Uniform Twister Plane Generator

Aleksei F. Deon and Yulian A. Menyaev

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

Abstract: Random plane generators may use various types of the random number algorithms to create multidimensional planes. At the same time, the discrete Descartes random planes have to be uniform. The matter is that using the concept of the uncontrolled random generation may lead to a result of weak quality due to initial sequences having either insufficient uniformity or skipping of the random numbers. This article offers a new approach for creating the absolute twisting uniform two-dimensional Descartes planes based on a model of complete twisting sequences of uniform random variables without repetitions or skipping. The simulation analyses confirm that the resulted random planes have an absolute uniformity. Moreover, combining the parameters of the original complete uniform sequences allows a significant increase in the number of created planes without using additional random access memory.

Keywords: Pseudorandom Number Generator, Stochastic Sequences, Congruential Numbers, Twister Generator, Random Plane, Random Field

Introduction

In our previous studies (Deon and Menyaev, 2016a; 2016b; 2017) there were proposed several pseudorandom number generators, particularly $nsDeonYuliTwist32D$, which offers a technique of using no congruential twisting array. This generator allows the creation of absolutely complete twister uniform sequences having various lengths.

The direction of Random Plane (RP) Generators (RPG) employs a stochastic process at the time of creating the points distributed on N-dimensional plane. Here we consider a two-Dimensional (2D) plane only. Other discrete-dimensional planes have the same initial properties. Each coordinate of RP-generated points may belong to its own Random Field (RF). An analysis of the last sources sums up the following selected types of random fields: Conditional RF (Quattioni et al., 2004; Sutton and McCallum, 2012), Markov RF (Sarawagi and Cohen, 2004; Bekkerman et al., 2006), Gaussian RF (Rimstad and Omre, 2014), uniformed RF (Xiao, 2010) and others (Qi et al., 2004; Dachian and Nahapetian, 2009). In the application areas the RPGs are often applied in graphical images (Kumar and Hebert, 2003), phone systems (Sung and Jurafsky, 2009), advertising applications, etc. Next, the RPGs are actively used in fundamental studies, starting from 2D theoretical modeling (Gnedenko, 1998; Feller, 2008), Monte Carlo plane simulation (Newman and Barkema, 1996; Spanos and Zeldin, 1998), factorial development (Kim and Zabih, 2002), realizations for training systems (Sha and Pereira, 2003), etc. and biomedical engineering (Menyaev and Zharov, 2005; 2006a; 2006b; Menyaev et al., 2013; 2016; Koonce et al., 2017).

The principles of all these studies are based on the conception of random planes, in which the Descartes plane features have to satisfy the following properties: (1) The generation process has to provide the uniqueness (i.e., no repetitions) of each point on the plane and (2) the generation process has to keep the completeness (i.e., without skipping) for all created points. These properties should be considered as a ‘natural filter’ for choosing the random number generator.

Let’s consider two of them in brief. If the generation uses the twister generator MT19937 (Matsumoto and Nishimura, 1998; Matsumoto et al., 2006; 2007; Saito and Matsumoto, 2008), then the result of this attempt is very discussable since this generator in DieHard Tests (Berger and Zorn, 2006; Novark and Berger, 2010; Alani, 2010) demonstrates a uniqueness level of 0.7, which is equivalent to the level of repeatability $1-0.7 = 0.3$. On the other hand, we may use the twister generator $nsDeonYuliTwist32D$ (Deon and Menyaev, 2017), which is guaranteed to create the complete uniform twisting sequences of an arbitrary size having no repetitions and skipping of elements. Now the question here is: Would it be possible to observe the Descartes properties in the current particular task?
Let’s consider this issue more in detail. For this, let us use the aforementioned twister generator, which is capable of creating complete uniform twisting sequences of arbitrary size. Below is the program code for modeling a grid of the discrete Descartes plane \( U \times V \). The program algorithm generates the integer random numbers independently along the \( U \) and \( V \) independent axes. Matrix \( A \) is the discrete plane indicator; its cells \( a(u, v) \in A \) with indices \( u \) and \( v \) correspond to the coordinates of discrete points \( <u, v> \in U \times V \). The value of cell \( a(u, v) \in A \) indicates the quantity of attempts to create independently the corresponding point \( <u, v> \) on the generated plane. Without loss of generality and in order to visualize the result, we assign the amount of discrete coordinates by given sets \( U = V = \{0, 1, 2, 3, 4, 5, 6, 7\} \), which in the binary representation corresponds to the length \( w = 3 \) bits for each coordinate. According to the previous studies (Deon and Menyaev, 2016b; 2017), let’s choose a twister generator \( nsDeonYuliTwist32D \), which operates on the basis of congruential model \( x_{t+1} = (ax_t + c) \mod W \) with constants \( a = 5, c = 1 \). The initial values of the twisting sequences are taken here as \( x_0 = 1 \in U \) and \( x_0 = 4 \in V \). Any other choice of parameters for generation is possible; the essence of the obtained results will not be changed. Program names \( P040101 \) and \( cP040101 \) are taken by chance. The chosen programming language is C# available in Microsoft Visual Studio. The use of other dialects of the older C versions (i.e., Win32) or C++ available in Microsoft Visual Studio. Without loss of generality and in order to visualize the result, we assign the amount of discrete coordinates by given sets \( U = V = \{0, 1, 2, 3, 4, 5, 6, 7\} \), which in the binary representation corresponds to the length \( w = 3 \) bits for each coordinate. According to the previous studies (Deon and Menyaev, 2016b; 2017), let’s choose a twister generator \( nsDeonYuliTwist32D \), which operates on the basis of congruential model \( x_{t+1} = (ax_t + c) \mod W \) with constants \( a = 5, c = 1 \). The initial values of the twisting sequences are taken here as \( x_0 = 1 \in U \) and \( x_0 = 4 \in V \). Any other choice of parameters for generation is possible; the essence of the obtained results will not be changed. Program names \( P040101 \) and \( cP040101 \) are taken by chance. The chosen programming language is C# available in Microsoft Visual Studio. The use of other dialects of the older C versions (i.e., Win32) or C++ available in Microsoft Visual Studio.
In matrix $A$ the cells with values that differed from 1, show that the independent generation of coordinates of the points on the plane doesn’t ensure uniform distribution of random points $<u, v>$. Some points are missed (values 0) and others are present several times (values > 1).

So, the aim of this article is to find a solution for the generation of uniform discrete twisting planes, which possess the Descartes property of a single presence of the random points in nodes of the discretization grid.

Theory

One of the options to represent the discrete Descartes plane is an enumeration of all points in the grid nodes, formed from the values of discretization on the corresponding axes. If the location of axes is independent, the grid has a rectangular view. Moreover, if discretization for both axes is the same and uniform, the grid has a square view.

Let’s assume that the square grid includes $N$ points of discretization along each axis. Thus, the total number of grid points is $N \times N = N^2$. To set these points in a random way, an algorithm is required that can provide a random move from one point to another.

Now would be a good time to point out and emphasize the following: The plane is random only if moving from one point to another, while creating the plane, utilizes the stochastic process.

In this case, the requirement of Descartes axes, which prescribes a unique representation of each point, has to be kept. The unambiguity is determined by the discretization of the axes. Uniqueness is provided by an appropriate procedure, which does not allow entering each point of the grid twice or more times. The skipping of vertices of the grid isn’t allowed either. In other words, each point is presented once during generation of all the grid points. In this case, the total enumeration of the points is $N^2$. Following this way, such a grid on the Descartes plane is called uniform and a random Descartes plane, which contains uniform grid, is called uniform Descartes RP.

There are many ways to specify the points on the grid. Let’s name a few of them:

- Rectangular left or right filling of the grid, when one of the axes is selected and at each location of discretization of this axis, the points along the discrete points of the other axis are placed on the grid
- Rectangular top or bottom filling of the grid under the same conditions
- Diagonal filling of the grid under the same conditions
- The secondary indexing of the discrete points along the Descartes axes

This is not a whole list of possible techniques. The options to choose aren’t limited and can be organized by the designer in any possible way. Note that items (1) – (3) create ordinary Descartes planes and item (4) allows obtaining the random Descartes planes, if the secondary index is a result of the stochastic process.

In this article, an option of secondary indexing of the discrete marks on the Descartes axes is adopted. Let’s demonstrate this by an example, in which the congruential generation of random numbers $x_{i+1} = (ax_i + c) \& \text{maskW}$ is used as secondary indexing base. In order to visualize the results, we take the complete uniform sequences of random numbers $x \in \{0, 1, 2, 3\} = \{00, 01, 10, 11\}$ having length $w = 2$ bits. In this case, each complete sequence contains $N = 2^w = 2^2 = 4$ elements. Without loss of generality, let’s assume that $a = 1 \in [L, N-1]$ and $c = 3 \in [L, N-1]$. In total, four congruential sequences are possible: $<0, 3, 2, 1>$, $<1, 0, 3, 2>$, $<2, 1, 0, 3>$, $<3, 2, 1, 0>$. These sequences allow creating various random tracks on uniform Descartes RP.

If random value $x_0 = 1$ is chosen as an initial value, then the designated generator $GU$ creates the sequence $U = <1, 0, 3, 2>$. From this it follows that the initial random vertex will be located on the vertical part of the grid with horizontal discrete mark 1 along the $U$ axis (Fig. 1).

If the second independent generator, which is designated as $GV$, uses the initial random value $x_0 = 3$, then sequence $V = <3, 2, 1, 0>$ is created. From this it’s obvious that the second coordinate has the value of 1 for the initial random point $<1, 3>$. The next vertex has coordinates $<0, 2>$. Both obtained vertices are connected by an arc, forming the beginning of the random track. Then, vertex $<3, 1>$ will be placed on this track. Finally, the vertex with coordinates $<2, 0>$ completes the random track. For clarity, the visual representation of this track is shown in Fig. 1.

Regarding the random sequence of secondary indices $<1, 0, 3, 2>$ along the $U$ axis, four sequences along the $V$ axis are possible:

1) $U = <1, 0, 3, 2>$
   $V = <3, 2, 1, 0>$
2) $U = <1, 0, 3, 2>$
   $V = <2, 1, 0, 3>$
3) $U = <1, 0, 3, 2>$
   $V = <1, 0, 3, 2>$
4) $U = <1, 0, 3, 2>$
   $V = <0, 3, 2, 1>$

These sequences $V$ can be interpreted as the left circular shift (Deon and Menyaev, 2016b) of the original sequence $<3, 2, 1, 0>$. Figure 2 shows four tracks of an interaction of the pairs of sequences on the $U$ and $V$ axes. These tracks provide an exact one-time generation of each vertex on a grid of the Descartes RP.
The program code for this task is presented below, in which all the vertices are generated on a grid of the Descartes RP. Random numbers have the length of \( w = 3 \) bits. In each sequences \( U \) and \( V \) there are \( N = 2^w = 2^3 = 8 \) random numbers with \( x \in [0,N-1] = \{0,1,2,3,4,5,6,7\} \). Both sequences \( U \) and \( V \) are given by a congruential model \( x_{i+1} = (ax_i + c) \mod W \) with coefficients \( a = 5 \) and \( c = 1 \). The first sequence begins with value \( x_0 = 1 \) and has the form \( U = <1, 6, 7, 4, 5, 2, 3, 0> \). The second sequence begins with value \( x_0 = 4 \) and has the form \( V = <4, 5, 2, 3, 0, 1, 6, 7> \). Each cell of matrix \( A \) corresponds to one vertex on a grid of RP. The value of cell \( a \in A \) shows how many times the corresponding vertex is generated. Program names \( P040201 \) and \( cP040201 \) are selected by chance.

```csharp
namespace P040201
{
    class cP040201
    {
        static void Main(string[] args)
        {
            uint w = 3;                                // number bit length
            int N = 1 << (int)w;     // U and V sequences length
            Console.WriteLine("w = {0}  N = {1}", w, N);
            uint[] U = new uint[8] {1,6,7,4,5,2,3,0};
            uint[] V = new uint[8] {4,5,2,3,0,1,6,7};
            int[,] A = new int[N, N];                  // result matrix
            for (int i = 0; i < N; i++)
            {
                Console.Write("i = {0,3} |   ", i);
                for (int j = 0; j < N; j++) A[i, j] = 0;
                Console.Write("U =");
                for (int m = 0; m < N; m++)
                {
                    Console.Write("{0,4}", U[m]);
                }
                Console.WriteLine();
                Console.Write("        |   V =");
                for (int m = 0; m < N; m++)
                {
                    Console.Write("{0,4}", V[m]);
                }
                Console.WriteLine();
            }
            uint r = V[0];                          // V shift beginning
            for (int m = 1; m < N; m++) V[m - 1] = V[m];
            V[N-1] = r;
            Console.WriteLine("Matrix A");
            for (int i = 0; i < N; i++)
            {
                for (int j = 0; j < N; j++)
                {
                    Console.Write("{0,4}", A[i, j]);
                }
                Console.WriteLine();
            }
            Console.ReadKey();                        // result viewing
        }
    }
}
```

After this code execution the listing below appears:
Matrix A
1 1 1 1 1 1 1 1 
1 1 1 1 1 1 1 1 
1 1 1 1 1 1 1 1 
1 1 1 1 1 1 1 1 
1 1 1 1 1 1 1 1 
1 1 1 1 1 1 1 1 
1 1 1 1 1 1 1 1 
1 1 1 1 1 1 1 1 

The values of 1 in the cells of matrix A show that each cell was updated once. This result reflects the single initialization of the corresponding vertices on a grid of uniform Descartes RP.

Similar tests of creating uniform planes with an arbitrary bit length \( w \) of the random numbers confirm a uniformity of the received RPs. Replacement of the congruential sequences by the different corresponding twisting sequences shows a similar result when generating the twisting RPs. Thus, the complete uniform twisting sequences can ensure a creation of the complete uniform twisting RPs. Now we may proceed to the generator designing.

**Construction and Results**

Program \( P040201 \), described in the previous section, utilizes two arrays \( U \) and \( V \) as initial congruential sequences. Such arrays can be created using the twister generator \( \text{nsDeonYuliTwist28DA} \) (Deon and Menyaev, 2016b). However, this solution will not be perfect for the case of large planes because it requires a lot of available Random Access Memory (RAM). For many reasons it may be unavailable on a computer. This limitation can be overcome by using the twister generator \( \text{nsDeonYuliTwist32D} \) (Deon and Menyaev, 2017), which doesn’t use arrays of the twisting sequences. However, a more satisfactory solution is the aforementioned program \( P040201 \), which implies the repeated generation of \( U \) and \( V \), whereas \( \text{nsDeonYuliTwist32D} \) doesn’t provide this. Thus, we come to the inference that before performing a generation of the twisting planes, it is necessary to have tools for the trivial operations with the twisting sequences.

**Simple Twister Generator**

Class \( \text{cDeonYuliSTwist32D} \), which is presented below, includes in its name a letter \( S \) indicating the meaning of the word ‘simple’. This class provides elementary operations with the twisting sequences and it does not use any arrays. The prototype of class \( \text{cDeonYuliSTwist32D} \) is class \( \text{cDeonYuliTwist32D} \). They are different in the issue that automatic setting of the congruential parameters \( a \) and \( c \) is excluded in class \( \text{cDeonYuliSTwist32D} \). An example of using this class is presented in this section later, in the description of the program code \( P040301 \).
//=======================================
public void SetW(uint sw)
{  w = (uint)Math.Abs(sw);         // number bit length
  xR = (a * xL + c) & maskW;  // end of xL,xR pair
  return xG;                   // twister of pair
}

public void RepeatTwist()
{  xL = xB;                       // for twister beginning
  xR = (a * xL + c) & maskW;  // end of xL,xR pair
}

public void ShiftTwist()
{  xL = xB;                       // for twister beginning
  xR = (a * xL + c) & maskW;  // end of xL,xR pair
}

public void Start()
{  N1 = 0xFFFFFFFF >> (32- (int)w);//max-number
    maskT = maskU;                // first twister bit
  maskU = 1U << ((int)w - 1);         // elder bit mask
  DeonYuli_SetC();                // c-value
  DeonYuli_PlusA();               // a-value
  x0 &= maskW;                   // sequence beginning
  StartCong(0);                  // for congruential generation
}

public void TimeStart()
{  x0 = (uint)DateTime.Now.Millisecond;// millisecs
  x0 = (uint)(N1 * xd);            // sequence beginning
  xL = xB;                        // for twister beginning
  xR = (a * xL + c) & maskW;  // end of xL,xR pair
}

public void SetX0(int sx)
{  x0 = (uint)sx;                        // sequence beginning
  x0 = (uint)(N1 * xd);            // sequence beginning
  xL = xB;                        // for twister beginning
  xR = (a * xL + c) & maskW;  // end of xL,xR pair
}

public void SetW(int sw)
{  w = (uint)Sw;    // for congenerator
  DeonYuli_SetW();
    w = (uint)Math.Abs(sw); // number bit length
  DeonYuli_SetW();
}

public void DeonYuli_SetW()
{  if (w >= N1 - 1) a -= 4U;    // on left from top edge
    if (a < z) a += 4U;      // on right from bottom edge
    a++;                               // true value for constant a
    if (a % 4U != 0U) a--;           // random condition
    for (uint i = 0U; i < 3U; i++)
        DeonYuliSTwist32D ST =
        DeonYuliSTwist32D ST =
        for (int j = 0; j < 3; j++)
            ST.xL = ST.xR = ST.xB = ST.xA = ST.xC = ST.xD = 0;
    ST.xL = a;                          // for twister beginning
    ST.xR = (a * ST.xL + c) & maskW;  // end of xL,xR pair
}

public void SetA(uint sa)
{  c = (uint)Math.Abs(sc);
  DeonYuli_SetC();                // c-value
}

public void SetC(uint sc)
{  if (c % 2U == 0U) c += 1;                  // only odd c
    c = (uint)(N1 * cd);                // related set of c
  if (c > N1) c = N1;                              // max-value
}

public void SetA(double sa)
{  a = (uint)Math.Abs(sa);
    a = (uint)(N1 * ad);                // related set of a
}

public void DeonYuli_PlusA()
{  if (a < 1U) {a = 1U; return; }
    if (ad > 1.0) ad = 1.0;
    c = (uint)(N1 * cd);
}

public void SetC(double sc)
{  if (c < 1) a = 1;                                     // min-value
    a = (uint)Math.Abs(sa);
    if (a < 1) {a = 1U; return; }
    if (ad > 1.0) ad = 1.0;
    c = (uint)(N1 * cd);
    c = (uint)Math.Abs(sc);
}

public void SetX0(double sx)
{  x0 = (uint)sx;                        // sequence beginning
    x0 = (uint)(N1 * xd);            // sequence beginning
  xL = xB;                        // for twister beginning
  xR = (a * xL + c) & maskW;  // end of xL,xR pair
}

if (a < N1) a = N1;                   // max-value

void DeonYuli_PlusA()
{  if (a < 1U) {a = 1U; return; }
    a = (uint)Math.Abs(sa);
    if (a < 1) {a = 1U; return; }
    if (ad > 1.0) ad = 1.0;
    c = (uint)(N1 * cd);
    a++;                               // true value for constant a
    if (a < z) a += 4U;      // on right from bottom edge
    if (a >= N1 - 1) a -= 4U;  // on left from top edge
}

public void DeonYuliSTwist32D()
{  double c = Math.Abs(sc);
    if (c < 1) a = 1;                                     // min-value
    if (a < N1) a = N1;                              // max-value
}

public void DeonYuliSTwist32D()
{  double c = Math.Abs(sc);
    if (c < 1) a = 1;                                     // min-value
    if (c > N1) c = N1;                              // max-value
}

To verify the correct utilization of the presented generator nsDeonYuliSTwist32D, let’s use the program code P040301, in which the sequences of all twisters of the random numbers having length $w = 3$ bits are generated. The total quantity of sequences is $w \cdot N = w \cdot 2^w = 3 \cdot 2^3 = 24$. Program names P040301 and cP040301 are taken by chance.

using nsDeonYuliSTwist32D; // s-twister uni-generator namespace P040301
{  class cP040301
    static void Main(string[] args)
        cDeonYuliSTwist32D ST =
new cDeonYuliSTwist32D();
ST.SetW(3); // number bit length
ST.SetA(5); // congruential constant a
ST.SetC(1); // congruential constant c
ST.SetX0(1); // sequence beginning
ST.Start(); // generator starts
int m = (int)ST.w; // random bit length
int n = (int)ST.N1 + 1; // sequence length
Console.WriteLine("w = {0}  N = {1}", w, N);
int N = (int)ST.N1 + 1;                // sequence length
int w = (int)ST.w;                      // random bit length
ST.Start();                                      // generator starts
ST.SetX0(1);                          // sequence beginning
ST.SetC(1);                       // congruential constant c
ST.SetA(5);                       // congruential constant a
ST.SetW(3);                              // bit length
new cDeonYuliSTwist32D();  // generator 2
ST.ShiftCong();                      // congruential shift
ST.RepeatTwist();

After the execution of P040301, the following listing
appears on the monitor:

k =  0  |  Cong =      1  6  7  4  5  2  3  0
k =  1  |  Twist 1 =   3  5  7  1  2  4  6  0
k =  2  |  Twist 2 =   7  3  6  2  5  1  4  0
k =  3  |  Cong =      6  7  4  5  2  3  0  1
k =  4  |  Twist 1 =   5  7  1  2  4  6  0  3
k =  5  |  Twist 2 =   3  6  2  5  1  4  0  7
k =  6  |  Cong =      7  4  5  2  3  0  1  6
k =  7  |  Twist 1 =   7  1  2  4  6  0  3  5
k =  8  |  Twist 2 =   6  2  5  1  4  0  7  3
k =  9  |  Cong =      4  5  2  3  0  1  6  7
k = 10  |  Twist 1 =   1  2  4  6  0  3  5  7
k = 11  |  Twist 2 =   2  5  1  4  0  7  3  6
k = 12  |  Cong =      5  2  3  0  1  6  7  4
k = 13  |  Twist 1 =   2  4  6  0  3  5  7  1
k = 14  |  Twist 2 =   5  1  4  0  7  3  6  2
k = 15  |  Cong =      2  3  0  1  6  7  4  5
k = 16  |  Twist 1 =   4  6  0  3  5  7  1  2
k = 17  |  Twist 2 =   1  4  0  7  3  6  2  5
k = 18  |  Cong =      3  0  1  6  7  4  5  2
k = 19  |  Twist 1 =   6  0  3  5  7  1  2  4
k = 20  |  Twist 2 =   4  0  7  3  6  2  5  1
k = 21  |  Cong =      0  1  6  7  4  5  2  3
k = 22  |  Twist 1 =   0  3  5  7  1  2  4  6
k = 23  |  Twist 2 =   0  7  3  6  2  5  1  4

These congruential twisting sequences coincide with
the result of tests of our previously developed twister
generator nsDeonYuliSTwist28DA utilizing a technique of
the congruential twisting array.

Running the program P040301 with other values of
the bit length w of the random numbers confirms a
completeness of the generated sequences. This is
sufficient to ensure the development of programs for
generating the Descartes uniform twisting RPs.

Twister Generator of Uniform Planes

When constructing a complete generator of uniform
twisting planes, let’s use two uniform complete generators
GU and GV of the aforementioned instrumental class
cDeonYuliSTwist32D. It allows organizing the next class
cDeonYuliPlaneTwist32D for the generation of all points
on a grid of the twisting plane. By default, the initial track
takes a diagonal of the grid points from the left-bottom
position to the right-top one, although this may be
changed by setting the independent beginnings for the
internal generators GU and GV. An example use of class
cDeonYuliPlaneTwist32D is presented in this section later,
located in the description of the program code P040302.

using nsDeonYuliSTwist32D; // s-twister uni-generator
namespace nsDeonYuliPlaneTwist32D
{  class cDeonYuliPlaneTwist32D
{  public uint w = 16; // uniform bit length
public uint N1 = 0; // max-number in track
public uint a = 5U; // congruential constant a
public uint c = 1U; // congruential constant c
public uint x0 = 1U; // constant of track beginning
public uint wU = 1U; // constant of track beginning
public cDeonYuliSTwist32D GU; // generator 1
public cDeonYuliSTwist32D GV; // generator 2
public uint nRU = 0; // ring shift number in GU
public uint nRV = 0; // ring shift number in GV
public uint nG = 0; // elements in GU and GV tracks
public uint nRV = 0; // ring shift number in GV
public uint nRV = 0; // ring shift number in GV

public cDeonYuliPlaneTwist32D()
{  public uint w = 16; // uniform number bit length
public uint N1 = 0; // max-number in track
public uint a = 5U; // congruential constant a
public uint c = 1U; // congruential constant c
public uint x0 = 1U; // constant of track beginning

public cDeonYuliPlaneTwist32D()
{  public uint w = 16; // uniform number bit length
public uint N1 = 0; // max-number in track
public uint a = 5U; // congruential constant a
public uint c = 1U; // congruential constant c
public uint x0 = 1U; // constant of track beginning
}}
```csharp
{ w = (uint)sw; // random number bit length
 DeonYuli_SetWN1ACX();
}

//-----------------------------------------------
public void SetW(uint sw)
{ w = sw; // random number bit length
 DeonYuli_SetWN1ACX();
}

//-----------------------------------------------
void DeonYuli_SetWN1ACX()
{ N1 = 0xFFFFFFFF >> (32- (int)w); // max-number
 a = (uint)((double)N1*0.39); // congruential const a
 c = a / 2; // congruential const c
 x0 = N1 / 2U; // constant of track beginning
}

//-----------------------------------------------
void DeonYuli_SetWN1ACX()
{ GU.w = w; // random number bit length in GU
 GU.a = a; // congruential constant a for GU
 GU.c = c; // congruential constant c for GU
 GU.x0 = x0; // sequence beginning in GU
 GV.w = w; // random number bit length in GV
 GV.a = a; // congruential constant a for GV
 GV.c = c; // congruential constant c for GV
 GV.x0 = x0; // sequential beginning in GV
}

//-----------------------------------------------
public void Start()
{ DeonYuli_SetN1ACX(); // congruential constants
 GU.Start(); // GU generator starts
 GV.Start(); // GV generator starts
 a = GU.a; // congruential constant a
 c = GU.c; // congruential constant c
 x0 = GU.x0; // beginning of U and V sequences
 nWU = 0; // twister shift number in GU generator
 nRU = 0; // ring shift number in GU generator
 nWV = 0; // twister shift number in GV generator
 nRV = 0; // ring shift number in GV generator
 nG = 0; // elements number in GU and GV tracks
 GU.StartCong(0); // for congruential GU generation
 GV.StartCong(0); // for congruential GV generation
}

//-----------------------------------------------
public void Next(ref uint u, ref uint v)
{ if (nWV == 0) GU.NextCong();
   else GU.NextTwist();
 if (nWU == 0) GV.NextCong();
   else GV.NextTwist();
 u = GU.xG; // <u,v> point coordinates
 v = GV.xG;
 if (nG < N1) {nG++; return;} // inside track
 nG = 0; // regular track beginning
 if (DeonYuli_RingCongGV()) return; // inside GV
 if (DeonYuli_RingTwistGV()) return; // inside GV
 if (DeonYuli_RingCongGU()) return; // inside GU
 DeonYuli_RingTwistGU(); // inside GU
} //-------------------------------
bool DeonYuli_RingCongGV()
{ if (nWV != 0) return false; // no congruential ring
   if (nWU == 0) GU.RepeatCong();
   else GU.RepeatTwist();
 if (nRV < N1) // congruent. ring opportunity in GV
   GV.ShiftCong(); // congruent shift in GV
 nRV++; // next ring number
 return true; // inside congruential track in GV
}

nRV = 0; // first ring in GV
 nWV = 1; // first twister in GV
 GV.StartTwist(nWV); // twister starts in GV
 return true; // inside GV
}

//-------------------------------
bool DeonYuli_RingTwistGV()
{ if (nWV == 0) return false; // no twister ring
   if (nWU == 0) GU.RepeatCong();
   else GU.RepeatTwist();
 if (nRV < N1) // twister ring opportunity in GV
   GV.ShiftTwist(); // twister shift in GV
 nRV++; // next ring number
 return true; // inside twister regime in GV
}

nRV = 0; // new ring in GV
 if (nWV < w - 1) // continue twisters
   nWV++; // next bit twister in GV
 GV.StartTwist(nWV); // twister starts in GV
 return true; // inside twister regime in GV
}

nRV = 0; // new ring in GV
 nWV = 0; // congruent. beginning (twister 0) in GV
 GV.StartCong(nWV); // GV generator starts
 return false; // rings R2 and W2 are over in GV
}

//-------------------------------
bool DeonYuli_RingCongGU()
{ if (nWU != 0) return false; // no congruential ring
 if (nRU < N1) // congruent. ring opportunity in GU
   GV.ShiftCong(); // congruent shift in GU
 nRU++; // next ring number
 return true; // inside congruential track in GU
}

nRU = 0; // first ring in GU
 nWU = 1; // first twister in GU
 GV.StartTwist(nWU); // twister starts in GU
 return true; // inside GU
}

//-------------------------------
bool DeonYuli_RingTwistGU()
{ if (nWU == 0) return false; // no twister ring
 if (nRU < N1) // twister ring opportunity in GU
   GV.ShiftTwist(); // twister shift in GU
 nRU++; // next ring number
 return true; // inside GU
}
```

(DOI: 10.3844/jcssp.2018.260.272)

return true; // inside twist track in GU
}  
nRU = 0; // new ring in GU
if (nWU < w - 1) // continue twisters
{  nWU++; // next bit twister in GU
  GU.StartTwist(nWU); // twister starts in GU
  return true; // inside twister regime in GU
}  
nRU = 0; // new ring in GU
nWU = 0; // congruent. beginning (twister 0) in GU
GU.StartCong(nWU); // GU generator beginning
return false; // all planes created; common beginning

To test an operation of the presented generator nsDeonYuliPlaneTwist32D, let's use the program code P040302 shown below, in which the points of the initial twisting plane of the random numbers having length \( w \) = 3 bits are generated. By default, the initial track of generation is a diagonal of the grid points from the left-bottom position to the right-top one. The uniformity of points on a plane would be confirmed by matrix \( A \), in which the cell values are the counters of generation of the corresponding points \(<u,v>\) on RP. Program names P040302 and cP040302 are taken by chance.

```csharp
using nsDeonYuliPlaneTwist32D; // twist-plane generator namespace P040302
}
class cP040302
{
  static void Main(string[] args)
  {
    cDeonYuliPlaneTwist32D TP =
      new cDeonYuliPlaneTwist32D();
    int w = 3; // random number bit length
    int N = 1 << w; // track length
    TP.SetW(w);
    TP.Start(); // generator starts
    Console.WriteLine("w = {0}   N = {1}", w, N);
    Console.WriteLine("a = {0}   c = {1}   x0 = {2}",
                     TP.a, TP.c, TP.x0);
    uint[] u = new uint[N]; // point u-coords on track
    uint[] v = new uint[N]; // point v-coords on track
    int[,] A = new int[N, N]; // result matrix
    for (int i = 0; i < N; i++)
    {  for (int j = 0; j < N; j++)
      {  A[i, j] = 0;
        if (int j = 0; j < N; j++)
        {  TP.Next(ref uu, ref vv); // point on grid
          u[j] = uu;
          v[j] = vv;
          A[uu, vv]++;
        }  
        Console.Write("U = ");
        for (int j = 0; j < N; j++)
        {  Console.Write("{0,4}", u[j]);
        }  
        Console.WriteLine();
        Console.WriteLine("V = ");
        for (int j = 0; j < N; j++)
        {  Console.Write("{0,4}", v[j]);
        }  
        Console.WriteLine();
      }  
    }  
    Console.WriteLine("Matrix A");
    for (int i = 0; i < N; i++)
    {  for (int j = 0; j < N; j++)
      {  Console.Write("{0,4}", A[i, j]);
      }  
      Console.WriteLine();
    }  
    Console.ReadKey(); // result viewing
  }
}

After the execution of P040302 code, the listing below appears. To reduce the listing size, we skipped some strings, which are indicated by a dashed line.

```
```
twister on axis $U$ and indicator $nRU$ is pointed to the circular shift number on this axis $U$. The values $nWU = 0$ and $nRU = 0$ correspond to the congruential initial sequence on axis $U$. Similar values of indicators $nWV$ and $nRV$ show the circular shift of the initial sequence on axis $V$.

The listing of results for the singular matrix $A$ confirms that each random point $<u, v>$ is created once. The same results are valid for other $w \leq 32$. This corresponds to a concept of the Descartes uniform RP. Thus, generator $nsDeonYuliPlaneTwist32D$ is ready for implementation.

**Discussion**

Twister generator $nsDeonYuliPlaneTwist32D$ is capable creating a set of uniform twisting RPs for each pair of the congruential constants $a, c \in [1, N - 1] = [1, 2^n - 1]$ in the sequences of random numbers having $w$ bit length. All the initial values $x_v \in [0, 2^n - 1]$ are automatically presented in circles of the twisters (Deon and Menyaev, 2016b). However, the question is how many twisting planes could be obtained for each pair of parameters $a$ and $c$?

In the previous section, it has been determined that when creating the initial RP $U \times V$, two random sequences $U$ and $V$ can be taken, which are created by the corresponding twisting generators. It is known that twister 0 is the initial uniform congruential sequence. Let’s denote it as $U_0$. Another sequence $V_0$ could be chosen arbitrarily but on the condition that it is also uniform. If the sizes or quantity of elements in the initial sequences are the same $card(U_0) = card(V_0)$, then it can be stated that among all the possible tracks of the random uniform plane there has to be a track that creates points on the second diagonal (from the left-bottom position to the right-top one) of the corresponding square discrete grid. This is somewhat reminiscent of the idea of a central abstract element and specifically in our case this second diagonal track is the central track of the discrete grid. In generator $nsDeonYuliPlaneTwist32D$ this is exactly what is done, shown at the beginning of the listing of results for the previous program $P0040302$.

$U_0 = \begin{bmatrix} 3 & 0 & 1 & 6 & 7 & 4 & 5 & 2 \\ \end{bmatrix}$

$V_0 = \begin{bmatrix} 3 & 0 & 1 & 6 & 7 & 4 & 5 & 2 \\ \end{bmatrix}$

This is not the only solution because as the central track, one could take the main diagonal (from the left-top position to the right-bottom one) of a grid. But since the initial sequences $U$ and $V$ are random, the order of the vertex passage $<3, 3>, <0, 0>, <1, 1>, <6, 6>, <7, 7>, <4, 4>, <5, 5>, <2, 2>$ is also random.

Now let’s fix the sequence $U$ while the sequence $V$ is shifted to the left by the circular technique by one position.

$U_0 = \begin{bmatrix} 3 & 0 & 1 & 6 & 7 & 4 & 5 & 2 \\ \end{bmatrix}$

$V_1 = \begin{bmatrix} 0 & 1 & 6 & 7 & 4 & 5 & 2 & 3 \\ \end{bmatrix}$

This combination of pairs gives the points of the first track $<3, 0>, <0, 1>, <1, 6>, <6, 7>, <7, 4>, <4, 5>, <5, 2>, <2, 3>$ on RP. None of these points can appear on the reverse main diagonal of a grid, since sequences $U_0$ and $V_1$ are uniform. Their uniformity follows from the determined properties of the complete twisting sequences (Deon and Menyaev, 2016b). Shifting of sequence $V$, provided that sequence $U_0$ is fixed, could be continued and so the second track on a grid might be obtained.

$U_0 = \begin{bmatrix} 3 & 0 & 1 & 6 & 7 & 4 & 5 & 2 \\ \end{bmatrix}$

$V_2 = \begin{bmatrix} 1 & 6 & 7 & 4 & 5 & 2 & 3 & 0 \\ \end{bmatrix}$

The second track contains the points, which also cannot occur on track 1 and track 0 of the reverse diagonal and again that is because of the properties of the complete uniform twisting sequences. In this example, only 8 options to present sequence $V$ are possible since the $9^{th}$ shift repeats the initial variation $V_4 = V_0$.

So, the shift operations for sequences comply with the corresponding varieties of the twisting sequences. In the presented example, the shifts of the congruential sequence, which comply with the congruential generation from the corresponding initial values, are considered. Thus, the congruential initial generation of sequences $U_0$ and $V_0$ initiates the creation of one RP using the complete sequence of shifts of one of the original sequences $V_1$ while another sequence $U_0$ is fixed.

By analyzing the result of the previous program $P040301$, it’s easy to see that among all sequences $V_{k+1}$, there are all 8 congruential shifts $V_k = 0$, $V_k = 3$, $V_k = 6$, $V_k = 9$, $V_k = 12$, $V_k = 15$, $V_k = 18$, $V_k = 21$ in amount of $N = 2^n$ $= 2^3 = 8$ random numbers having length of 3 bits in the complete sequence. As a result, it turns out that tracks $<U_0, V_0>$ form the initial plane $L_0$.

Similar arguments apply to plane $L_1$. If we refer again to the result of the previous program $P040301$, this example shows that plane $L_1$ is created using the congruential twisting tracks $<U_k = 0, V_{k+1} = 1, 4, 7, 10, 13, 16, 19, 22>$. Plane $L_2$ is created by using shifts of the next twister $<U_k = 0, V_{k+1} = 5, 8, 11, 14, 17, 20, 23>$. Next is plane $L_3$, but to create it we need to perform the congruential circular shift of sequence $U_0$ by one random step to the left, which leads to obtaining the new distribution of the random values $U_k \in \mathbb{Z}$ along the axis $U$. Then, the next plane $L_3$, which can be obtained with the help of tracks $<U_k = 3, V_{k+1} = 0, 3, 6, 9, 12, 15, 18, 21>$. A summary of all the points obtained leads us to the conclusion that number $card(L)$ of the complete set of RPs is defined by multiplication of two things. The first is $card(U_0) = w \cdot N = \ L_2_w = 2^n$, which is a quantity of the different options of the congruential twisting forms of
sequence $U$; the second is $\text{card}(V) = w$, which is a quantity of the various twisters in sequence $V$ including the congruential twister 0, i.e.,: $\text{card}(L) = \text{card}(U(U_i))$, $\text{card}(V) = wN\cdot w = w^2N$.

Below is the program code P040101, in which the uniformity of all the random twisting planes $L(w = 3)$ is checked. Matrix $A$ is helpful for this task. Since in the uniform RPs each vertex is created once, after the full enumeration of all the RPs the quantity of generations of each vertex has to be equal to the amount of planes. In other words, the counters of cells of matrix $A$ have to have the same values and moreover, they have to be equal to the quantity of the generated RPs. To clarify, the program below uses the random numbers with a bit length $w = 3$. The names P040401 and cP040401 are selected by chance.

using nsDeonYuliPlaneTwist32D; //twist-plane generator namespace P040401 {
    class cP040401 {
        static void Main(string[] args) {
            cDeonYuliPlaneTwist32D TP =
                new cDeonYuliPlaneTwist32D();
            int w = 3;                     // random number bit length
            int N = 1 << w;                                  // track length
            TP.SetW(w);
            TP.Start();                                     // generator starts
            Console.WriteLine("w = {0}   N = {1}", w, N);
            Console.WriteLine("a = {0}   c = {1}   x0 = {2}",
                TP.a, TP.c, TP.x0);
            uint[] u = new uint[N];    // point u-coords on track
            uint[] v = new uint[N];    // point v-coords on track
            uint[,] A = new uint[N, N];
            for (int i = 0; i < N; i++)
                for (int j = 0; j < N; j++) A[i, j] = 0;
            uint uu = 0;           // point job coordinates on plane
            uint vv = 0;
            int plane = 0;                                   // plane number
            int k = 0;                                          // track number
            while (true) {
                uint nWU = TP.nWU, nRU = TP.nRU;
                uint nWV = TP.nRV, nRV = TP.nRV;
                for (int j = 0; j < N; j++)
                    for (int i = 0; i < N; i++)
                        if (k % N == 0) plane++;
                TP.Next(ref uu, ref vv);           // point on grid
                u[j] = uu;
                v[j] = vv;
                A[uu, vv]++;
            }
            if (k % N == 0) plane++;
            Console.WriteLine("plane = {0}", plane);
            k++;                                     // next track
            // if (k < 569) continue;
            Console.WriteLine("k = {0} ", k);
            Console.WriteLine("Matrix A");
            for (int i = 0; i < N; i++)
                for (int j = 0; j < N; j++)
                    Console.Write("{0,4}", A[i, j]);
            Console.WriteLine();
            if (k % N == 0) {
                Console.WriteLine("Matrix A");
                for (int i = 0; i < N; i++)
                    for (int j = 0; j < N; j++)
                        Console.Write("{0,4}", A[i, j]);
                Console.WriteLine();
            } Console.ReadKey();        // regular result viewing
        }
    }
}

After executing the program P040401, the following listing below appears on the monitor. The skipped strings are indicated by a dashed line.

$w = 3   N = 8$
$a = 5   c = 1   x0 = 3$
$plane = 1$
$k = 1   nWU = 0   nRU = 0   nWV = 0   nRV = 0$
$U = 3   0   1   6   7   4   5   2$
$V = 3   0   1   6   7   4   5   2$
$plane = 1$
$k = 2   nWU = 0   nRU = 0   nWV = 1   nRV = 1$
$U = 3   0   1   6   7   4   5   2$
$V = 0   1   6   7   4   5   2$
$- - - - -$
$plane = 72$

Matrix A

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

plane = 72
k = 575  nWU = 2  nRU = 7  nWV = 6  nRV = 6
U = 2 5 1 4 0 7 3 6
V = 6 2 5 1 4 0 7 3
plane = 72

k = 576  nWU = 2  nRU = 7  nWV = 7  nRV = 7
U = 2 5 1 4 0 7 3 6
V = 2 5 1 4 0 7 3 6

Matrix A

\[
\begin{array}{cccccccc}
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
72 & 72 & 72 & 72 & 72 & 72 & 72 & 72 \\
\end{array}
\]

So, the values of the elements of matrix \( A \) confirm the analytical calculations for \( \text{card}(L) = w^2N \) and for arbitrary \( w \leq 32 \). Generator \textit{nsDeonYuliPlaneTwist32D} creates a complete set of uniform twisting RPs, which is considered as the primary task of this article.

**Conclusion**

Analysis of the sources indicates that algorithms of the generators of uniform planes do not take into account the potential of the sequences having absolutely uniform distribution. Techniques of those generating algorithms do not guarantee the absolute uniformity of the complete random planes. To overcome this limitation, we proposed here the generators of the complete uniform sequences, which include unique twisting techniques described in our previous works. However, their direct application for the described task is hampered by the required properties of Descartes uniform planes. To satisfy this requirement, a new class \textit{nsDeonYuliStwist32D} was constructed and now with its help it is possible to create the dynamic objects of the simplest twisters without using congruential arrays. Applying secondary indexing technique allows for getting a generator of Descartes twisting random planes, which ensures the completeness and uniqueness of all the random variables on a grid of the Descartes plane. The performed tests confirm the absolute uniform distribution of the generated random values on a plane. In addition, a variety of the initial twisting sequences allows getting a set of the twisting planes for each pair of the congruential constants. In perspective, the obtained results can be used in a large number of applied tasks, which use the spatial plane distributions.

**Acknowledgment**

The authors are thankful to Matthew Vandenberg, Jacqueline Nolan, Julia Alex Watts and Walter Harrington (University of Arkansas for Medical Sciences, Little Rock, AR, USA) for the proofreading.

**Author’s Contributions**

Both authors equally contributed to this work.

**Funding Information**

The authors have no support or funding to report.

**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. No ethical issues were involved and the authors have no conflict of interest to disclose.

**References**


