Journal of Universal Computer Science, vol. 23, no. 4 (2017), 353-384
submitted: 23/11/16, accepted: 16/2/17, appeared: 28/4/17 © J.UCS

Twister Generator of Arbitrary Uniform Sequences

Aleksei F. Deon
(Department of Information Systems and Computer Science
N.E. Bauman Moscow State Technical University, Moscow, Russia
deonalex@mail.ru)

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

Abstract: Twisting generators for pseudorandom numbers may use a congruential array to simulate stochastic sequences. Typically, the computer program controls the quantity of elements in array to limit the random access memory. This technique may have limitations in situations where the stochastic sequences have an insufficient size for some application tasks, ranging from theoretical mathematics and technic constructions to biological and medical studies. This paper proposes a novel approach to generate complete stochastic sequences which don't need a congruential twisting array. The results of simulation confirm that received random numbers are distributed absolutely uniformly in the set of unique sequences. Moreover, combination of this novel approach with an algorithm of tuning for twisting generation affords the length extension of created sequences without requiring additional computer random access memory.

Key Words: Pseudorandom Number Generator, Stochastic Sequences, Congruential Numbers, Twister Generator
Categories: G.2.1, G.3, F.2

1 Introduction

This article continues previously presented approaches [Deon and Menyaev 2016], where the principle of twisting generation of complete uniformed sequences is discussed. Typical pseudorandom number generators (PRNG) are broadly used in cryptography [Shamir 1983; Lewko and Waters 2009; Claessen and Palka 2013], testing of technical systems [Eichenauer-Herrmann and Niederreiter 1994; Hellekalek 1995; Sussman et al. 2006; Mandal et al. 2016], analyzing of teletraffic [Li 2010, 2017], theoretical simulation of natural processes [Niederreiter 1992; Meka and Zuckerman 2010; Goplan et al. 2011], theoretical mathematics [Leva 1992; Applebaum, 2012; White et al. 2008; Langdon 2009], biological verification studies [Juratly et al. 2015, 2016; Cai et al., 2016; Sarimollaoglu et al., 2014], clinical medicine [Menyaev et al. 2013, 2016; Tong et al. 2014; Chapman et al. 2015; Carey et al. 2016] and development of medical equipment [Zharov et al. 2001; Menyaev and Zharov 2005, 2006; Menyaev and Zharova 2006]. Previously, the length of 15 or 16 bits for numbers from intervals $\left[\overline{0:2^{15}-1}\right] = \left[\overline{0:32767}\right]$ and $\left[\overline{0:2^{16}-1}\right] = \left[\overline{0:65535}\right]$ accordingly was sufficient for random numbers. However, the modern tasks require the lengths of 32-64 bits for the diapasons of random numbers. Using the technology of twisting generation, which is based on congruential arrays [Matsumoto and Nishimura 1998; Matsumoto et al., 2006, 2007; Bos et al. 2011; Deon and Menyaev 2016] may be faced with limitations in some cases. Let's analyze this more in detail.

To build a twister, a congruential generation is needed. It allows organizing the random value x_{i+1} followed by current value x_i using function $f(x_i)$ limited by modulus *m*:

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

Historically, function $f(x_i)$ linear dependences from the congruential constants a and c:

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

In generator MT19937 [Matsumoto and Nishimura 1998] a realization of index *i* is accomplished by array mt, which could be placed and then initialized in computer RAM by using the following code:

```
#define N 624
static unsigned long mt[N];
static int mti;
mt[0] = seed & 0xffffffff;
for ( mti = 1; mti < N; mti++ )
  mt[mti] = ( 69069 * mt[mti-1] ) & 0xffffffff;
```

The elements in array mt include the random numbers created by linear congruential equation (2) with constants $a = 69069$ and $\mathrm{c} = 0$. The quality of generated numbers is defined by an orthogonal transformation of matrix A [Matsumoto et al., 2006, 2007]. Now our current interest is directed to the address space, which is required for 624 words having length of 4 bytes or 32 bits (that time type *long* signed 4 bytes, later it became 8 bytes). However, this size may be insufficient to realize some tasks in the address space of $log_2(624 \cdot 4) = log_2 2496 < 12$ bits using the common data bus of a computer or microcontroller.

The same approach is used for twisting random numbers with a length of 64 bits [Saito and Matsumoto 2008]. In that variation a congruential generation of an array is applied, and in turn it consists of 312 elements of random numbers.

```
#define NN 312
static unsigned long long mt[NN];
mt[0] = seed;
for (mti=1; mti<NN; mti++)
  mt[mti] =  (6364136223846793005ULL * (mt[mti-1] ^ (mt[mti-1] >> 62)) + mti);
```

This 2nd variation has the same address space, i.e. $log_2(312 \cdot 8) = log_2 2496 < 12$. Herein, one can see that the global twister of a whole sequence having 2496 bits totally, cannot provide completeness of uniform generation of random numbers. In other words, the initial congruential array has to contain more elements.

In our previous study [Deon and Menyaev 2016] it has been shown that absolute uniformity of generation is reachable only in complete sequences of random numbers. In that case each complete sequence consists of non-repeatable random numbers having w bit length. The interval of generation is defined by length w of each number $x \in [\overline{0:2^w-1}]$. In this interval the global circular twister provides absolute uniformity of generation for the initial congruential array.

```
static public int w = 16;                    // number bit length
static public int N = 1 << w;                  // sequence length
static public int[] x = new int[N];                  // sequence
static public int maskW = (int)(0xFFFFFFFF >> (32 - w));
x[0] = x0;                       // the beginning of sequence
for ( int i = 1; i < N; i++ )
   x[i] = ( a * x[i - 1] + c ) & maskW;
```

In this listing the length of a random number is 16 bits. Therefore, a congruential generation realizes the interval of all the numbers as $x \in [\overline{0:2^w-1}] = [\overline{0:2^{16}-1}] = [\overline{0:65535}]$. All of them are presented in array x only once.

While this seems promising, what should be done if the task of generation requires the numbers having accidentally received bit length w? If the technology of MT19937 is applied for that task, there is no guarantee that all the numbers are created without unpredictable skipping of elements within uniform generation. On the other side, if a technology of complete sets is used, the whole array could not be placed in the RAM of a computer or microcontroller; i.e. if the length of number is 32 bits, the 32-address buses will not have enough space for the computer program due to all available bytes being occupied by the array used for congruential twisting generation. The same story may occur with 64 bit random numbers when 64-address buses are used.

Following this, the aim of the current article is to find the solution for generation of complete sequences having uniformly distributed random numbers in interval $[\overline{0:2^w-1}]$ with accidental bit length w, i.e. without congruential-twister array technology.

2 Fundamentals

Let's consider a sequence of numbers with equal bit length w. The initial number is defined as x_0. All other numbers may be derived using congruential formulas (1) and (2). If the amount of unique non-repeatable numbers is equal 2^w, the sequence is completed due to the fact that it contains all the numbers from interval $[\overline{0:2^w-1}]$. It has been confirmed experimentally [Deon and Menyaev 2016], that congruential sequences with 2^w length may have the property of completeness if the constant a is the subject to the condition $(a-1)\ mod\ 4 = 0$, and also constant c takes odd values and is the subject to the condition $c\ mod\ 2 \neq 0$. Both constants a and c do not exceed the interval limited by $[\overline{0:2^w-1}]$.

In complete congruential sequences the global circular twister creates complete sequences as well. In addition, it should be taken into account that complete sequences have uniform singular distribution of their elements. This follows directly from the definition of completeness. Therefore, the properties of complete congruential and twisting sequences are sufficient for the following next constructions. Our goal is to abandon the initial congruential array, but the task of complete congruential twisting generation of uniformly distributed random sequences still has to be fulfilled.

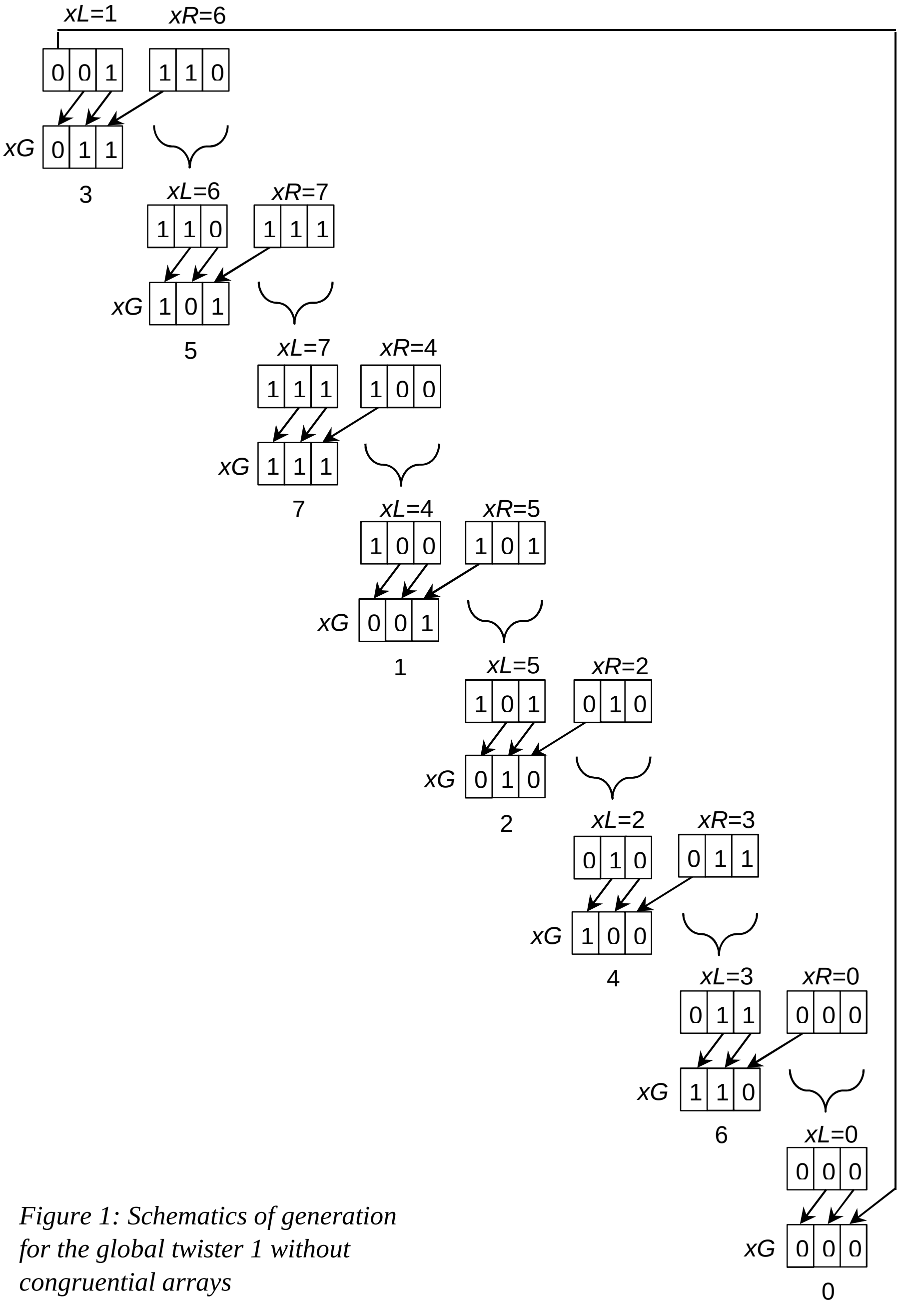

Figure 1: Schematics of generation for the global twister 1 without congruential arrays

Let there be a seed $x0 \in [\overline{1: 2^{w} - 1}]$ with a bit mask as $maskW = 0xFFFFFFFF \gg (32 - w)$. Next, let's take into account the pair of adjacent congruential numbers $< xL, xR >$ which are placed to the left and to the right. In the beginning, let's define the left value as $xL = x0$; right value is defined by the congruence $xR = (a \cdot xL + c)\ \&\ maskW$. So, the congruential sequence is generated if both operations $xL = xR$ and $xR = (a \cdot xL + c)\ \&\ maskW$ are accomplished.

The quantity of iterations is one less than the total quantity of numbers in the sequence due to the initial number coinciding with the beginning of sequence $xL = x0$. Thus, for generation of congruential twister 0 the working array couldn't be used; the subsequent random value is generated at the next stage of the iteration.

Now let's consider the twister 1, which is by definition a global circular shift to the left of congruential sequence (twister 0), with a step size of 1 bit. The forming of twister 1 for the sequence having $w = 3$ bits is displayed in Fig.1.

The length of the complete sequence is $N = 2^{w} = 2^{3} = 8$. To demonstrate an example, let's use the following values: $x_0 = 1$, $a = 5$, $c = 1$. For this variation the congruential generation creates the sequence 1 6 7 4 5 2 3 0. Based on this, twister 1 provides the following next sequence 3 5 7 1 2 4 6 0, and its elements are presented as xG values in Fig.1. In the initial iteration an adjacent congruential numbers are $xL = 1$, $xR = 6$. A simultaneous shift to the left with a step of 1 bit provides $xG = 3$. The 2nd iteration starts from the equating of xR taken from the initial iteration to $xL = xR = 6$. At the same time, the new value of xR is derived as said above: $xR = (a \cdot xL + c)\ \&\ maskW = (5 \cdot 6 + 1)\ \&\ 111_2 = 7$. The new value of $xG = 5$ is generated similarly as in the initial generation: it is a simultaneous shift to the left of $xL = 6$ and $xR = 7$ with a step of 1 bit. So, after realizing of all 8 iterations the new 8 generated numbers xG for complete twisting sequence will be received.

Below is the program code to realize described twisting technology in which the array of the initial congruential generation with subsequent global circular shift is abandoned. In each new global twister *nT* the shift of adjacent congruential numbers is made $w - 1$ times in the interval of $nW \in [\overline{1: w - 1}]$ bits with the mask *maskT*. This program code is organized as the static object class prepared in C# dialect from Microsoft Visual Studio 2013; the similar code may be demonstrated for the language C (dialect Win32) or C++ (dialect CLR). Anyway, the result is the same. The names *P030202* and *cP030202* are chosen by chance.

```
namespace P030202
{ class cP030202
  {  static public uint w = 3U;                // number bit length
     static public uint N1 = 0xFFFFFFFF >> (32 - (int)w);
     static public uint x0 = 1U;             // sequence beginning
     static public uint xB = x0;     // current twister beginning
     static public uint xG;              // created random number
     static public uint xL = 0U, xR = x0;    // paired numbers
     static public uint a = 5U;         // congruential constant a
     static public uint c = 1U;         // congruential constant c
     static public uint nW = 0U;// paired twister number in w
```

```
    static public uint nT = 0U;       // twister number in nwN
    static public uint nV = 0U;          // element number in x
    static public uint maskW = 0xFFFFFFFF >> (32-(int)w);
    static public uint maskU = 1U << ((int)w - 1);      // elder
    static public uint maskT = maskU;      // twister first bits
//-------------------------------------------------------------------
    static void Main(string[] args)
    {  uint N = N1 + 1;                          // sequence length
      Console.WriteLine("w = {0}  N1 = {1}  N = {2}",
                    w, N1, N);
      Console.WriteLine("a = {0}  c = {1}", a, c);
      int k = 1;                               // sequence number
      for (int nT = 0; nT < N; nT++)
      {  for (nW = 0; nW < w; nW++)
        {  maskT = maskU;           // twister mask beginning
          for (int m = 1; m < nW; m++)
            maskT |= maskU >> m;
          Console.Write("k = {0,3} | ", k++);
          xR = xB;                      // sequence beginning
          for (int i = 0; i < N; i++)
          {  xL = xR;                          // pair left value
            xR = Cong(xL);                    // pair right value
            if (nW == 0) xG = xL;          // congruential pair
            else xG = TwistPair();                 // twister pair
            Console.Write("{0,3}", xG);
          }
          Console.Write("  |  nT = {0,2}  nW = {1}  ",
                      nT, nW);
          if (nW == 0) Console.WriteLine();
          else Console.WriteLine("maskT={0:X}", maskT);
        }
        xB = Cong(xB);                          // next beginning
      }
      Console.ReadKey();                         // result viewing
    }
//-------------------------------------------------------------------
    static uint Cong(uint z)
    { return (a * z + c) & maskW;                   // next value
    }
//-------------------------------------------------------------------
    static uint TwistPair()
    {  uint g = (xR & maskT) >> (int)(w - nW);        // elder
      return ((xL << (int)nW) & maskW) | g;       // younger
    }
//=========================================
  }
}
```

After executing this code the following listing appears.

```
w = 3  N1 = 7  N = 8
a = 5  c = 1
k =  1 |  1  6  7  4  5  2  3  0  | nT=0 nW=0
k =  2 |  3  5  7  1  2  4  6  0  | nT=0 nW=1 maskT=4
k =  3 |  7  3  6  2  5  1  4  0  | nT=0 nW=2 maskT=6
k =  4 |  6  7  4  5  2  3  0  1  | nT=1 nW=0
k =  5 |  5  7  1  2  4  6  0  3  | nT=1 nW=1 maskT=4
k =  6 |  3  6  2  5  1  4  0  7  | nT=1 nW=2 maskT=6
k =  7 |  7  4  5  2  3  0  1  6  | nT=2 nW=0
k =  8 |  7  1  2  4  6  0  3  5  | nT=2 nW=1 maskT=4
k =  9 |  6  2  5  1  4  0  7  3  | nT=2 nW=2 maskT=6
k = 10 |  4  5  2  3  0  1  6  7  | nT=3 nW=0
k = 11 |  1  2  4  6  0  3  5  7  | nT=3 nW=1 maskT=4
k = 12 |  2  5  1  4  0  7  3  6  | nT=3 nW=2 maskT=6
k = 13 |  5  2  3  0  1  6  7  4  | nT=4 nW=0
k = 14 |  2  4  6  0  3  5  7  1  | nT=4 nW=1 maskT=4
k = 15 |  5  1  4  0  7  3  6  2  | nT=4 nW=2 maskT=6
k = 16 |  2  3  0  1  6  7  4  5  | nT=5 nW=0
k = 17 |  4  6  0  3  5  7  1  2  | nT=5 nW=1 maskT=4
k = 18 |  1  4  0  7  3  6  2  5  | nT=5 nW=2 maskT=6
k = 19 |  3  0  1  6  7  4  5  2  | nT=6 nW=0
k = 20 |  6  0  3  5  7  1  2  4  | nT=6 nW=1 maskT=4
k = 21 |  4  0  7  3  6  2  5  1  | nT=6 nW=2 maskT=6
k = 22 |  0  1  6  7  4  5  2  3  | nT=7 nW=0
k = 23 |  0  3  5  7  1  2  4  6  | nT=7 nW=1 maskT=4
k = 24 |  0  7  3  6  2  5  1  4  | nT=7 nW=2 maskT=6
```

This result may be compared with similar studies in a previous simulation [Deon and Menyaev 2016]: they are equal. However, the difference is that in a current program code the algorithm realized in function *Main()* does not use the initial congruential array which is applied in [Matsumoto and Nishimura 1998; Deon and Menyaev 2016]. Thus, the achieved fact is that generation of received complete sequences of random numbers isn’t limited by technical options of the computer equipment. The length of the complete sequence may be any size long, however, the bit length *w* of random numbers may be limited by capabilities of common data bus of 32 or 64 bits used for logical operations in computers.

3 Code States

In the previous section, the simulation technique of the complete sequence uses the generalizing cycles for calculation of random numbers. Practical applications mostly need the technology providing the random value after a single order. To obtain that code let us use the programming technique based on the method of storing state *st* for generator. The state transition diagram from the current state to the next one is shown in Fig. 2. The assignment is the following:

- state 101 provides the congruential generation of random numbers in complete sequence;
- state 102 allows tuning of parameters for twisting generation;
- state 103 realizes the twisting generation of random numbers in complete sequence;
- state 104 sets the congruential beginning for the next congruential and twisting complete sequence;
- state 105 organizes the ending of generation process for the all possible congruential and twisting complete sequences, and it sets the repeatable beginning state of generator.

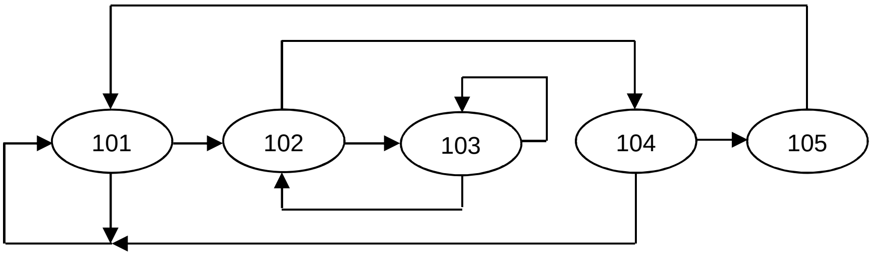

Figure 2: Code states for generation of random numbers

Below is the program code which realizes the common generation of random numbers. In the main function *Main()* each random value *uint x* is created independently. All of them together are collected in complete random sequences. Functions *Cong()* and *TwistPair()* are the same as in program *P030202* in previous section *Fundamentals*. The names *P030301* and *cP030301* are chosen by chance.

```
namespace P030301
{  class cP030301
   {  static public uint w = 3U;                // number bit length
      static public uint N1 = 0xFFFFFFFF >> (32 - (int)w);
      static public uint x0 = 1U;              // sequence beginning
      static public uint xB = x0;    // current twister beginning
      static public uint xG;              // created random number
      static public uint xL = 0U, xR = x0;    // paired numbers
      static public uint a = 5U;          // congruential constant a
      static public uint c = 1U;          // congruential constant c
      static public uint st1 = 101U;// state of generation of xG
      static public uint nW = 0U;// paired twister number in w
      static public uint nT = 0U;       // twister number in nwN
      static public uint nV = 0U;           // element number in x
      static public uint maskW = 0xFFFFFFFF >> (32-(int)w);
      static public uint maskU = 1U << ((int)w -1);      // elder
      static public uint maskT = maskU;       // twister first bits
//---------------------------------------------------------------------
      static void Main ( string[] args )
```

```
   {  uint N = N1 + 1;
      Console.WriteLine ( "w = {0}  N = {1}", w, N );
      Console.WriteLine ( "a = {0}  c = {1}", a, c );
      for ( int k = 1; k <= w * N; k++ )
      {  Console.Write ( "k = {0,3}  | ", k );
         for ( int i = 0; i < N; i++ )
         {  uint x = Next ();                  // random number
            Console.Write ( "{0,3}", x );
         }
         Console.Write ( "  |  nT = {0,2}  nW = {1}",
                                   nT, nW );
         Console.WriteLine ();
      }
      Console.ReadKey ();                      // result viewing
   }
//-----------------------------------------------------------------------
   static public uint Next ()
   {  Next1 ();                    // random number generation
     return xG;                    // generated random number
   }
//-----------------------------------------------------------------------
   static bool Next1()
   {  bool FlagNext1 = false;             // xG will be created
      bool FlagWhile1 = true;             // looking for twister
      while ( FlagWhile1 )                 // st1 states running
      {  switch ( st1 )                      // states switching
        {  case 101U:              // congruential generation
             xL = xR;                         // beginning of pair
             xR = Cong ( xL );                   // ending of pair
             xG = xL;                         // generated number
             if ( nV < N1 ) nV++;                  // next number
             else st1 = 102U;
             FlagWhile1 = false;          // number is created
             break;
          case 102U:        // preparation to pair twister nW
             nW++;               // a pair twister number in w
             if ( nW < w )
             {  maskT = maskU;   // elder 1 in twister mask
                for ( int m = 1; m < nW; m++ )
                   maskT |= maskU >> m;    // twister mask
                xL = xB;                     // twister beginning
                xR = Cong( xL );                    // pair xL, xR
                nV = 0U;         // value number in twister nT
                st1 = 103U;               // generate twister nT
             }
             else st1 = 104U;
             break;
```

```
            case 103U:                        // twister generation
               xG = TwistPair ();           // result of generation
               xL = xR;              // beginning of the next pair
               xR = Cong ( xL );                          // next pair
               if ( nV == N1 ) st1 = 102U;
               else nV++;                       // next value number
               FlagWhile1 = false;          // number is created
               break;
            case 104U:     // the end of twisters nW inside nT
               if ( nT < N1 )
               {  nT++;    // nT number for twister group nW
                  xB = Cong( xB );                     // beginning
                  xR = xB;
                  nW = 0U;                 // twister number in w
                  nV = 0U;              // value number in twister
                  st1 = 101;           // generation of twister nT
               }
               else st1 = 105U;
               break;
            case 105U:                        // initial parameters
               nW = 0U;
               nT = 0U;
               xR = x0;
               st1 = 101U;                    // initial parameters
               break;
          }  // switch
        }      // while
        return FlagNext1;                  // result of generation
    }
//---------------------------------------------------------------------
// functions Cong and TwistPair
//==========================================
  }
}
```

After this code execution the listing below appears; it is presented here with abridgments for what the dash lines are used.

```
w = 3  N = 8
a = 5  c = 1
k =   1 | 1  6  7  4  5  2  3  0
k =   2 | 3  5  7  1  2  4  6  0
k =   3 | 7  3  6  2  5  1  4  0
k =   4 | 6  7  4  5  2  3  0  1
k =   5 | 5  7  1  2  4  6  0  3
k =   6 | 3  6  2  5  1  4  0  7
- - - - -
k =  22 | 0  1  6  7  4  5  2  3
```

```
k=  23 | 0  3  5  7  1  2  4  6
k=  24 | 0  7  3  6  2  5  1  4
```

Complete sequences in the listing are similar with ones which appear after execution of program *P030202* in the previous section *Fundamentals*. The only difference is that now each random value is received without using the cycles of transitional twisters.

4 Constructions and results

When creating the twisting sequences an important note should be addressed when choosing parameters a and c which are received in accordance with (2) for generation of random numbers having w bits. Both parameters have to belong to the interval $a, c \in [\overline{0{:}\, 2^w - 1}]$ and have to satisfy the following properties:

$$\begin{cases} (a-1)\ mod\ w = 0 \\ c\ mod\ 2\ \neq 0\,, \quad \text{for odd numbers} \end{cases} \tag{3}$$

The automatic tuning of parameter a satisfying the subinterval $[\overline{ab, ae}] \subset [\overline{0{:}\, 2^w - 1}]$ may be performed in various ways. In this current work for constant a the algorithm of double interval modeling is applied. The diagram of two subintervals $a1 + a2 = [\overline{a1b, a1e}] + [\overline{a2b, a2e}] \subset [\overline{0{:}\, 2^w - 1}]$ is showed in Fig.3.

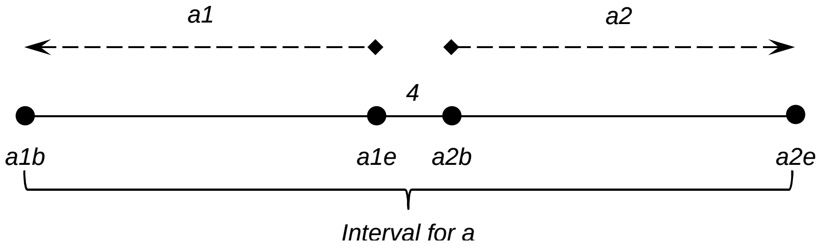

Figure 3: Schematics of interval realization for congruential constant a

The value $a1e$ is to the left from $N/2$, while value $a2b = a1e + 4$ is to the right from $N/2$. Moving of a in interval $a1$ is accomplished from the right to the left, i.e. from $a1e$ to $a1b$ with a step of -4; moving of a in interval $a2$ is accomplished from the left to the right, i.e. from $a2b$ to $a2e$ with a step of +4. This choice for a was made artificially to provide better confusion for generation. Congruential parameter c could take all the odd numbers in the interval $[\overline{1{:}\, 2^w - 1}]$ that range from 1 to $2^w - 1$.

In Fig.4 the code states are shown for the random number generation with automatic tuning of congruential constants and masks of twisters. The states starting from 10x are the same as in previous section *Code State*, thus, their description is skipped below. The other assignments are the following:

- state 1 begins the total generation;
- state 2 sets the parameters for block 1 to continue the generation after

congruential constants are changed;

- state 201 provides the changing of constant *c*;
- state 202 defines the subintervals *a1* or *a2* for constant *a*;
- state 203 derives the new value for *a* in subinterval *a1*;
- state 204 derives the new value for *a* in subinterval *a2*;
- state 205 ends the total generation and provides transition to state 1.

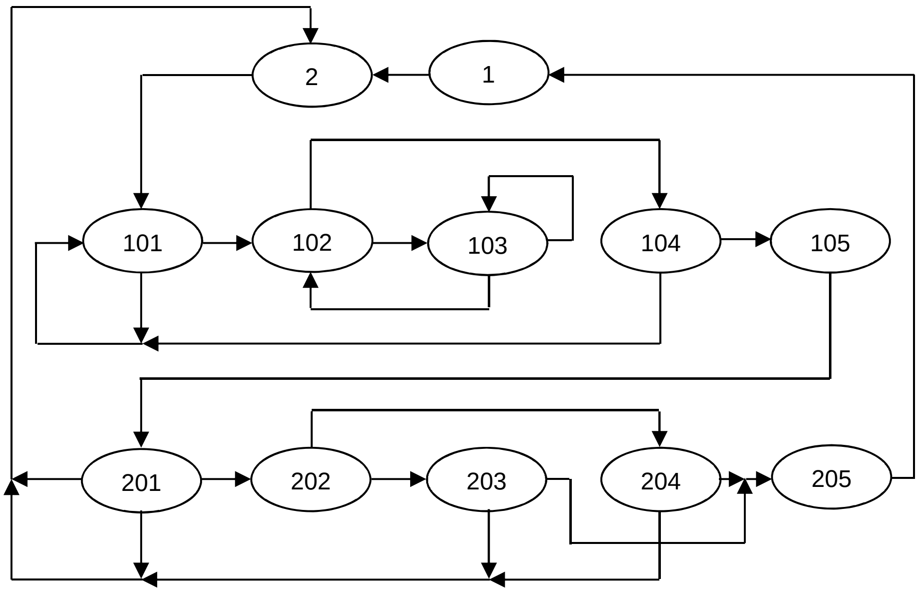

Figure 4: Code states for tuning of random number generation

Below is the program code for the name space *nsDeonYuliTwist32D* in which dynamic class *cDeonYuliTwist32D* provides twisting generation of random numbers having any length up to 32 bits.

```
namespace nsDeonYuliTwist32D
{  class cDeonYuliTwist32D
   {  public uint w = 16U;                        // number bit length
      public uint N1 = 0U;                        // maximal number
      public uint x0 = 1U;                     // sequence beginning
      uint xB = 1U;                       // current twister beginning
      uint xG = 0U;                         // created random number
      uint xL = 0U, xR = 1U;                          // paired numbers
      double abf = 0.39;                   // relative beginning of a
      double aef = 0.39;                       // relative ending of a
      public uint a1b = 1U, a1e = 0U;                  // interval a1
      uint a1s = 0U;                             // state of interval a1
      public uint a2b = 1U, a2e = 0U;                  // interval a2
```

```
    uint a2s = 0U;                          // state of interval a2
    uint a1 = 5U;                         // constant for interval a1
    uint a2 = 5U;                         // constant for interval a2
    uint nA = 1U;                 // constant number for a1 or a2
    public uint a = 5U;           // current value of constant a
    double cbf = 0.1;                  // relative beginning of c
    double cef = 0.3;                       // relative ending of c
    public uint cb = 1U, ce = 0U;                      // interval c
    public uint c = 1U;              // congruential constant c
    uint stG = 0U;                         // state group number
    uint st0 = 1U;                            // initial state group
    uint st1 = 101U;                      // xG generation group
    uint st2 = 201U;                   // parameter change group
    public uint nW = 0U;          // pair twister number in w
    public uint nT = 0U;                        // twister number
    public uint nV = 0U;                // element number in x
    public uint maskW = 0U;                     // number mask
    public uint maskU = 0U;                    // elder bit mask
    public uint maskT = 0U;                          // twister bits
//---------------------------------------------------------------------
    public cDeonYuliTwist32D()
    {  N1 = 0xFFFFFFFF >> (32 - (int)w);// max-number
       x0 = N1 / 7;                        // sequence beginning
    }
//---------------------------------------------------------------------
    public uint Next()
    {  bool FlagNext = true;
       while (FlagNext)
       {  switch (stG)                              // state groups
          {  case 0U:                         // initial state group
                FlagNext = DeonYuli_Next0();
                break;
             case 1U:                      // xG generation group
                FlagNext = DeonYuli_Next1();
                break;
             case 2U:                 // change parameter group
                FlagNext = DeonYuli_Next2();
                break;
          }  // switch
       }     // while
       return xG;                      // created random number
    }
//---------------------------------------------------------------------
    bool DeonYuli_Next0()
    {  bool FlagWhile0 = true;            // parameter setting
       while (FlagWhile0)                   // st0 states running
       {  switch (st0)                         // states switching
```

```
        {  case 1U:                              // initial actions
              nA = 1U;              // generation begins in a1
              a1s = 1U;                 // create twister 0 in a1
              a2s = 0U;                      // a2 while not used
              a1 = a1e;                      // end of interval a1
              a = a1;                          // current constant a
              a2 = a2b – 4U;    // left from beginning of a2
              c = cb;                   // beginning of interval c
              st0 = 2U;                 // generation parameters
              break;
           case 2U:                // for changed parameters
              xB = x0;                     // sequence beginning
              xR = xB;                       // end of pair xL, xR
              nT = 0U;                         // twister 0 number
              nW = 0U;                    // twister number in w
              nV = 0U;                    // initial value number
              stG = 1U;             // xG group of generation
              st1 = 101U;                        // xG generation
              FlagWhile0 = false;                       // finish job
              break;
        }   // switch
     }      // while
     return true;                 // xG generation is necessary
  }
//---------------------------------------------------------------------
  bool DeonYuli_Next1()
  {  bool FlagNext1 = false;           // xG will be created
     bool FlagWhile1 = true;        // looking for a twister
     while (FlagWhile1)                  // st1 states running
     {  switch (st1)                        // states switching
        {  case 101U:             // congruential generation
              xL = xR;                      // beginning of pair
              xR = DeonYuli_Cong(xL);    // pair xL, xR
              xG = xL;                      // generated number
              if (nV < N1) nV++;                // next number
              else st1 = 102U;
              FlagWhile1 = false;      // number is created
              break;
           case 102U:                    // for pair twister nW
              nW++;               // pair twister number in w
              if (nW < w)
              {  maskT = maskU;// elder 1 in twister mask
                 for (int m = 1; m < nW; m++)
                    maskT |= maskU >> m; // twister mask
                 xL = xB;               // beginning of twister
                 xR = DeonYuli_Cong(xL); // pair xL, xR
                 nV = 0U;    // value number in twister nT
```

```
          st1 = 103U;            // generate twister nT
        }
        else st1 = 104U;
        break;
      case 103U:                      // twister generation
        xG = DeonYuli_TwistPair();   // pair twister
        xL = xR;           // beginning of the next pair
        xR = DeonYuli_Cong(xL);    // пара xL, xR
        if (nV == N1) st1 = 102U;
        else nV++;                   // next value number
        FlagWhile1 = false;      // number is created
        break;
      case 104U:                 // nW twisters end in nT
        if (nT < N1)
        {  nT++;      // nT number for twisters in nW
          xB = DeonYuli_Cong(xB);
          xR = xB;
          nW = 0U;              // twister number in w
          nV = 0U;          // value number in twister
          st1 = 101U;          // nT twister generation
        }
        else st1 = 105U;
        break;
      case 105U:                      // constant changing
        stG = 2U;            // change parameter group
        st2 = 201U;             // parameters changing
        FlagWhile1 = false;          // group move out
        FlagNext1 = true;       // moving into group 2
        break;
    }   // switch
  }      // while
  return FlagNext1;                 // result of generation
}
//-------------------------------------------------------------------
bool DeonYuli_Next2()
{  bool FlagNext2 = true;                   // to be changed
   bool FlagWhile2 = true;                    // running flag
   while (FlagWhile2)
   {  switch (st2)
      {  case 201U:                    // change parameter c
           c += 2U;                          // next constant c
           if (c <= ce)
           {  stG = 0U;              // initial action group
              st0 = 2U;            // current initial action
              FlagWhile2 = false;            // work is over
              FlagNext2 = true;      // move into group 0
           }
```

```
  else st2 = 202U;
  break;
case 202U:                    // change interval for a
  c = cb;                           // initial value c
  if (nA == 1U) nA = 2U; else nA = 1U;
  if (nA == 1U) st2 = 203U;        // interval a1
  else st2 = 204U;                  // interval a2
  break;
case 203U:                      // new value from a1
  a1 -= 4U;
  if (a1 < a1b)                         // a1 is over
  {  a1s = 2U;                // interval a1 is over
     st2 = 205U;              // states of intervals
     break;
  }
  a = a1;                         // current constant a
  c = cb;                 // beginning of constant c
  a1s = 1U;                   // interval a1 is active
  stG = 0U;                   // initial action group
  st0 = 2U;                 // current initial actions
  FlagWhile2 = false;               // work is over
  FlagNext2 = true;         // move into group 0
  break;
case 204U:                      // new value from a2
  a2 += 4U;
  if (a2 > a2e)                      // a2 is finished
  {  a2s = 2U;               // interval a2 is over
     st2 = 205U;                    // interval states
     break;
  }
  a = a2;                         // current constant a
  c = cb;                 // beginning of constant c
  a2s = 1U;                   // interval a2 is active
  stG = 0U;                   // initial action group
  st0 = 2U;          // common initial generation
  FlagWhile2 = false;               // work is over
  FlagNext2 = true;         // move into group 0
  break;
case 205U:           // one of a1 or a2 is finished
  if (a2s != 2U) st2 = 204U;
  else if (a1s != 2U) st2 = 203U;
       else
       {  stG = 0U;                         // group 0
          st0 = 1U;            // common beginning
          FlagWhile2 = false;     // work is over
          FlagNext2 = true;// move into group 0
       }
```

```
            break;
        }
     }
     return FlagNext2;                          // work is over
  }
//-------------------------------------------------------------------
  uint DeonYuli_Cong(uint z)
  {  return (a * z + c) & maskW;        // the next number
  }
//-------------------------------------------------------------------
  uint DeonYuli_TwistPair()
  {  uint g = (xR & maskT) >> (int)(w - nW);      // elder
     return ((xL << (int)nW) & maskW) | g;    // younger
  }
//-------------------------------------------------------------------
  public void Start()
  {  N1 = 0xFFFFFFFF >> (32 - (int)w); // max-number
     maskW = 0xFFFFFFFF >> (32 - (int)w);
     maskU = 1U << ((int)w - 1);          // elder bit mask
     maskT = maskU;                        // first twister bit
     DeonYuli_SetA();               // set a1 and a2 borders
     DeonYuli_SetC();                          // set c borders
     x0 &= maskW;
     stG = 0U;                         // xG generation group
     st0 = 1U;                      // generator initialization
  }
//-------------------------------------------------------------------
  public void TimeStart()
  {  x0 = (uint)DateTime.Now.Millisecond;
     Start();                                 // generator starts
  }
//-------------------------------------------------------------------
  public void SetW(int sw)
  {  w = (uint)Math.Abs(sw);          // number bit length
     if (w < 3U) w = 3U;                          // min-length
     else if (w > 32U) w = 32U;               // max-length
     N1 = 0xFFFFFFFF >> (32 - (int)w); // max-number
     x0 = N1 / 7U;                      // sequence beginning
  }
//-------------------------------------------------------------------
  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 = (uint)(N1 * abf);             // bottom edge for a1
       a1b = DeonYuli_PlusA(a1b);       // beginning for a1
       a2e = (uint)(N1 * aef);                  // top edge for a2
       a2e = DeonYuli_MinusA(a2e);         // ending for a2
       uint r = a2e - a1b;
       if (a1b >= a2e)               // interval for a as a point
       {  a1e = a1b;                           // a1 is a one point
          a2b = a1b;                           // interval a2 as a1
          a2e = a2b;                           // a2 is a one point
          return;
       }
       if (r == 4U)                        // one point a1 and a2
       {  a1e = a1b;                              // a1 is one point
          a2b = a2e;                              // a2 is one point
          return;
       }
       if (r == 8U)          // a1 has 2 points, a2 – one point
       {  a1e = a1b + 4U;                          // ending of a1
          a2b = a2e;                          // beginning of a2
          return;
       }
       a1e = (a1b + a2e) / 2U;                    // middle for a
       a1e = DeonYuli_MinusA(a1e);   // left from middle
       a2b = a1e + 4U;                     // right from middle
    }
//-------------------------------------------------------------------
    uint DeonYuli_PlusA(uint a)
    {  if (a < 1U) { a = 1U; return a; }
       uint z = a;                              // bottom edge for a
       for (uint i = 0U; i < 3U; i++)
          if (a % 4U != 0U) a--;          // uniform condition
          else break;
       a++;                           // true value for constant a
       if (a < z) a += 4U;           // right from bottom edge
       if (a >= N1 - 1) a -= 4U;            // left from top edge
       return a;
    }
//-------------------------------------------------------------------
    uint DeonYuli_MinusA(uint a)
    {  if (a < 1U) { a = 1U; return a; }
       uint z = a;                              // bottom edge for a
       for (uint i = 0U; i < 3U; i++)
          if (a % 4U != 0U) a--;         // uniform condition
          else break;
       a++;                          // true value for constant a
```

```
         if (a > z) a -= 4U;                   // left from top edge
         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 = (uint)(N1 * cbf);              // bottom edge for c
         if (cb % 2U == 0U) cb += 1; // only odd value for c
         if (cb > N1) cb = N1;                        // max-value
         ce = (uint)(N1 * cef);                  // top edge for c
         if (ce % 2U== 0U) ce -= 1U;                // only odd
         if (ce > N1 - 1) ce = N1;                   // max-value
         if (cb > ce) ce = cb;
         c = cb;       // beginning of congruential constant c
      }
//----------------------------------------------------------------------
      public void SetX0(double xs)
      {  x0 = (uint)(N1 * Math.Abs(xs));
      }
//==========================================
   }
}
```

In class *cDeonYuliTwist32D* several variables are reserved. They can be tuned with the help of encapsulated functions. As a first example let's use the values for default settings to generate several random numbers having $w = 16$ bits length and belonging to interval $[0: 2^w - 1] = [0: 2^{16} - 1] = [0: 65535]$. The program code for this task is below. The names *P030401* and *cP030401* are chosen by chance.

```
using nsDeonYuliTwist32D;          // twister generator class
namespace P030401
{  class cP030401
   {  static void Main(string[] args)
      {  cDeonYuliTwist32D CT =
                                  new cDeonYuliTwist32D ();
         CT.Start();
         for ( int j = 0; j < 8; j++ )
         {  uint z = CT.Next ();               // random number
            Console.Write ( "{0,7} ", z );              // monitor
         }
         Console.ReadKey ();                      // result viewing
```

```
    }
  }
}
```

After execution the following result appears:

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

These random numbers are equal to the other ones under the same conditions but with help of twisting array in program *P020401* [Deon and Menyaev 2016].

Now let's look at another variant in which there is no possibility to use the congruential twisting array for computers having 32 bits of common data bus because of a lack of memory space for the program. The next program *P030402* can generate random numbers with a length of 32 bits using technology without congruential twisting array. Program names *P030402* and *cP030402* are chosen by chance.

```
using nsDeonYuliTwist32D;         // twister generator class
namespace P030402
{  class cP030402
   {  static void Main(string[] args)
      {  cDeonYuliTwist32D CT =
                              new cDeonYuliTwist32D ();
         CT.SetW ( 32 );                    // number bit length
         CT.Start();                        // generator starts
         Console.WriteLine(
              "CT.x0 = {0}  CT.a = {1}  CT.c = {2}",
               CT.x0, CT.a, CT.c);
         for ( int j = 0; j < 8; j++ )
         {  uint z = CT.Next ();              // random number
            Console.WriteLine ( "{0,10} ", z );        // monitor
         }
         Console.ReadKey();                    // result viewing
      }
   }
}
```

The result of execution is the following:

```
 CT.x0 = 613566756  CT.a = 5  CT.c = 429496729
 613566756
3767299885
3711097170
  85104163
2840182256
2787589065
 706196094
2953448863
```

In the next code let's consider complete automatic tuning of parameters for generator. Below is the program code, which allows tuning of the generator to different values of w, $x0$, $a1b$, $a2e$. As an example, the values are taken as: $w = 4$, $N = 2^w = 16$, $a1b = 1$, $a2e = 13$, $x0 = 1.0 \cdot (N - 1) = 15$. For each meaning of constant $a = 5,9,1,13$ and constant $c = 1, 3, 5, 7, 9, 11, 13, 15$. Program names *P030403* and *cP030403* are chosen by chance.

```
using nsDeonYuliTwist32D;              // twisting generator
namespace P030403
{  class P030403
   {  static void Main(string[] args)
      {  cDeonYuliTwist32D CT =
                                  new cDeonYuliTwist32D();
         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.SetX0(1.0);                 // sequence beginning
         CT.Start();                         // generator starts
         int N = (int)CT.N1 + 1;            // sequence length
         Console.WriteLine("w = {0}   N = {1}", CT.w, 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);
         Console.WriteLine("x0 = {0}", CT.x0);
         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 < 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 < 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 program execution; it is presented with abridgements where dash lines are for skipped strings.

```
w = 4   N = 16
a1b = 1   a1e = 5
a2b = 9   a2e = 13
cb = 1   ce = 15
x0 = 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 = 11  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   7  9  4 10  1 11  6 12  3 13  0 14  5 15  2  8   a = 13  c = 15
Finish
NN = 32768
```

A total of $4 \cdot 16 \cdot 4 \cdot 8 = 2048$ sequences have been received. Since each sequence contains 16 non-repeatable random values, the total amount of generated numbers is $2048 \cdot 16 = 32768$. This result is equal to that one which was received with help of twisting array under the same conditions in program *P020403* [Deon and Menyaev 2016].

5 Discussion

Presented in the previous section, generator *nsDeonYuliTwist32D* is able to create random numbers having accidental bit length in diapason $3 \leq w \leq 32$. These numbers are distributed uniformly in the interval $[\overline{0: 2^w - 1}]$. Among other things, the uniformity $U(w)$ for any number can be either unique $U(w) = 1$, or multivariate $U(w) > 1$. If all the complete sequences are generated, the uniformity $U(w)$ for all the numbers in all of them is the same. If single complete sequence is generated and it has 2^w numbers, that means each number may appear only once in this sequence, i.e. $U(w) = 1$. The difficulty, as it's mentioned above, is that computers with 32 bits of common data bus can't provide the array capable to contain $2^w = 2^{32}$ elements due to a lack of memory space for the computer program in this case. The external hard drive might help, but that solution is limited by slow processes. So, let's try to use RAM only.

Below is the program code in which the array of counters *cX* is used; it contains $ncX = 2^{28} = 268{,}435{,}456$ elements. Based on this array, the generator creates a single complete sequence with a length of $2^w = 2^{32} = 4{,}294{,}967{,}296$ elements. Each element is $w = 32$ bits long. Therefore, any generated random number x

belongs to interval $x \in \overline{[0: 2^w - 1]} = \overline{[0: 2^{32} - 1]} = \overline{[0: 4{,}294{,}967{,}295]}$. For each element in array *cX* let's count the amount of same values for which the index j is used. Because the generator creates all the numbers in complete sequence it means that counters in the beginning add up the quantity of elements $x \in \overline{[0: 268{,}435{,}456]}$. If the meaning of each element in *cX* is equal to 1, it means the generator at the initial iteration has created the random numbers from interval $\overline{[0: 268{,}435{,}456]}$ which may be found once only. This also corresponds to the statement that in a complete sequence there are 268,435,456 initial numbers from the interval of complete sequence$\overline{[0: 4{,}294{,}967{,}296]}$.

By organizing the program code so it performs $2^4 = 16$ iterations for 2^4 intervals; and with the total amount of elements being 2^{28}, which are then sorted out successively and fully within complete sequence $\overline{[0: 2^{32} - 1]}$, now it is possible to find out the answer to the question about unique properties of the numbers in a single complete sequence. This statement has to correspond with the uniform unique distribution $U(w) = 1$, which is provided by *nsDeonYuliTwist32D* generator for the initial complete twister 0. Next program names *P030511* and *cP030511* are chosen by chance.

```
using nsDeonYuliTwist32D;          // twister generator class
namespace P030511
{  class cP030511
   {  static void Main ( string[] args )
      {  cDeonYuliTwist32D CT =
                                   new cDeonYuliTwist32D();
         CT.SetW ( 32 );                // set a number bit length
         CT.Start ();                          // generator starts
         uint w = CT.w;                     // number bit length
         uint N1 = CT.N1;                         // max-number
         Console.WriteLine ( "w = {0}  N1 = {1}", w, N1 );
         uint N28_1 = 0xFFFFFFF;                  // work max
         uint N28 = N28_1 + 1;          // work interval length
         Console.WriteLine ( "N28_1 = {0:X}  N28 = {1}",
                                   N28_1, N28 );
         uint[] cX = new uint[N28];        // repeating counter
         uint Total1 = 0;    // total quantity of single numbers
         uint q1 = 0U;          // quantity of one time numbers
         uint q2 = 0U;        // quantity of two times numbers
         uint cXD = N28_1;                      // last index in cX
         uint cXB = 0U;                   // beginning of interval
         uint cXE = cXB + cXD;           // ending of interval
         Console.WriteLine ( "m     cXB     cXE   q1   q2”);
         for ( uint m = 1U; m <= 16U; m++ ) //with intervals
         {  Console.Write ( "{0,2}", m );
            Console.Write ( "  {0,10}  {1,10}",  cXB, cXE);
            for ( uint i = 0U; i <= cXD; i++ ) cX[i] = 0U;
            uint n = 0U;       // number counter inside interval
            while (true)
```

```
          {  uint x = CT.Next();            // random number
             if (cXB <= x && x <= cXE) cX[x - cXB]++;
             if ( n == N1 ) break;             // end of interval
             n++;                  // quantity of created numbers
          }
          q1 = 0U;             // quantity of one time numbers
          q2 = 0U;           // quantity of two times numbers
          for (uint i = 0; i < N28; i++)
             if (cX[i] == 1) q1++;        // one time numbers
             else if (cX[i] == 2) q2++;// two times numbers
          Console.WriteLine("  {0}  {1}", q1, q2);
          if (m == 16) break;
          Total1 += q1;          // total of one time numbers
          cXB = cXE + 1;                   // the next interval
          cXE = cXB + cXD;
       }
       Total1 = (q1 - 1) + Total1;
       Console.WriteLine("Total = {0} + 1", Total1);
       Console.ReadKey();                   // result viewing
    }
  }
}
```

After the program execution the listing below appears on monitor.

```
w = 32  N1 = 4294967295
N28_1 = FFFFFFF  N28 = 268435456
 m        cXB          cXE          q1      q2
 1                0    268435455   268435456   0
 2    268435456    536870911   268435456   0
 3    536870912    805306367   268435456   0
 4    805306368   1073741823   268435456   0
 5   1073741824   1342177279   268435456   0
 6   1342177280   1610612735   268435456   0
 7   1610612736   1879048191   268435456   0
 8   1879048192   2147483647   268435456   0
 9   2147483648   2415919103   268435456   0
10   2415919104   2684354559   268435456   0
11   2684354560   2952790015   268435456   0
12   2952790016   3221225471   268435456   0
13   3221225472   3489660927   268435456   0
14   3489660928   3758096383   268435456   0
15   3758096384   4026531839   268435456   0
16   4026531840   4294967295   268435456   0
Total1 = 4294967295 + 1
```

These received results show that in the general interval of generation $[0: 2^{32} - 1]$ all subintervals with length of 2^{28} numbers are passed 16 times. Parameters *cXB* and *cXE* point out the beginning and the ending of each subinterval. Values for counters *q1*, which are for momentary observation of random numbers, are equal for all the subintervals and was determined as 268435456. That is equal to the subinterval length $2^{28} = 268{,}436{,}456$ which means each random number was definitely observed once. The values $q2 = 0$ demonstrate that no unique number is generated twice or more times. It means the generator indeed exhibits properties of uniform generation of random numbers.

The total amount of bits $N_b(w, N)$ for the sequence consisting in N numbers, with them being w bits long, may be found as:

$$N_b(w, N) = w \cdot N. \tag{4}$$

Let's consider the proposed generator *nsDeonYuliTwist32D* in comparison with the well-known generator MT19937 [Matsumoto and Nishimura 1998], which uses a congruential twisting array having 624 elements, and each element is 32 bits long. In this case the bit length of sequence is the following: $MT_b(w, N) = MT_b(32{,}624) = 32 \cdot 624 - 31 = 19968 - 31 = 19937$. Diminution of value 31 is because the generator MT19937 doesn't take into account the circle of needless bits. In MT19937 only the transformation of the next congruential twisting number with a length of 32 bits happens. That means, a recurrence interval in this case is $R_{MT19937} = 2^{19937} - 1$.

In the generator *nsDeonYuliTwist32D* considered here, the maximal sequences are created with the values having length of 32 bits. In this case the bit length of sequence is determined as follows: $N_b(w, N) = N_b(32, 2^{32}) = 32 \cdot 4294967296 = 2^{37} = 137{,}484{,}953{,}472$. In turn, a recurrence interval is defined as $R_{nsDeonYuliTwist32D} = 2^{2^{37}} - 1$. This value is extremely big, and in general it means that potency (or computational capability) of generator *nsDeonYuliTwist32D* is in excess of generator MT19937 due to $R_{nsDeonYuliTwist32D} \gg R_{MT19937}$.

The complete twister consists of the initial congruential sequence and the sequences which are received by single circular bit shifts $N_t = w \cdot N - 1$ times. Thus, total amount $N_T(w, N_t)$ of sequences of complete twister contains the sum of single congruential sequence, i.e. twister 0, and all other twisting sequences:

$$N_T(w, N_t) = 1 + N_t = 1 + (w \cdot N - 1) = w \cdot N = w \cdot 2^w. \tag{5}$$

From the definitions of $N_T(w, N_t)$ and $N_b(w, N)$ it follows that for complete sequences they are equal $N_T(w, N_t) = N_b(w, N)$. For the case of no twisters the equation is $N_T(w, N_t = 0) = 1 + 0 = 1$, i.e. only a single complete congruential sequence is generated. Each number in it is presented once, and level of uniformity or repentances in this case is $U_T(w, N_t = 0) = 1$.

The previous result confirms the fact that the generation of 2^w numbers creates single complete congruential sequence having a momentary uniform distribution of random numbers $U_T(w, N_t = 0) = 1$. However, in complete twister there is a total amount of $N_T(w, N_t) = w \cdot N = w \cdot 2^w$ sequences. In this case the level of uniformity is equal to the amount of twisting sequences because each number in single complete sequence is presented once:

$$U_T(w, N_t) = N_T(w, N_t) = w \cdot N = w \cdot 2^w. \tag{6}$$

Let's take the benefits from the previous program *P030511* in which the array of counters having $2^{28} = 268{,}435{,}456$ elements was used. Because the quantity of twisters is equal to the amount of unique complete sequences, that means a uniformity of distribution is defined by the level of $U_T(w = 28, N_t) = N_T(w = 28, N_t) = w \cdot 2^w = 28 \cdot 2^{28} = 7{,}516{,}192{,}768$. However, this value is bigger than maximum for 32 bits number which is $2^{32} - 1 = 4{,}294{,}967{,}295$.

To define a permissible amount of bits w, it is required to take into consideration the maximal value of repentances for counter, i.e. solving the equation $w \cdot 2^w = 2^{32}$ is needed. By using binary logarithm it is obvious that for $w \leq 32$ this equation has no integer number solution. Thus, the testing of complete twister for the single pair of congruential constants a and c should be done for the numbers having $w = 27$ bit length. In this case the uniformity is $U_T(w = 27, N_t) = N_T(w = 27, N_t) = w \cdot 2^w = 27 \cdot 2^{27} = 3{,}623{,}876{,}656$ and it complies with a range of values $2^{31} < U_T(w = 27, N_t + 1) < 2^{32}$.

Below is the program code which performs the complete generation of twisting sequences for the random numbers having $w = 12$ bit length. Each random number has to be appeared the same amount of times as the twisting sequences because in each sequence it may be found once. Program names *P030512* and *cP030512* are chosen by chance.

```
using nsDeonYuliTwist32D;          // twister generator class
namespace P030512
{  class cP030512
   {  static void Main(string[] args)
      {  cDeonYuliTwist32D CT =
                                   new cDeonYuliTwist32D();
         CT.SetW ( 12 );               // set a number bit length
         CT.SetA ( 0.3, 0.3 );               // single constant a
         CT.SetC ( 0.2, 0.2 );               // single constant c
         CT.Start ();                         // generator starts
         uint w = CT.w;                     // number bit length
         uint N = CT.N1 + 1;                   // sequence length
         Console.WriteLine ( "w = {0}  N = {1}", w, N );
         uint[] cX = new uint[N];           // repeating counter
         for ( uint i = 0U; i < N; i++ ) cX[i] = 0U;
         int mStart = DateTime.Now.Minute;
         int sStart = DateTime.Now.Second;
         for ( uint nW = 0U; nW < w; nW++ )         // twisters
         {
//          Console.WriteLine ( "nW = {0}", nW );
            for ( uint nT = 0U; nT < N; nT++ )       // twisters
               for ( uint i = 0U; i < N; i++ )
               {  uint x = CT.Next ();         // random numbers
                  cX[x]++;                     // number x counter
               }
         }
         int sFinish = DateTime.Now.Second;
         int mFinish = DateTime.Now.Minute;
```

```
      Console.WriteLine(
                  "mStart  = {0,2}  sStart  = {1,2}",
                   mStart, sStart);
      Console.WriteLine (
                  "mFinish = {0,2}  sFinish = {1,2}",
                   mFinish, sFinish);
      uint nwN = w * N;                // quantity of twisters
      Console.WriteLine ( "nwN = {0}", nwN );
      uint countX = 0;
      for ( uint i = 0; i < N; i++ )
        if ( cX[i] == nwN ) countX++;          // one time
      Console.WriteLine ( "countX = {0}", countX );
      Console.ReadKey ();                   // result viewing
    }
  }
}
```

After execution the result appears as the following:

```
w = 12  N = 4096
mStart  =  8  sStart  =  1
mFinish =  8  sFinish = 16
nwN = 49152
countX = 4096
```

This listing suggests that length of random numbers is 12 bits; each complete sequence contains 4096 random numbers; duration of generation lasts $T_s(w = 12) = 15$ seconds; a total of 49152 complete twisting sequences are created, and that is maximum of unique sequences in which it is possible to generate for the single pair of constants a and c. Therefore, all 4096 numbers have the same amount of repentances with a level of distribution uniformity $U_T(w, w \cdot N_t) = w \cdot N = 12 \cdot 4096 = 49152$. And they cover uniformly the interval of random numbers $x \in [\overline{0: 2^w - 1}] = [\overline{0: 4095}]$.

Besides this, the next step is to estimate the time period required for generation of single number. Because each complete sequence contains $N = 2^w$ random numbers, that means a total amount $N_s(w, N)$ of them could be found as multiplication of all the numbers in single sequence by amount of twisting sequences:

$$N_s(w, N) = N \cdot N_T(w, w \cdot N_t) = N \cdot w \cdot N = 2^w \cdot w \cdot 2^w = w \cdot 2^{2w}. \quad (7)$$

For the previous example it is $N_s(w, N) = w \cdot 2^{2w} = 12 \cdot 2^{2 \cdot 12} = 12 \cdot 16{,}777{,}216 = 201{,}326{,}592$ of random values. So, the generation time for the single random number is:

$$t_1 = \frac{T_S(w)}{N_S(w, N)}. \quad (8)$$

Substituting into this formula the values from the previous example, the generation time can be estimated as $15/201{,}326{,}592 = 0.0000000745s = 0.0745\mu s$. In Table

1 there are results of generation of the random numbers having different bit length w.

w	$N = 2^w$	$N_s = w \cdot 2^{2w}$	T_s	$t_1 \cdot 10^{-6}$ (μs)
12	4,096	201,326,592	15”	0.0745
13	8,192	872,415,232	1’04”	0.0734
14	16,384	3,758,096,384	4’36”	0,0734
15	32768	16,106,127,360	19’42”	0,0734
16	65,536	68,719,476,736	1h24’04”	0,0734
17	131,072	292,057,776,128	5h57’17”	0,0734
18	264,144	1,263,950,581,248	25h13’12”	0,0734
19	524,288	5,222,680,231,936	106h29’05”	0,0734
20	1,048,576	21,990,232,555,520	448h21’23”	0,0734

Table 1: Generation time of complete twister for different bit length

The calculations considered above are directed to single pair of congruential constants a and c, but *nsDeonYuliTwist32D* generator allows tuning of the generation for different combinations of pairs a and c. For the complete sequences the constant a has to satisfy the condition $(a-1)\ mod\ 4 = 0$. Due to $a \in [\overline{1, N}] = [\overline{1\colon 2^w - 1}]$, the number of different values N_a is defined as:

$$N_a = \frac{N}{4} = \frac{2^w}{2^2} = 2^{w-2}. \tag{9}$$

When generating complete sequences the value of constant c has to be odd. Therefore, the total number of values for c is defined as:

$$N_c = \frac{N}{2} = \frac{2^w}{2^1} = w^{w-1}. \tag{10}$$

Thus, last two formulas allow defining the total number N_{ac} of pairs of a and c:

$$N_{ac} = N_a \cdot N_c = 2^{w-2} \cdot 2^{w-1} = 2^{2w-3}. \tag{11}$$

Taking into consideration the quantity of generations in all possible cases for sequences of twisters and congruential constants, the final estimation $N_T(w, N_{ac})$ of amount of created non-repeatable twisting complete sequences looks like:

$$N_T(w, N_{ac}) = N_T \cdot N_{ac} = w2^{2w} \cdot 2^{2w-3} = w \cdot 2^{4w-3}. \tag{12}$$

Because each sequence contains unique non-repeatable whole numbers it means the absolute level of uniform distribution coincides with a total amount of twisters:

$$U_T(w, N_{ac}) = N_T(w, N_{ac}) = w \cdot 2^{4w-3}. \tag{13}$$

At the same time, amount of random numbers $N_s(w, N_{ac})$ having w bit length is:

$$N_s(w, N_{ac}) = N \cdot N_T(w, N_{ac}) = 2^w \cdot w \cdot 2^{4w-3} = w \cdot 2^{5w-3}. \tag{14}$$

So, if random values with a length of 32 bits are created, the repeated generation of complete twisting sequence will occur after generation of $N_s(w = 32, N_{ac}) = 32 \cdot 2^{5 \cdot 32 - 3} = 2^5 \cdot 2^{157} = 2^{162}$ numbers. This will require plenty of time for the computers having generation time period for the single number as 0.0734 µs.

We have demonstrated that object twisting generator *nsDeonYuliTwist32D* is capable to create sufficiently long sequences of random numbers having absolute level of uniform distribution. More importantly, this generator does not use an additional array for storing of transitional congruential twisting sequences, and thus it takes the minimum possible memory of computer RAM.

6 Conclusion

Analysis of sources shows that algorithms of modern twisting generators are mainly based on the initial congruential array which is limited by the size. This kind of technology uses a bit shifting inside the array and resulted in the appearance of new twisting sequences. However, the problem is that the limited size of arrays may be unacceptable for some practical implementations. Increasing the length of sequences requires an enlarging of amount of elements in an array. It has been proven previously that only complete sequences satisfy the properties of absolute uniformity, but their longitude depends on a bit length of generated random numbers. If the bit length is very large it will limit the available memory to place the application program. To overcome this kind of limitation, in this paper we have proposed an algorithm in which no congruential twisting array is used. This solution allows generation of very large sequences. In this case they are restricted by permissible bit length of numbers to be processed directly by the commands of the computer processor. Presented results of testing confirm these statements and demonstrate the proved uniform distribution of generated numbers. The using of an automatic tuning of congruential parameters allows a controlled level of repetition for generating uniform distribution. In general, all the aspects disclosed herein may be used for a large number of application tasks.

Ethic, Contribution, Funding and Acknowledgments

This article is original and contains unpublished material. The authors equally contributed in this work, and they have no support or funding to report. The authors are thankful to Matthew Vandenberg, Robert Weingold, Walter Harrington, Jacqueline Nolan and Julia Alex Watts (University of Arkansas for Medical Sciences, Little Rock, USA) for the proofreading.

References

[Applebaum 2012] Applebaum B. (2012). Pseudorandom generators with long stretch and low locality from random local one-way functions. STOC '12. Proceedings of the 44th annual ACM symposium on theory of computing. May 19-22, New York, pp:805-816. DOI: 10.1145/2213977.2214050

[Bos et al. 2011] Bos J.W., T. Kleinjung, A.K. Lenstra, and P.L. Montgomery. (2011). Efficient SIMD Arithmetic Modulo a Mersenne Number. ARITH '11. Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic. July 25-27, Tubingen, pp:213-221. DOI:

10.1109/ARITH.2011.37

[Cai et al. 2016] Cai C., K.A. Carey, D.A. Nedosekin, Y.A. Menyaev, and M. Sarimollaoglu, et al. (2016). In Vivo Photoacoustic Flow Cytometry for Early Malaria Diagnosis. Cytometry A, 89A:531-542. DOI: 10.1002/cyto.a.22854

[Cai et al. 2016] Cai C., D.A. Nedosekin, Y.A. Menyaev, M. Sarimollaoglu, and M.A. Proskurnin, et al. (2016). Photoacoustic Flow Cytometry for Single Sickle Cell Detection In Vitro and In Vivo. Anal. Cell. Pathol., 2642361:1-11. DOI: 10.1155/2016/2642361

[Carey et al. 2016] Carey K.A., Y.A. Menyaev, C. Cai, J.S. Stumhofer, and D.A. Nedosekin, et al. (2016). Bioinspired hemozoin nanocrystals as high contrast photoacoustic agents for ultrasensitive malaria diagnosis. J. Nanomed. Nanotechnol., 7(3):49. DOI: 10.4172/2157-7439.C1.031

[Chapman et al. 2015] Chapman K.R., J.G. Burdon, E. Piitulainen, R.A. Sandhaus, and N. Seersholm, et al. (2015). Intravenous augmentation treatment and lung density in severe α1 antitrypsin deficiency (RAPID): a randomised, double-blind, placebo-controlled trial. Lancet, 386(9991):360-368. DOI: 10.1016/S0140-6736(15)60860-1

[Claessen and Palka 2013] Claessen K. and M.H. Palka. (2013). Splittable pseudorandom number generators using cryptographic hashing. Haskell '13. Proceedings of the 2013 ACM SIGPLAN symposium on Haskell. Sept. 23-24, Boston MA, pp:47-58. DOI: 10.1145/2503778.2503784

[Deon and Menyaev 2016] Deon A. and Y. Menyaev. (2016). The Complete Set Simulation of Stochastic Sequences without Repeated and Skipped Elements. J. Univers. Comput. Sci., 22(8):1023-1047.

[Deon and Menyaev 2016] Deon A. and Y. Menyaev. (2016). Parametrical Tuning of Twisting Generators. J. Comput. Sci., 12(8):363-378. DOI: 10.3844/jcssp.2016.363.378

[Eichenauer-Herrmann and Niederreiter 1994] Eichenauer-Herrmann J. and H. Niederreiter. (1994). Digital inversive pseudorandom numbers. ACM TOMACS, 4(4):339-349. DOI: 10.1145/200883.200896

[Gopalan et al. 2011] Gopalan P., R. Meka, O. Reingold, and D. Zuckerman. (2011). Pseudorandom generators for combinatorial shapes. STOC '11. Proceedings of the 43rd annual ACM symposium on theory of computing. June 6-8, San Jose CA, pp:253-262. DOI: 10.1145/1993636.1993671

[Hellekalek 1995] Hellekalek P. (1995). Inversive pseudorandom number generators: concepts, results and links. WSC '95. Proceedings of the 27th conference on winter simulation. Dec. 3-6, Washington DC, pp:255-262. DOI: 10.1145/224401.224612

[Juratly et al. 2015] 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(9):e0137613. DOI: 10.1371/journal.pone.0137613.

[Juratly et al. 2016] 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(5):e0156269. DOI: 10.1371/journal.pone.0156269

[Langdon 2009] Langdon W.B. (2009). A fast high quality pseudo random number generator for nVidia CUDA. GECCO '09. Proceedings of the 11th annual conference on genetic and evolutionary computation. July 8-12, Quebec, pp:2511-2514. DOI: 10.1145/1570256.1570353

[Leva 1992] Leva J.L. (1992). A fast normal random number generator. ACM TOMS, 18(4):449-453. DOI: 10.1145/138351.138364

[Lewko and Waters 2009] Lewko A.B., and B. Waters. (2009). Efficient pseudorandom functions from the decisional linear assumption and weaker variants. CCS '09. Proceedings of the 16th ACM conference on computer and communications security. Nov. 9-13, Chicago IL, pp:112-120. DOI: 10.1145/1653662.1653677

[Li 2010] Li M. (2010). Generation of teletraffic of generalized Cauchy type. Phys. Scr., 81(2):025007. DOI: 10.1088/0031-8949/81/02/025007

[Li 2017] Li M. (2017). Record length requirement of long-range dependent teletraffic. Physica A, 472:164-187. DOI: 10.1016/j.physa.2016.12.069

[Mandal et al. 2016] Mandal K., X. Fan, and G. Gong. (2016). Design and Implementation of Warbler Family of Lightweight Pseudorandom Number Generators for Smart Devices. ACM TECS, 15(1): Article No.1. DOI: 10.1145/2808230

[Matsumoto and Nishimura 1998] Matsumoto M. and T. Nishimura, (1998). Mersenne twister: a 623-dimensionnally equidistributed uniform pseudorandom number generator. ACM TOMACS, 8(1):3-30. DOI: 10.1145/272991.272995

[Matsumoto et al. 2006] Matsumoto M., M. Saito, H. Haramoto, and T. Nishimura. (2006). Pseudorandom Number Generation: Impossibility and Compromise. J. Univers. Comput. Sci., 12(6):672-690. DOI: 10.3217/jucs-012-06-0672

[Matsumoto et al. 2007] Matsumoto M., I. Wada, A. Kuramoto, and H. Ashihara. (2007). Common defects in initialization of pseudorandom number generators. ACM TOMACS, 17(4): Article No.15. DOI: 10.1145/1276927.1276928

[Meka and Zuckerman 2010] Meka R. and D. Zuckerman. Pseudorandom generators for polynomial threshold functions. STOC '10. Proceedings of the 42nd ACM symposium on theory of computing. June 5-8, Cambridge MA, pp:427-436. DOI: 10.1145/1806689.1806749

[Menyaev and Zharov 2005] Menyaev Y.A. and V.P. Zharov. (2005). Phototherapeutic technologies for oncology. Proceedings of SPIE, 5973:271-278. DOI: 10.1117/12.640217

[Menyaev and Zharov 2006] Menyaev, Y.A. and V.P. Zharov, (2006). Experience in Development of Therapeutic Photomatrix Equipment. Biomedical Engineering, 40(2):57-63. DOI: 10.1007/s10527-006-0042-6

[Menyaev and Zharov 2006] Menyaev, Y.A. and V.P. Zharov, (2006). Experience in the Use of Therapeutic Photomatrix Equipment. Biomedical Engineering, 40(3):144-147. DOI: 10.1007/s10527-006-0064-0

[Menyaev and Zharova 2006] Menyaev, Y.A. and I.Z. Zharova. (2006). A technique for surgical treatment of infected wounds based on photodynamic and ultrasound therapy. Biomedical Engineering, 40(6):284-290. DOI: 10.1007/s10527-006-0102-y

[Menyaev et al. 2013] 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(12):3030-41. DOI: 10.1364/BOE.4.003030

[Menyaev et al. 2016] 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(9):3643-58. DOI: 10.1364/BOE.7.003643

[Niederreiter 1992] Niederreiter H. (1992). New methods for pseudorandom numbers and pseudorandom vector generation. WSC '92. Proceedings of the 24th conference on winter simulation. Dec. 13-16, Arlington WV, pp:264-269. DOI: 10.1145/167293.167348

[Saito and Matsumoto 2008] 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 Berlin

Heidelberg, pp:607-622. ISBN: 978-3-540-74496-2. DOI:10.1007/978-3-540-74496-2_36

[Sarimollaoglu et al. 2014] 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):1-11. DOI: 10.1016/j.pacs.2013.11.002

[Shamir 1983] Shamir A. (1983). On the generation of cryptographically strong pseudorandom sequences. ACM TOCS, 1(1):38-44. DOI: 10.1145/357353.357357

[Sussman et al. 2006] Sussman M., W. Crutchfield, and M. Papakipos. (2006). Pseudorandom number generation on the GPU. GH '06. Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware. Sept. 3-4, Vienna, pp:87-94. DOI: 10.1145/1283900.1283914

[Tong et al. 2014] Tong Y., J. Sun, S.S. Chow, and P. Li. (2014). Cloud-assisted mobile-access of health data with privacy and auditability. IEEE J. Biomed. Health. Inform., 18(2):419-429. DOI: 10.1109/JBHI.2013.2294932

[White et al. 2008] White D.R., J. Clark, J. Jacob, and S.M. Poulding. (2008). Searching for resource-efficient programs: low-power pseudorandom number generators. GECCO '08. Proceedings of the 10th annual conference on genetic and evolutionary computation. July 12-16, Atlanta GA, pp:1775-1782. DOI: 10.1145/1389095.1389437

[Zharov et al. 2001] Zharov V.P., Y.A. Menyaev, Y.Y. Gorchak, K.V. Utkina, and Y.A. Menyaev. (2001). Methods for photoultrasonic treatment of festering wounds in oncological patients. Crit. Rev. Biomed. Eng., 29(1):111-124. DOI: 10.1615/CritRevBiomedEng.v29.i1.50

[Zharov et al. 2001] Zharov V.P., Y.A. Menyaev, R.K. Kabisov, S.V. Al'kov, and A.V. Nesterov et al. (2001). Design and application of low-frequency ultrasound and its combination with laser radiation in surgery and therapy. Crit. Rev. Biomed. Eng., 29(3):502-519. DOI: 10.1615/CritRevBiomedEng.v29.i3.130