =   4   x =  6 7 4 5 2 3 0 1
k =   5   x =  5 7 1 2 4 6 0 3
k =   6   x =  3 6 2 5 1 4 0 7
k =   7   x =  7 4 5 2 3 0 1 6
k =   8   x =  7 1 2 4 6 0 3 5
k =   9   x =  6 2 5 1 4 0 7 3
k =  10   x =  4 5 2 3 0 1 6 7
k =  11   x =  1 2 4 6 0 3 5 7
k =  12   x =  2 5 1 4 0 7 3 6
k =  13   x =  5 2 3 0 1 6 7 4
k =  14   x =  2 4 6 0 3 5 7 1
k =  15   x =  5 1 4 0 7 3 6 2
k =  16   x =  2 3 0 1 6 7 4 5
k =  17   x =  4 6 0 3 5 7 1 2
k =  18   x =  1 4 0 7 3 6 2 5
k =  19   x =  3 0 1 6 7 4 5 2
k =  20   x =  6 0 3 5 7 1 2 4
k =  21   x =  4 0 7 3 6 2 5 1
k =  22   x =  0 1 6 7 4 5 2 3
k =  23   x =  0 3 5 7 1 2 4 6
k =  24   x =  0 7 3 6 2 5 1 4
```

In section 1.2 we considered an example when initial value is taken from interval $x_0 \in \left[\overline{1, N-1}\right] = \left[\overline{1, 2^w - 1}\right]$. On the other side, the twister allows reaching the set of $w \cdot N = w \cdot 2^w$ sequences. Now let's find out the answer to the next question: for how many unique non-repeatable sequences might be achieved if congruential and twisting techniques are combined? It may be assumed that the total number of unique congruential sequences, where each of them has been generated by twister, is defined as $(w \cdot 2^w) \cdot N$. However, this assumption is wrong and the next testing trial clarifies that.

With help of function *MatrixAdd()* the program code locates generated congruential and twisting sequences in supportive matrix *MS*. In each string there are one sequence and three additional elements. The function *MatrixCheck()* compares the sequences in all strings. The first additional element is to plot the result of unicity and two others are to locate the congruential constants *a* и *c*:

```
static void Main ( string[] args )
{ int w = 3;                                  // number bit length
  int N = 8;                                   // sequence length
  int maskW = 0x7;                             // number mask
  int maskU = 0x4;                             // elder bit mask
  Console.WriteLine ( "w = {0}  N = {1}", w, N );
  int[] x = new int[N];                  // stochastic sequence
  int[,] MS = new int[2000, N + 3];                  // matrix
  int M = 0;                  // amount of sequences in matrix
  int k = 0;                        // complete sequence number
  int a = 5, c = 1;                     // congruential constants
  for ( int x0 = 1; x0 < N; x0++ )
  { Cong_Start ( x, N, a, c, x0, maskW );
    if ( Repeating ( x, N ) ) continue;              // repeating
    MatrixAdd ( MS, ref M, x, N, a, c );
    Console.WriteLine ( "a = {0}  c = {1}", a, c );
    k++;                                          // sequence number
    Console.Write ( "k = {0,4}  ", k );
    qWrite ( "x = ", x, N, true );
    for ( int i = 1; i < 24; i++ )
    { Twist ( x, w, N, maskW, maskU );           // twister
      if ( Repeating ( x, N ) ) continue;
      MatrixAdd ( MS, ref M, x, N, a, c );
      k++;
      Console.Write ( "k = {0,4}  ", k );
      qWrite ( "x = ", x, N, true );
    }
  }
  MatrixCheck ( MS, M, N, 1 );  // coincidence checking
  Console.WriteLine ( "Matrix of unique sequences" );
  MatrixWrite ( MS, M, N );                // matrix monitor
  Console.ReadKey();                        // result viewing
}
//---------------------------------------------------------------------
static void MatrixAdd ( int[,] MS, ref int M,
                        int[] x, int N, int a, int c)
{ for ( int j = 0; j < N; j++ ) MS[M, j] = x[j];
  MS[M, N] = 0;                        // coincidence number
  MS[M, N + 1] = a;                                // constant a
  MS[M, N + 2] = c;                                // constant c
  M++;                        // amount of sequences in matrix
}
```

```
//--------------------------------------------------------------------
static void MatrixCheck ( int[,] MS, int M,
                          int N, int bMS )
{ for ( int i = bMS; i < M; i++ )
  for ( int k = 0; k < i; k++ )
  { int j = 0;
    for ( ; j < N; j++ )
      if ( MS[i, j] != MS[k, j] ) break;
    if ( j == N )                    // coincidence is found
    { MS[i, N] = k + 1;               // coincidence number
      break;
    }
  }
}
//--------------------------------------------------------------------
static void MatrixWrite ( int[,] MS, int M, int N )
{ for ( uint i = 0; i < M; i++ )
  { Console.Write ( "i = {0,5} |", i + 1 );
    for ( int j = 0; j < N; j++ )
      Console.Write ( "{0,3}", MS[i, j] );
      Console.Write ( " | " );
      Console.Write ( "{0,3}", MS[i, N] );
      Console.Write ( "{0,5}", MS[i, N + 1] );
      Console.WriteLine ( "{0,5}", MS[i, N + 2] );
  }
}
```

The listing below is the result of execution. Dash lines show abridgments. So, the total number of all the sequences generated by congruential and twisting techniques is 7·24 = 168 and each of them consists of 8 different numbers. All those sequences are located in matrix *MS* subsequently. The 2nd part of listing presents the result of uniformity checking. If the string includes 0 (at the first position in the right part of string with additional elements) it means that sequence is unique. So, the total quantity is $w \cdot N = 3 \cdot 8 = 24$ and those sequences are on the first 24 strings of the 2nd part of listing. The sequences on strings from 25 to 168 aren't unique because they repeat previous 24 sequences in different combinations:

```
w = 3 N = 8
a = 5 c = 1
k =   1   x =  1 6 7 4 5 2 3 0
k =   2   x =  3 5 7 1 2 4 6 0
- - - - -
k =   6   x =  3 6 2 5 1 4 0 7
- - - - -
k =  16   x =  2 3 0 1 6 7 4 5
- - - - -
k = 167   x =  5 7 1 2 4 6 0 3
k = 168   x =  3 6 2 5 1 4 0 7
```

Matrix of unique sequences:

```
k =   1 |  1 6 7 4 5 2 3 0 |  0 5 1
k =   2 |  3 5 7 1 2 4 6 0 |  0 5 1
- - - - -
k =  16 |  2 3 0 1 6 7 4 5 |  0 5 1
- - - - -
k =  24 |  0 7 3 6 2 5 1 4 |  0 5 1
k =  25 |  2 3 0 1 6 7 4 5 | 16 5 1
- - - - -
k = 168 |  3 6 2 5 1 4 0 7 |  6 5 1
```

Another interesting fact to note is that congruential sequence on string 25 is the same as twisting sequence on string 16 and last the 168th twisting sequence is the same as the 6th one.

So, the last received result is that congruential technique while generating the random sequences having any numbers could be the part of twisting technique. The simulation with different bit lengths of *w* confirms this statement.

3. Parameters

Showed in previous sections the peculiarities of congruential and twisting generators allow us to ask the next question regarding design of concrete generator for complete uniform stochastic sequences. One of the basic characteristics of generator tuning is a bit length *w* of produced numbers. For real technical systems the lengths of 16, 24, 32 and 64 bits are demanded. The next required parameter is the amount of numbers in each sequence. For uniform generating the abundant value $N = 2^w$ might be taken because this allows convenient application of the twisting technique for each sequence.

Now it's time to talk about formulas for generation the numbers in sequences. Usually initial values are derived by using congruential technique in linear mode as $x_{i+1} = (ax_i+c) mod\ m$. For the complete sequences the operation *mod m* may be changed to conjunction with a bit length *w*. This is equal, so can be presented as:

$$x_{i+1} = \left(ax_i + c\right) \& w \tag{7}$$

The initial number x_0, i.e., seed, for beginning of generation may be pointed directly and manually, or by using computer timer, i.e., automatically. For complete sequences the interest is focusing on numbers which belong to interval $\left[\overline{0, N-1}\right] = \left[\overline{0, 2^w - 1}\right]$. In section 2 it has been shown experimentally that following two directions of generation are possible:

- Create all the initial numbers for initiation of generation in current sequence from interval $x_0 \in \left[\overline{1, 2^w - 1}\right]$ and number 0 hasn't been using as starting point in sequence

- Create the initial stochastic sequence by using congruential technique and other stochastic sequences are generated by using global twister which allows getting the sequences with a beginning number 0

In congruential linear formula (7) the parameter a is chosen under conditions to reach complete sequences for all the numbers having w bit length. It is known that uniform sequences having $N = 2^w$ numbers are created when $(a\text{-}1)mod\ 4 = 0$. For example, if $w = 3$, the sequence is completed and consist of $N = 2^w = 2^3 = 8$ numbers. Thus, the parameter a has to be chosen as $a = 1$ or 5, due to possible values belonging to interval $a \in \left[\overline{1, N-1}\right] = \left[\overline{1,7}\right]$. At the same time, if $w = 4$ the parameter a may take the values 1, 5, 9, 13 from interval $a \in \left[\overline{1,2^4-1}\right] = \left[\overline{1,15}\right]$. By using the program code described in section 1.2 the simulation technique may verify this.

The parameter c in formula (7) above is chosen under the same condition to reach complete generated sequence. To get this, the parameter c has to be taken as odd numbers from interval $c \in \left[\overline{1,2^w-1}\right]$. Simulation technique described in section 1.2 may also verify this.

As an example, the program code below confirms the recommendations in choosing of parameters a and c, where bit length of generated numbers is taken as $w = 4$ and matrix verification of unicity for each sequence is done. The functions *MatrixAdd()*, *MatrixCheck()* and *MatrixWrite()* are the same as described in section 2:

```
static void Main ( string[] args )
{ int w = 4;                                  // number bit length
  int N = 16;                                 // sequence length
  Console.WriteLine ( "w = {0}   N = {1}", w, N );
  int maskW = (int)( 0xFFFFFFFF >> (32 - w); // mask
  int maskU = 1 << ( w – 1 );                 // elder bit mask
 Console.WriteLine("maskW = {0:X} maskU = {1:X}",
                    maskW, maskU);
  int[] x = new int[N];                       // stochastic sequence
  int[,] MS = new int[3000, N + 3];           // matrix
  int M = 0;                                  // amount of sequences in matrix
  for ( int a =1; a < N; a += 4 )
    for ( int c = 1; c < N; c += 2 )
    { int x0 = 1;
      Cong_Start(x, N, a, c, x0, maskW);
      if ( Repeating ( x, N ) ) continue;     // repeating
      MatrixAdd ( MS, ref M, x, N, a, c );
      for ( int i = 1; i < w * N; i++ )
      {  Twist ( x, w, N, maskW, maskU );     // twister
        if (Repeating ( x, N ) ) continue;    // repeating
        MatrixAdd ( MS, ref M, x, N, a, c );
      }
    }
  MatrixCheck ( MS, M, N, 1 );  // coincidence checking
  Console.WriteLine ( "Matrix of unique sequences" );
  MatrixWrite ( MS, M, N );
  Console.WriteLine ( "Finish" );
  Console.ReadKey();                          // result viewing
}
```

The listing below is the result of execution. Dash lines show abridgments. At the end of each string the first additional element is the result of checking of unicity and two others show the congruential constants a и c. The quantity of packs of sequences is $w \cdot N = 4 \cdot 16 = 64$, which means total amount of unique sequences is $w \cdot N \cdot 4_a \cdot 8_c = 4 \cdot 16 \cdot 4 \cdot 8 = 2048$:

```
w = 4   N = 16
maskW = F   maskU = 8
Matrix of unique sequences
k =   1 |1  2  3  4  5  6  7  8  9 10 11 12 13  14 15 0 |0  1 1
k =   2 |2  4  6  8 10 12 15  1  3  5 7  9 11  13 14  0 |0  1 1
- - - - -
k =  64 |0  9  1 10  2 11  3 12  4 13 5 14  6  15  7  8 |0  1 1
k =  65 |1  4  7 10 13  0  3  6  9 12 15  2  5  8 11 14 |0  1 3
- - - - -
k= 133 |6 11  0  5 10 15  4  9 14  3  8 13 2  7 12  1 |0  1 5
- - - - -
k= 600 |7 15  4  8 1  9  6  10  3 11  0 12  5 13  2 14 |0  5 3
- - - - -
k=1025|1 10  11 4 5 14 15  8  9  2  3 12 13  6   7  0 |0  9 1
- - - - -
k=2048|0 14  5 15  2  8 7  9  4 10 1 11  6 12  3 13|0 13 15
Finish
```

This listing shows that function *Repeating()* hasn't found any sequences having repeated numbers from the total of 2048. The first additional element in each string points to this because it takes 0 for each generated sequence. In other words, this result means that definitely now we are capable to generate all the unique sequences by using this kind of parameter a. As it has been said above, this result is achieved due to odd values for constant c. Moreover, for any other possible values of a and odd values of c all the generated stochastic sequences are unique.

4. Construction and Results

From the special aspects which are considered in section 3, now let's consider the practical realization. Below is the program code for name space *nsDeonYuliTwist28DA*, in which the class *cDeonYuliTwist28DA* includes all the required parameters and functions of generating (the using of class *cDeonYuliTwist28DA* is presented in the following program P020401, after this program code):

```
namespace nsDeonYuliTwist28DA
{ class cDeonYuliTwist28DA
  { public int w = 16;                    // number bit length
    public int N = 0;                     // sequence length
    public int N1 = 0;                    // maximum number
    int[] x;                                        // sequence
    public int x0 = 1;                  // sequence beginning
    public double abf = 0.39;           // beginning part of a
    public double aef = 0.39;           // ending part of a
    public int a1b = 1, a1e = 0;        // internal of a1
    int a1s = 0;                        // a1 interval state
    public int a2b = 1, a2e = 0;        // interval of a2
    int a2s = 0;                        // a2 interval state
    int a1 = 5;                         // a1 interval constant
    int a2 = 5;                         // a2 interval constant
    int nA = 1;                   // a1 or a2 constant number
    public int a = 5;             // constant a current value
    public double cbf = 0.1;            // beginning part of  c
    public double cef = 0.9;            // ending part of c
    public int cb = 1, ce = 0;          // ending part of c
    public int c = 1;                // congruential constant
    public int st = 1;                    // generator state
    int nW = 0;                      // twister shift number
    int nT = 0;                   // nT twister number in N
    int nV = 0;                      // element number in x
    uint maskW = 0U;                        // number mask
    uint maskU = 0U;                        // elder bit mask
//---------------------------------------------------------------------
    public cDeonYuliTwist28DA()
    { N = 1 << w;                           // sequence length
      N1 = N - 1;                        // maximum number
      x0 = N1 / 7;                          // generator starts
    }
//---------------------------------------------------------------------
    public int Next()
    { bool flagW = true;                      // perpetual loop
      while (flagW)                           // status voyage
      {  switch (st)                          // status switch
        { case 1:                  // initialization of generator
          nA = 1;                  // generation starts inside a1
          a1s = 1;                 // create twister 0 inside a1
          a1 = a1e;                                 // a1 ending
          a = a1;                          // current constant a
          a2s = 0;                          // a2 while not used
          a2 = a2b - 4;             // on the left to interval a2
          c = cb;                    // beginning of interval c
          st = 2;                    // twister 0 congruention
          break;
        case 2:                                 // initial twister 0
          DeonYuli_Cong(a,c);          // initialization of x
          nW = 0;                       // twister shift number
          nT = 0;              // twister nT number inside nW
          nV = 0;                       // initial value number
          st = 101;                          // array x is ready
          break;
        case 101:                                 // take from x
          if (nV <= N1) flagW = false;
          else st = 102;                 // change the twister
          break;
        case 102:         // next twister with the same a, c
          nW++;                    // next bit of shift in word
          if (nW < w) { st = 103; break; }
          nT++;     // twister beginning from next value
          nW = 0;                                 // without shift
          if (nT < N) st = 103;                // next twister
          else st = 201;                     // next constant c
          break;
        case 103:                                 // next twister
          DeonYuli_Twist();                    // next twister
          nV = 0;                      // initial value number
          st = 101;                             // take from x
          break;
        case 201:                    // change twister with c
          c += 2;                             // next constant c
          if (c <= ce) st = 2;                    // new twister
          else st = 202;                     // next constant a
          break;
        case 202:                           // change interval a
          c = cb;                           // initial value of c
          if (nA == 1) nA = 2; else nA = 1;
          if (nA == 1) st = 203;                 // interval a1
          else st = 204;                         // interval a2
          break;
        case 203:                    // new constant from a1
          a1 -= 4;
          a = a1;
          if (a1b <= a1) { a1s = 1; st = 2; }
          else { a1s = 2; st = 205; }             // a1 is over
          break;
        case 204:                    // new constant from a2
          a2 += 4;
          a = a2;
          if (a2 <= a2e) { a2s = 1; st = 2; }
          else { a2s = 2; st = 205; }             // a2 is over
          break;
        case 205:                           // a1 or a2 is over
          if (a2s != 2) st = 204;
          else if (a1s != 2) st = 203;
                else st = 1;                 // initial situation
          break;
        }                                                 // switch
      }                                                    // while
      return x[nV++];                       // random number
    }
//---------------------------------------------------------------------
    void DeonYuli_Cong ( int a, int c )
    { x[0] = x0;                          // sequence beginning
      for ( int i = 1; i < N; i++ )
        x[i] = (int) ( a * x[i - 1] + c ) & maskW );
    }
//---------------------------------------------------------------------
    void DeonYuli_Twist ()
```

```
    { uint z = (uint)( x[0] & maskU ) >> ( w – 1 );   // left
      for ( int j = 0; j < N - 1; j++ )
      { uint g = (uint)( x[j + 1] & maskU ) >> ( w – 1 );
        x[j] = (int)(( (x[j] << 1 ) & maskW ) | g);
      }
      x[N - 1] =
          (int)(( ( x[N - 1] << 1 ) & maskW ) | z);   // loop
    }
//---------------------------------------------------------------------
    public void Start()
    { N = 1 << w;                          // sequence length
      N1 = N - 1;                          // maximum number
      maskW = 0xFFFFFFFF >> (32 - w);      // mask
      maskU = (uint)(0x1 << (w - 1));      // elder bit mask
      DeonYuli_SetA();                     // set a1 and a2 borders
      DeonYuli_SetC();                     // set c border
      if (x0 > N1) x0 = N1;
      else if (x0 == 0) x0 = 1;
      x = new int[N];                      // sequence
      st = 1;                              // initialization of generator
    }
//---------------------------------------------------------------------
    public void TimeStart ()
    { x0 = (int)DateTime.Now.Millisecond;
      Start();                             // generator starts
    }
//---------------------------------------------------------------------
    public void SetW( int sw )
    { w = Math.Abs( sw );    // random number bit length
      if ( w < 3 ) w = 3;                  // minimum length
      else if ( w > 28 ) w = 28;           // maximum length
    }
//---------------------------------------------------------------------
    public void SetA( double sab, double sae )
    { abf = Math.Abs( sab );
      aef = Math.Abs( sae );
      if ( abf > 1.0 ) abf = 1.0;
      if ( aef > 1.0 ) aef = 1.0;
      if ( abf > aef ) aef = abf;
    }
//---------------------------------------------------------------------
    void DeonYuli_SetA ()
    { a1b = (int)( N1 * abf );         // bottom border for a1
      a1b = DeonYuli_PlusA ( a1b );    // beginning of a1
      a2e = (int)( N1 * aef );         // top border for a2
      a2e = DeonYuli_MinusA ( a2e );   // ending of a2
      int r = a2e - a1b;
      if ( a1b >= a2e )                // interval a like a point
      { a1e = a1b;                     // a1 is one point
        a2b = a1b;                     // a2 like a1
        a2e = a2b;                     // a2 is one point
        return;
      }
      if ( r == 4 )                    // one-point a1 and a2
      { a1e = a1b;                     // a1 is one point
        a2b = a2e;                     // a2 is one point
        return;
      }
      if ( r == 8 )          // a1 has 2 points, a2 – one point
      { a1e = a1b + 4;                 // ending of a1
        a2b = a2e;                     // beginning of a2
        return;
      }
      a1e = ( a1b + a2e ) / 2;         // middle of a
      a2b = a1e;
      a1e = DeonYuli_MinusA( a1e );    // to the left
      a2b = a1e + 4;                   // to the right of middle
    }
//---------------------------------------------------------------------
    int DeonYuli_PlusA ( int a )
    { if ( a < 1 ) { a = 1; return a; }
      int z = a;                       // bottom border for a
      for ( int i = 0; i < 3; i++ )
        if ( a % 4 != 0 ) a--;         // uniform condition
        else break;
      a++;                             // real value of constant a
      if ( a < z ) a += 4;     // to the right of bottom border
      if ( a >= N1 ) a -= 4;   // to the left of top border
      return a;
    }
//---------------------------------------------------------------------
    int DeonYuli_MinusA ( int a )
    { if (a < 1) { a = 1; return a; }
      int z = a;                       // bottom border of a
      for (int i = 0; i < 3; i++)
      if (a % 4 != 0) a--;             // uniform condition
      else break;
      a++;                             // real value of constant a
      if (a > z) a -= 4;               // to the left of top border
      return a;
    }
//---------------------------------------------------------------------
    public void SetC ( double scb, double sce )
    { cbf = Math.Abs( scb );
      cef = Math.Abs( sce );
      if ( cbf > 1.0 ) cbf = 1.0;
      if ( cef > 1.0 ) cef = 1.0;
      if ( cbf > cef ) cef = cbf;
    }
//---------------------------------------------------------------------
    void DeonYuli_SetC ()
    { cb = (int)( N1 * cbf );          // bottom border of c
      if ( cb % 2 == 0 ) cb += 1;      // only odd c
      if ( cb > N1 ) cb = N1;          // maximal value
      ce = (int)( N1 * cef );          // top border of c
      if ( ce % 2 == 0 ) ce -= 1;      // only odd c
      if ( ce > N1 ) ce = N1;          // maximal value
      if ( cb > ce ) ce = cb;
      c = cb;        // beginning of congruential constant c
    }
//---------------------------------------------------------------------
    public void SetX0 ( double xs )
```

```
    { x0 = (int)( N1 * Math.Abs( xs ) );
    }
//=====================================
  }
}
```

In class *cDeonYuliTwist28DA* some variables are reserved. They may be tuned with help of encapsulated functions. As a first example let's use default arguments for the simple task to generate several random numbers having $w = 16$ bits from intervals $\left[\overline{0, 2^w - 1}\right] = \left[\overline{0, 2^{16} - 1}\right] = \left[\overline{0, 65535}\right]$. The program code for this is presented below. The names P020401 and cP020401 are chosen by chance:

```
using nsDeonYuliTwist28DA;          // twisting generator
namespace P020401
{ class cP020401
  { static void Main(string[] args)
    { cDeonYuliTwist28DA CT =
                    new cDeonYuliTwist28DA();
      CT.Start();                        // generator starts
      for (int j = 0; j < 8; j++)
      { int z = CT.Next();
        Console.Write("{0,7} ", z);          // monitor
      }
      Console.ReadKey();                 // result viewing
    }
  }
}
```

The listing below is the result of execution:

```
9362  36699  52924   2805  8774  14575  51504  13129
```

For the next generating of new sequence as twister 0, the constant *a* in class *DeonYuliTwist28DA* is chosen by turns from two different intervals as it is shown in Fig. 2.

The value *a*1*e* is to the left from *N*/2 and value *a*2*b* = *a*1*b*+4 is to the right from *N*/2. Moving of *a* in interval *a*1 is accomplished from the right to the left, i.e., from *a*1*e* to *a*1*b* with a step of -4; moving of *a* in interval *a*2 is accomplished from the left to the right, i.e., from *a*2*b* to *a*2*e* with a step of +4. This choice of defining of *a* has been made artificially to provide better confusion while applying the generation.

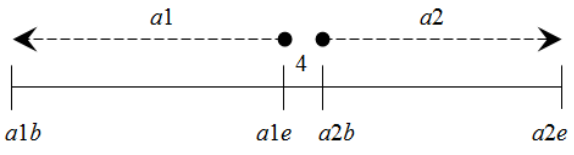

Fig. 2. Schematics of interval realization for constant *a*

Any other algorithms of confusion may be chosen if an additional task to control the process of generation is demanded. For example, if time sensor *TimeStart()* is applied, the initial value of the stochastic sequence in this case is defined by component having less than millisecond size of real time.

The value *a*1*b* is defined by parameter *b* and value *a*2*e* is defined by parameter *e* in the interface of their assigned function *SetA()*. In the case of complete generation of *a* the total numbers belong to interval $\left[\overline{1, N - 3}\right]$ with a step of ±4. The call of tuning function in this case looks as *SetA(0.0, 1.0)*.

The program code presented below realizes the ability of tuning such parameters as *w*, *x0*, *a1b*, *a2e*, which may possess different meanings. Parameter *c* takes all the odd numbers from interval $\left[\overline{1, N - 1}\right]$, which start from 1 and finish at *N*-1. As it has been shown in section 3, the first number may be defined directly while other numbers - in the line of *c*. This kind of defining is appropriate due to the twisting technique that is applied for complete sequences and it provides generation of all non-repeatable numbers in each next sequence. As an example, let's consider the program code containing the following values: $w = 4$, $N = 2^w = 16$, $a1b = 1$, $a2e = 13$, $x_0 = 1.0/5.0 * N = 3$; for the each value of $a = 5,9,1,13$, an assign parameter *c* takes the values $c = 1,3,5,7,9,11,13,15$. The names P020403 and cP020403 are chosen by chance:

```
using nsDeonYuliTwist28DA;          // twisting generator
namespace P020403
{ class cP020403
  { static void Main(string[] args)
    { cDeonYuliTwist28DA CT =
                    new cDeonYuliTwist28DA();
      CT.SetW(4);                    // number bit length
      CT.SetA(0.0, 1.0);                       // all of a
      CT.SetC(0.0, 1.0);                       // all of c
      CT.Start();                      //generator starts
      Console.WriteLine("w = {0}   N = {1}",
                         CT.w, CT.N);
      Console.WriteLine("a1b = {0}  a1e = {1}",
                         CT.a1b, CT.a1e);
      Console.WriteLine("a2b = {0}  a2e = {1}",
                         CT.a2b, CT.a2e);
      Console.WriteLine("cb = {0}  ce = {1}",
                         CT.cb, CT.ce);
      int k = 0;                           // sequence number
      int NN = 0;            // quantity of random numbers
      for (int nw = 0; nw < CT.w; nw++)
        for (int nt = 0; nt < CT.N; nt++)
          for (int na = 1; na <= 4; na++)
            for (int nc = 1; nc <= 8; nc++)
            { Console.Write("k={0,4} | ", ++k);
```

```
            for (int i = 0; i < CT.N; i++)
            { Console.Write("{0,3}", CT.Next());
              NN++;
            }
            Console.WriteLine(" a={0,2} c ={1,2}",
                              CT.a, CT.c);
            if (k % 250 == 0) Console.ReadKey();
          }
      Console.WriteLine("Finish");
      Console.WriteLine("NN = {0}", NN);
      Console.ReadKey();                    // result viewing
    }
  }
}
```

The listing below is the result of execution. Dash lines show abridgments:

```
w = 4   N = 16
a1b = 1   a1e = 5
a2b = 9   a2e = 13
cb = 1   ce = 15
k =  1 15 12 13 2 11  8  9 14  7 4 5 10 3 0 1  6
a=5 c=1
k =  2 15  9 10  5  7  1  3 12 14 8 11 4 6 0 2 13
a=5 c=1
- - - - -
k =  1000 6 10 1 13 4 8 7 11 2 14 5 9 0 12 3 15
a = 9   c = 15
- - - - -
k =  1230 9 6 5 2 0 15 12 11 8 7 4 3 1 14 13 10
a = 1   c = 7
- - - - -
k =  1900 8 5 13 6 10 7 15 0 12 1 9 2 14 3 11 4
a = 13   c = 11
- - - - -
k =  2048 11 6 12 3 13 0 14 5 15 2 8 7 9 4 10 1
a = 13 c = 15
Finish
NN = 32768
```

A total of $4 \cdot 16 \cdot 4 \cdot 8 = 2048$ sequences have been received. Each sequence consists of 16 non-repeatable random numbers which suggest that the total amount of generated elements is $2048 \cdot 16 = 32768$.

5. Discussion

In general, for the quantity N_s of generated sequences to have complete non-repeatable elements depends on the following reasons:

- Bit length w of random numbers
- Amount of numbers in one sequence $N=2^w$
- Amount of twisters $N_T = w \cdot N = w \cdot 2^w$ for each pair of congruential constants a and c
- Amount of possible variations $N_a = N/4$ of a and of possible variations $N_c = N/2$ of c

If all the mentioned parameters are combined in one equation it appears as the following:

$$\begin{aligned} N_s &= N_T * N_a * N_c \\ &= w \cdot 2^w . \frac{2^w}{4} \cdot \frac{2^w}{2} = w \frac{2^{3w}}{2^3} = w \cdot 2^{3w-3} = w \cdot 2^{3(w-1)} \end{aligned} \quad (8)$$

The quantity N_{ns} of generated numbers in all completed sequences is defined as:

$$N_{ns} = N \cdot N_s = 2^w \cdot w \cdot 2^{3w-3} = w \cdot 2^{4w-3} . \quad (9)$$

The bit length N_{bs} of non-repeating elements is defined as the quantity of bits in all the numbers of all non-repeatable sequences:

$$\begin{aligned} N_{bs} &= w \cdot N_{ns} = w \cdot N \cdot N_s = w \cdot 2^w \cdot w \cdot 2^{3(w-1)} = \\ &= w^2 \cdot 2^{4w-3} \end{aligned} \quad (10)$$

Deriving N_s for $w = 4$ leads to $N_s(w = 4) = 4 \cdot 2^{3 \cdot 3} = 4 \cdot 512 = 2048$ and this value is confirmed directly by using counter k which has been used at the end of the section 4.

In the practical application of RNG the interest is addressed to double-byte numbers having length of $w = 2^4 = 16$ bits. In this case, the amount of random numbers in one complete sequence is $N(w = 16) = 2^w = 2^{16} = 65536$. The total amount of non-repeatable sequences is defined as: $N_s (w = 16) = w \cdot 2^{3w-3} = 2^{3 \cdot 16+1} = 2^{49}$. Thus, the amount of all the generated numbers is $N_{ns}(w = 16) = N \cdot N_{ns} = 2^4 \cdot 2^{49} = 2^{51}$. So, the bit length in this case is defined as: $N_{bs}(w = 16) = w \cdot N_{ns} = 2^4 \cdot 2^{51} = 2^{55}$.

Let's consider this result in comparison with the generator MT19937 (Matsumoto and Nishimura, 1998). By following the program code for this generator it's possible to find out that bit length of generated random numbers is $w_{32} = 32$. The length of the array for initial congruential generation is defined as $MT_{32} = 624$ elements. A global twister doesn't use the circular movement of elder bit of sequence. The data supports that the total amount of generated random numbers is:

$$\begin{aligned} MT_{ns32} &= w_{32} \cdot MT_{32} - 31 \\ &= 32 * 624 - 31 = 19968 - 31 = 19937 \end{aligned} \quad (11)$$

Diminution of value 31 is because the twister doesn't take into account the circle of needless bits. Theoretically, it appears that in all 19937 numbers it is possible to reach a non-repeatable sequence with 2^{19937} bits. However, this is hypothetical case only due to congruential generator for twister 0 with following single-bit twisters may provide just only $MT_{ns} = 19937$ numbers. To reach more numbers it's required to define a new initial value $x0$, because congruential constants a and c are pointed as stationary values and thus they can't be changed automatically.

The same properties have been found for the generator *MT*19937-64, for which the bit length of generated random numbers is w_{64} = 64. To retain the twisting properties for the bit sequence of 19937, the array of initial generation for twister 0 is halved in distribution space and thus it consists of 312 elements. This means that the length of the array after substitution is the following: MT_{64} = 312 = 624/2. The calculation of the total amount of random numbers generated is:

$$\begin{aligned} MT_{ns32} &= w_{64} \cdot MT_{64} - 31 = 64 * 312 - 31 \\ &= 64 / 2 * (312 * 2) - 31 = 19968 - 31 = 19937 \end{aligned} \quad (12)$$

The general properties for both Mersenne twister generators are equal, however the 64-bit version has for the benefit of parallel speed-up calculations. This is important for the 128-bit registers of modern processors (Saito and Matsumoto, 1998).

Significantly to note, congruential initial generation can't provide the rearrangement of numbers in sequence due to the fact that congruence can never be equal to factorial; they have very different natures of mathematical phenomena. In principle the congruential initial generation can't realize the technique of creating the theoretically completed random numbers having uniform distribution. Let's additionally clarify this having used the discrete mathematics.

It is of great importance to pay attention to the combinatory properties of uniform sequences with no repeats. For this goal let's assume that uniform sequence consists of *N* numbers and each of them is found once. The question is how many times such uniform sequences may be reached? The answer gives the combinatorial analysis (Waerden, 1991a; 1991b; Johnsonbaugh, 2008): that is factorial *N*! due to uniform sequences admit any rearrangement of *N* numbers. Congruential and twister generation can't provide the same result as factorial; complete sets of stochastic sequences can realize this.

Let's continue the discussion regarding the twisting generator tuning proposed here. It's highly important to be satisfied that all the random numbers generated by *nsDeonYuliTwist28DA* are found an equal quantity of times. This is because the requirement of uniformity explains that all the elements in complete generation indeed must be presented equal quantity of times. In the simplest case, all the numbers are found once in single sequence. Therefore, in the set of rearrangements, any number can be presented as many times as the quantity of rearrangements is applied. This is because in each uniform rearrangement any number is found once. So, now we have a simple and well-organized tool to test the uniformity for generators.

Below is the program code where each element of array*xC* is a counter for random numbers, so the index of counter is equal to the random number. In this code the uniformity for twister 0 which corresponds to initial congruential generation is verified. The bit length of 16 bits for each random number is taken, thus whole length of sequence includes $N = 2^w = 2^{16}$ = 65536 of random numbers:

```
using nsDeonYuliTwist28DA;          // twisting generator
namespace P020501
{ class cP020501
  { static void Main(string[] args)
    { cDeonYuliTwist28DA CT =
                          new cDeonYuliTwist28DA();
      CT.Start();                          // generator starts
      Console.WriteLine("w = {0}  N = {1}",
                              CT.w, CT.N);
      int[] cX = new int[CT.N];          // array of counters
      for (int i = 0; i < CT.N; i++) cX[i] = 0;
      for (int n = 0; n < CT.N; n++)
      { int z = CT.Next();
        cX[z]++;                 // counter for random number
      }
      int count0 = 0; // amount of non-appeared elements
      int count1 = 0; // amount of single-valued elements
      int count2 = 0;// amount of double-valued elements
      for (int i = 0; i < CT.N; i++)
      { if (cX[i] == 1) count1++;                    // 1 time
        else if (cX[i] == 2) count2++;               // 2 times
             else if (cX[i] == 0) count0++;          // never
      }
      Console.Write("count0 = {0}   ", count0);
      Console.Write("count1 = {0}   ", count1);
      Console.WriteLine("count2 = {0}   ", count2);
      Console.ReadKey();                   // result viewing
    }
  }
}
```

The listing below is the result of execution:

```
w = 16   N = 65536
count0 = 0   count1 = 65536   count2 = 0
```

To be sure that single complete twister having given values for congruential constants *a* and *c* satisfies to uniformity, it's necessary to generate *nwN* = *w*·*N* = 16·65536 = 1048576 of random numbers. This task may be solved by the following code, where each uniformly distributed random value has to be found 16 times. The names P020502 and cP020502 are chosen by chance:

```
using nsDeonYuliTwist28DA;          // twisting generator
namespace P020502
{ class cP020502
  { static void Main(string[] args)
    { cDeonYuliTwist28DA CT =
                      new cDeonYuliTwist28DA();
      CT.Start();                          // generator starts
```

```
      int nwN = CT.w * CT.N;        // quantity of twisters
      Console.WriteLine(
                    "w = {0}  N = {1}  nwN = {2}",
                     CT.w, CT.N, nwN);
      int[] cX = new int[CT.N];          // array of counters
      for (int i = 0; i < CT.N; i++) cX[i] = 0;
      for (int n = 0; n < nwN; n++)
      { int z = CT.Next();
        cX[z]++;                    // random number counter
      }
      int count16 = 0;       // amount of 16-times elements
      for (int i = 0; i < CT.N; i++)
        if (cX[i] == 16) count16++;                // 16-times
      Console.WriteLine("count16 = {0}   ", count16);
      Console.ReadKey();                      // result viewing
    }
  }
}
```

Two strings appear after executing the last code:

```
w = 16   N = 65536  nwN = 1048576
count16 = 65536
```

The task of testing which is discussed above is completed. Thorough investigation of twisting generation requires special resources like a powerful processor, additional random-access memory, hard-drive storage, etc. Fortunately, the principles of verification are the same as have been described in detail in this study.

6. Conclusion

In the beginning we started with the fact that congruential technique of random number generation can't provide the uniform distribution for all congruential constants and initial values in the linear function $x_{i+1} = (ax_i+c) mod\ m$. Fortunately, the result may be improved to uniform distribution if complete sequences are considered in the assumption that total amount of random numbers is $N = m$. In this case the sequences could be organized as completed and uniformly distributed where each element is found once. To speed up the calculation capability for complete sequences the modulus operation may be changed to the operation of bit conjunction (&) with a mask having $w = \log_2 N$ of bit length. By using circular rotation for elder bit of current sequence while applying the left global twister, the value $w \cdot N$ of unique sequences has been reached. In this case each sequence consists of N non-repeatable random numbers having uniform distribution and presented just once. Herein, by using matrix verification it's confirmed experimentally that all the various combinations of pairs of congruential constants a и c can include any initial settings of seed x_0. All these fundamental properties have allowed us to consider the questions addressed to realize tuning of twisting generators, where the intervals for congruential constants are chosen. The maximal length of intervals for those constants, which are required for the complete sequences, provides the maximum possible quantity of generating twisting sequences and twisting random numbers. In general, the techniques presented in this work seem to be very promising for many applications and primarily for such areas as information technology, cryptography, engineering, biology, medicine and others.

Funding Information

The authors have no support or funding to report.

Author's Contributions

The authors equally contributed in this work.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Aluru, S., 1997. Lagged fibonacci random number generators for distributed memory parallel computers. J. Parallel Distr. Com., 45: 1-12. DOI: 10.1006/jpdc.1997.1363

Blum, L., M. Blum and M. Shub, 1986. A simple unpredictable pseudo-random number generator. SIAM J. Comput., 15: 364-383. DOI: 10.1137/0215025

Bos, J.W., T. Kleinjung, A.K. Lenstra and P.L. Montgomery, 2011. Efficient SIMD arithmetic modulo a mersenne number. Proceedings of the IEEE 20th Symposium on Computer Arithmetic, Jul. 25-27, IEEE Xplore Press, pp: 213-221. DOI: 10.1109/ARITH.2011.37

Cai, C., K.A. Carey, D.A. Nedosekin, Y.A. Menyaev and M. Sarimollaoglu *et al*., 2016a. *In vivo* photoacoustic flow cytometry for early malaria diagnosis. Cytometry A, 89A:531-542. DOI: 10.1002/cyto.a.22854

Cai, C., D.A. Nedosekin, Y.A. Menyaev, M. Sarimollaoglu and M.A. Proskurnin *et al*., 2016b. Photoacoustic flow cytometry for single sickle cell detection *in vitro* and *in vivo*. Anal. Cell. Pathol., 2642361: 1-11. DOI: 10.1155/2016/2642361

Chandrasekaran, S. and A. Amira, 2008. High performance FPGA implementation of the mersenne twister. Proceedings of the 4th IEEE International Symposium on Electronic Design, Test and Applications, Jan. 23-25, IEEE Xplore Press, pp: 482-485. DOI: 10.1109/DELTA.2008.113

Entacher, K., 1998. Bad subsequences of well-known linear congruential pseudorandom number generators. ACM Trans. Model. Comput. Simulat., 8: 61-70. DOI: 10.1145/272991.273009

Entacher, K., 1999. Parallel streams of linear random numbers in the spectral test. ACM Trans. Model. Comput. Simulat., 9: 31-44. DOI: 10.1145/301677.301682

Johnsonbaugh, R., 2008. Discrete Mathematics. 7th Edn., Pearson Prentice Hall, Upper Saddle River, ISBN-10: 0131354302, pp: 766.

Juratly, M.A., E.R. Siegel, D.A. Nedosekin, M. Sarimollaoglu and A. Jamshidi-Parsian *et al*., 2015. *In vivo* long-term monitoring of circulating tumor cells fluctuation during medical interventions. PLoS One, 10: e0137613-e0137613. DOI: 10.1371/journal.pone.0137613

Juratly, M.A., Y.A. Menyaev, M. Sarimollaoglu, E.R. Siegel and D.A. Nedosekin *et al*., 2016. Real-time label-free embolus detection using *in vivo* photoacoustic flow cytometry. PLoS One, 11: e0156269. DOI: 10.1371/journal.pone.0156269

Knuth, D.E., 1997. The Art of Computer Programming: Seminumerical algorithms. 3rd Edn., Addison-Wesley, Reading, ISBN-10: 0201896842, pp: 762.

Leeb, H. and S. Wegenkittl, 1997. Inversive and linear congruential pseudorandom number generators in empirical tests. ACM Trans. Model. Comput. Simulat., 7: 272-286. DOI: 10.1145/249204.249208

Leonard, P. and D. Jackson, 2015. Efficient evolution of high entropy RNGs using single node genetic programming. Proceedings of the Annual Conference on Genetic and Evolutionary Computation, Jul. 11-15, Madrid, Spain, pp: 1071-1078. DOI: 10.1145/2739480.2754820

Lewis, T.G. and W.H. Payne, 1973. Generalized feedback shift register pseudorandom number algorithm. J. ACM, 20: 456-486. DOI: 10.1145/321765.321777

Makino, J., 1994. Lagged-fibonacci random number generators on parallel computers. Parallel Comput., 20: 1357-1367. DOI: 10.1016/0167-8191(94)90042-6

Matsumoto, M. and T. Nishimura, 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simulat., 8: 3-30. DOI: 10.1145/272991.272995

Matsumoto, M. and Y. Kurita, 1992. Twisted GFSR generators. ACM Trans. Model. Comput. Simulat., 2: 179-194. DOI: 10.1145/146382.146383

Matsumoto, M. and Y. Kurita, 1994. Twisted GFSR generators II. ACM Trans. Model. Comput. Simulat., 4: 254-266. DOI: 10.1145/189443.189445

Menyaev, Y.A. and V.P. Zharov, 2006a. Experience in development of therapeutic photomatrix equipment. Biomed. Eng., 40: 57-63. DOI: 10.1007/s10527-006-0042-6

Menyaev, Y.A. and V.P. Zharov, 2006b. Experience in the use of therapeutic photomatrix equipment. Biomed. Eng., 40: 144-147. DOI: 10.1007/s10527-006-0064-0

Menyaev, Y.A. and I.Z. Zharova, 2006. A technique for surgical treatment of infected wounds based on photodynamic and ultrasound therapy. Biomed. Eng., 40: 284-290. DOI: 10.1007/s10527-006-0102-y

Menyaev, Y.A., V.P. Zharov, E.A. Mishanin, A.P. Kuzmich and S.E. Bessonov, 2006. Combined photovacuum therapy of copulative dysfunction. Proc. SPIE, 6078: 241-248. DOI: 10.1117/12.656713

Menyaev, Y.A., D.A. Nedosekin, M. Sarimollaoglu, M.A. Juratli and E.I. Galanzha *et al*., 2013. Optical clearing in photoacoustic flowcytometry. Biomed. Opt. Express, 4: 3030-3041. DOI: 10.1364/BOE.4.003030

Menyaev, Y.A., K.A. Carey, D.A. Nedosekin, M. Sarimollaoglu and E.I. Galanzha *et al*., 2016. Preclinical photoacoustic models: Application for ultrasensitive single cell malaria diagnosis in large vein and artery. Biomed. Opt. Express, 7: 3643-3658. DOI: 10.1364/BOE.7.003643

Niederreiter, H., 1995. Some linear and nonlinear methods for pseudorandom number generation. Proceedings of the 27th Conference on Winter Simulation, Dec. 03-06, Arlington, VA, USA, pp: 250-254. DOI: 10.1145/224401.224611

Nishimura, T., 2000. Tables of 64-bit mersenne twisters. ACM Trans. Model. Comput. Simulat., 10: 348-357. DOI: 10.1145/369534.369540

Ozturk, E., B. Sunar and E. Savas, 2004. Low-power elliptic curve cryptography using scaled modular arithmetic. Proceedings of the 6th International Workshop Cryptographic Hardware and Embedded Systems, Aug. 11-13, Cambridge, MA, USA, pp: 92-106. DOI: 10.1007/978-3-540-28632-5_7

Panneton, F., P. L'Ecuyer and M. Matsumoto, 2006. Improved long-period generators based on linear recurrences modulo 2. ACM Trans. Math. Software, 32: 1-16. DOI: 10.1145/1132973.1132974

Park, S.K. and K.W. Miller, 1998. Random number generators: Good ones are hard to find. Commun. ACM, 31: 1192-1201. DOI: 10.1145/63039.63042

Pellicer-Lostao, C. and R. Lopez-Ruiz, 2008. Pseudo-random bit generation based on 2D chaotic maps of logistic type and its applications in chaotic cryptography. Proceedings of the International Conference on Computational Science and its Applications, Jun. 30-Jul. 3, Perugia, Italy, pp: 784-796. DOI: 10.1007/978-3-540-69848-7_62

Politano, G., A. Benso, A. Savino and S. Di Carlo, 2014. ReNE: A Cytoscape Plugin for Regulatory Network Enhancement. PLoS ONE, 9: e115585. DOI: 10.1371/journal.pone.0115585

Politano, G., F. Orso, M. Raimo, A. Benso and A. Savino *et al*., 2016. CyTRANSFINDER: A Cytoscape 3.3 plugin for three-component (TF, gene, miRNA) signal transduction pathway construction. BMC Bioinformatics, 17: 157. DOI: 10.1186/s12859-016-0964-2

Rababbah, A., 2004. Jacobi-bernstein basis transformation. CMAM, 4: 206-2014. DOI: 10.2478/cmam-2004-0012

Rababbah, A., 2007. High Accuracy Hermite approximation for space curves in R^d. J. Math. Anal. Applied, 325: 920-931. DOI: 10.1016/j.jmaa.2006.02.054

Rahimov, H., M. Babaie and H. Hassanabadi, 2011. Improving middle square method RNG using chaotic map. Applied Math., 2: 482-486. DOI: 10.4236/am.2011.24062

Riguzzi, F., 2016. The distribution semantics for normal programs with function symbols. Int. J. Approx Reason, 77:1-19. DOI: 10.1016/j.ijar.2016.05.005

Saito, M. and M. Matsumoto, 2008. SIMD-Oriented Fast Mersenne Twister: A 128-bit Pseudorandom Number Generator. In: Monte Carlo and Quasi-Monte Carlo Methods 2006, Keller, A., S. Heinrich and H. Niederreiter, (Eds.), Springer Science and Business Media, Berlin, ISBN-10: 3540744967, pp: 607-622.

Sarimollaoglu, M., D.A. Nedosekin, Y.A. Menyaev, M.A. Juratly and V.P. Zharov, 2014. Nonlinear photoacoustic signal amplification from single targets in absorption background. Photoacoustics, 2: 1-11. DOI: 10.1016/j.pacs.2013.11.002

Schildt, H., 2010. C# 4.0: The Complete Reference. 1st Edn., Tata McGraw-Hill Education, New York. ISBN-10: 007070368X, pp: 949.

Tusnoo, Y., T. Saito, T. Suzaki, M. Shigeri and H. Miyauchi, 2003. Cryptanalysis of DES implemented on computers with cache. Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, Sept. 8-10, Cologne, Germany, pp: 62-76. DOI: 10.1007/978-3-540-45238-6_6

Waerden, B.L. van der, 1991. Algebra: Volume I. 1st Edn., Springer-Verlag, New York, ISBN-10: 978-0-387-40624-4, pp: 265.

Waerden, B.L. van der, 1991. Algebra: Volume II. 1st Edn., Springer-Verlag, New York, ISBN: 978-0-387-40625-1, pp: 284.

Wiese, K.C., A. Hendriks, A. Deschenes and B.B. Youssef, 2005. The impact of pseudorandom number quality on *P-RnaPredict*, a parallel genetic algorithm for RNA secondary structure prediction. Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation, Jun. 25-29, Washington, DC, USA, pp: 479-480. DOI: 10.1145/1068009.1068089

Zhou, Q., X. Liao, K.W. Wong, Y. Hua and D. Xiao, 2009. True random number generator based on mouse movement and chaotic hash function. Inform. Sci., 179: 3442-3450. DOI: 10.1016/j.ins.2009.06.005