algorithms

MDPI

Article

Phase Congruential White Noise Generator

Aleksei F. Deon [1], Oleg K. Karaduta [2] and Yulian A. Menyaev [3,*]

[1] Department of Information and Control Systems, Bauman Moscow State Technical University, 2nd Baumanskaya St., 5/1, 105005 Moscow, Russia; deonalex@mail.ru
[2] Biochemistry and Molecular Biology Department, University of Arkansas for Medical Sciences, 4301 W. Markham St., Little Rock, AR 72205, USA; OKKaraduta@uams.edu
[3] Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Sciences, 4301 W. Markham St., Little Rock, AR 72205, USA
* Correspondence: YAMenyaev@uams.edu

Abstract: White noise generators can use uniform random sequences as a basis. However, such a technology may lead to deficient results if the original sequences have insufficient uniformity or omissions of random variables. This article offers a new approach for creating a phase signal generator with an improved matrix of autocorrelation coefficients. As a result, the generated signals of the white noise process have absolutely uniform intensities at the eigen Fourier frequencies. The simulation results confirm that the received signals have an adequate approximation of uniform white noise.

Keywords: pseudorandom number generator; congruential stochastic sequences; white noise; stochastic Fourier spectrum

Citation: Deon, A.F.; Karaduta, O.K.; Menyaev, Y.A. Phase Congruential White Noise Generator. *Algorithms* **2021**, *14*, 118. https://doi.org/10.3390/a14040118

Academic Editors: Shankarachary Ragi and Edwin K. P. Chong

Received: 5 February 2021
Accepted: 2 April 2021
Published: 5 April 2021

1. Introduction

The concept of white noise realization corresponds to randomness in appearance and distribution of signals [1–4]. For audible signals, the conforming range is the band of frequencies from 20 to 20,000 Hz. The randomness of such signals in this range is usually perceived by the human ear as a hissing sound with different volume intensities. White noise also manifests itself in myriad natural phenomena. For example, noises of various intensities of the sea waves, waterfall, rain, wind, etc. Outside nature, in technical systems, white noise appears in p-n junctions of semiconductors, in roars of different engines (vehicles, aircrafts, etc.), in the overlapping of many sounds associated with big cities and metropolitan life (traffic, construction, honking, life support systems), and so on.

An analysis of the literature shows that most of the articles describe different applications of white noise, as well as its recognition in transmitted signals. This includes the areas such as theoretical and applied mathematics [5–9], physical research [3,4,10–13], electronic and radio engineering [14–23], acoustics and noise phenomena [1–4,24,25], computer algorithms [26,27], geological prospecting and exploration [28,29], medical and biological research [30–36], psychology and psychiatry [37,38], and others. It is worth noting that significant results have been achieved in those fields.

On the other hand, the literature describes the methods of white noise creation, as well as its use in the artificial modeling of various situations. A fundamental feature of this direction is the condition that a white noise generator is required initially. The first approach, in this case, uses techniques of recording the physical phenomena with subsequent digitizing and postprocessing filtration [39,40]. This method provides quasi "natural" or "realistic" white noise, however the numerical accuracy of the registration of stochastic quantities is often not sufficient. Moreover, the technical realization is quite expensive. The second approach in creating a white noise generator utilizes the techniques of computer-based algorithms. In this case, the generation accuracy is quite high. However, different algorithmic methods have varying degrees of generation quality and usually

they are associated with diverse disadvantages. To analyze the grade of generation, the verification methods are commonly applied.

As an example, in [41] the following form for white noise generation is proposed; it should be noted here that this is just one of many ways to implement a white noise generator. As a basis for forming n values c_j of white noise (provided that $j \in \overline{[0, n-1]}$) the function *rand*() from *Microsoft Visual Studio* could be utilized. The algorithmic expression used in [41] can be represented here in the following mathematical form:

$$c_j = 4\left(\frac{rand()}{randmax} - 0.5\right) \tag{1}$$

By using this expression, the resulting values can be obtained in the range $[-2 : +2]$. This simple formula allows creation of the elements in the required diapason. The next mandatory step is to check the condition of how much the obtained values c_j really are the elements of a process with white noise properties. For such verification it is necessary to consider how the white noise process and the properties of its elements are formed, and here it all depends on the quality of the function *rand*() and its maximum value of *randmax*. Let us consider this in more details.

From a general point of view, the numerical sequences form a stationary random process $\mathbb{P}$ if the characteristics of this process are not changing during its implementation. One such characteristic is the consistent observation of the presence of numerical values over the time period. Commonly, the term observation is replaced by the equivalent concept of counting. Therefore, the summation of all countings makes up the process with elements $c_j \in \mathbb{P}$. A number of n consecutive countings can be collected together as an information signal S. Formally, there can be many signals, and each of them is a subset of the process $S \subset \mathbb{P}$. Let us denote the initial counting in signal S as $c_0 \in S$; the next one should have the following index $c_1 \in S$. Carrying on with the numbering of countings, the last one in this case would have the designation $c_{n-1} \in S$. This means that only the first initial n countings are considered in signal S. Let us assign this initial signal as $S_0 \subset \mathbb{P}$.

Considering the random process further, it becomes obvious that $c_n \notin S_0$ since there are only n countings in that signal, starting from index 0 to $n-1$. This means that the next signal S_1 starts from the counting c_n. However, the initial counting in each signal (S_0, S_1, etc.) should have a zero index relative to the beginning of the signal. To overcome this issue, the dual indexing, such as c_{ij}, is used. The first index i indicates the signal number S_i, while the second one j specifies the assigned number of the counting in the corresponding signal. In this case the next counting $c_n \in \mathbb{P}$ in a random process is denoted in a random signal $S_1 \subset \mathbb{P}$ by a double indexing $c_n = c_{10} \in S_1$. This approach allows consideration of the countings of a random process from the point of view related to their sorting by signals.

Usually, the sequential signals are considered, and each of them contains an equal number of n countings with equal time intervals τ between them. In this case, the time T_S of each information signal S could be designated as $T_S = (n-1)\tau$. If the time τ between countings admits a continual interpretation, then continuous stationary random processes are considered. If the interval time τ and the number of countings n are finite in the signal S, then such processes are discrete. Generally, the mathematical transition from continuous signals to discrete ones is carried out using the Dirac delta function.

In a random discrete process, the countings are following one by one sequentially. Dividing a process into n sequential countings can be interpreted as observing the sequential discrete signal. Thus, the initial n countings provide forming the signal S_0; the next n countings organize the subsequent signal S_1, and so on. For the next step in signal analysis, it is necessary to check whether the countings in each signal are independent. The consistent observation of countings in a period of time is not exhaustive evidence of their statistical independence. Therefore, it is necessary to analyze their interaction among other countings within the signal. For this purpose, the adjacent pairs of signals are evaluated using the mathematical correlation method. In this case, it is convenient to introduce the

formal autovectors for each pair of signals. The first pair consists of the initial signal S_0 and the adjacent one S_1; the next pair is formed from signals S_1 and S_2, and so on. Thus, the correlation approach uses adjacent signals S_k and S_{k+1}.

To any pair of adjacent signals, for example, S_0 and S_1, there is correspondence of n autovectors ${}^0Z_0$, ${}^0Z_1, \cdots, {}^0Z_{n-1}$. The upper left index 0 underlines the origin of countings, which are taken from signal S_0. Countings of the original signal $c_{0j} \in S_0$ are initially located in the autovector ${}^0z_{0j} \in {}^0Z_0$, which means that countings in the signal S_0 are transferred to the initial autovector ${}^0Z_0$ as follows:

$$\begin{aligned} {}^0z_{0,0} &= c_{0,0}, \\ {}^0z_{0,1} &= c_{0,1}, \ldots \\ {}^0z_{0,n-1} &= c_{0,n-1} \end{aligned} \tag{2}$$

Using the initial autovector (2), the next autovector ${}^0Z_1$ could be obtained, which is shifted one counting to the right relative to the autovector ${}^0Z_0$. Autovector ${}^0Z_1$ contains part of countings of the signal S_0 starting from the counting $c_{01} \in S_0$. Herewith, only the last counting in the autovector ${}^0z_{1,n-1} \in {}^0Z_1$ belongs to the signal S_1. Formation of autovector ${}^0Z_1$ contains the following steps:

$$\begin{aligned} {}^0z_{1,0} &= c_{0,1}, \\ {}^0z_{1,1} &= c_{0,2}, \ldots \\ {}^0z_{1,n-2} &= c_{0,n-1}, \\ {}^0z_{1,n-1} &= c_{1,0} \end{aligned} \tag{3}$$

From Formulas (2) and (3) it follows that the last autovector relative to the pair of signals S_0 and S_1 contains only one counting from the signal S_0, and the remaining $n-1$ countings are taken from the signal S_1 as follows:

$$\begin{aligned} {}^0z_{n-1,0} &= c_{0,n-1}, \\ {}^0z_{n-1,1} &= c_{1,0}, \ldots \\ {}^0z_{n-1,n-2} &= c_{1,n-3}, \\ {}^0z_{n-1,n-1} &= c_{1,n-2} \end{aligned} \tag{4}$$

Thus, with respect to signal S_0 Formula (4) completes the formation of all autovectors ${}^0Z_0, \cdots, {}^0Z_{n-1}$.

Now it is obviously seen that such a structure of autovectors is able to compose the matrix with dimensions $n \times n$. The countings of each autovector ${}^0Z_i$ occupy the corresponding row i in the matrix ${}^0Z$. The values in line 0 correspond to the signal S_0 and the autovector ${}^0Z = S_0$. Line 1 contains the countings of autovector ${}^0Z_1$. The last row of the matrix ${}^0Z$ keeps the autovector ${}^0Z_{n-1}$.

Below is the program *P070101*, in which according to Formula (1) the autovectors ${}^0Z_i$ are created in the matrix ${}^0Z$ using the *Random* generator from the algorithmic language *C#*. The theoretical value of the amount of countings n has been replaced by a program variable named *NS*. As an example, in this particular case, the number of countings in signal is taken as $NS = 33$, although the values for *NS* can be chosen arbitrarily up to $NS < 2^{31}$. The values of countings $c_{0,i}$ of the theoretical initial signal S_0 are stored in the elements of the program array *s*0. Theoretical autovectors ${}^0Z_i$ are located in the program matrix *z*.

```
namespace P070101
{   class cP070101
    {   static void Main(string[] args)
        {  const int NS = 33;                          // signal counter quantity
           Console.WriteLine("NS = {0}", NS);
```

```
            Random rdm = new Random(0);              // integer random generator
            double max = (double)0x7FFFFFFF;
            double[] s0 = new double [NS];                    // initial signal s0
            for (int i = 0; i < NS; i++)
               s0[i] = 4.0 * ((double)rdm.Next() / max - 0.5);
            Console.Write("S0 =");
            for (int i = 0; i < NS; i++)
            { if (i % 6 == 0) Console.WriteLine();
               Console.Write("{0,8:F3}", s0[i]);
            }
            Console.WriteLine();                              // matrix z for vectors
            double[,] z = new double[NS,NS];
            MatrixZ(z, s0, rdm , NS, max);
            Console.WriteLine("Z = ");
            for (int i = 0; i < NS; i++)
            {  Console.Write("{0,8:F3}", z[i, 0]);
               Console.Write("{0,8:F3}", z[i, 1]);
               Console.Write("{0,8:F3}", z[i, 2]);
               Console.Write(" - - - -");
               Console.Write("{0,8:F3}", z[i, NS-2]);
               Console.WriteLine("{0,8:F3}", z[i, NS-1]);
            }
            Console.ReadKey();                                // result viewing
         }
//-------------------------------------------------------------------------
      static void MatrixZ (double [,] z, double[] s0,
                           Random rdm, int NS, double max)
      {  for (int j = 0; j < NS; j++) z[0, j] = s0[j];
         for (int k = 1; k < NS; k++) // vector shift
         {   for (int j = 0; j < NS - 1; j++)
                z[k, j] = z[k - 1, j + 1];
             z[k, NS - 1] = 4.0*((double)rdm.Next()/max-0.5);
         }
      }
   }
}
```

After starting the program *P070101*, the following outcome appears on the monitor. In order to reduce the presentation of the entire listing of results, omitted values are replaced by a dash.

```
NS = 33
S0 =
   0.905     1.269     1.072     0.233    −1.176     0.236
   1.624    −0.231     1.910    −0.905    −0.832    −0.131
   0.531    −0.122     1.929    −1.879     1.449     1.981
   0.709    −0.742     1.268     1.392     1.968    −1.869
   0.800     0.105     1.736     0.750     0.187    −1.676
  −1.252    −0.187    −0.811
Z =
   0.905     1.269     1.072 - - - - - - - - - - −0.187    −0.811
   1.269     1.072     0.233 - - - - - - - - - - −0.811     1.954
   1.072     0.233    −1.176 - - - - - - - - - -  1.954     0.571
   0.233    −1.176     0.236 - - - - - - - - - -  0.571     1.052
```

```
 −1.176      0.236     1.624 - - - - - - - - - - - -  1.052     −1.878
- - - -
 −0.187     −0.811     1.954 - - - - - - - - - - - -  1.581      1.586
 −0.811      1.954     0.571 - - - - - - - - - - - -  1.586     −1.654
```

According to Formulas (2)–(4), in the matrix ${}^0Z$ of this listing there are the autovectors ${}^0Z_1$, ${}^0Z_1, \cdots, {}^0Z_{n-1}$, which are presented line by line.

An analysis of the current results obtained shows that these autovectors are not taken into account by the *random* function during the realization of the white noise generation. However, the white noise process has to keep the properties of statistical independence of the autovectors in each of its S_i signal. This is one of the fundamentals of the theory of transformations of linear vectors, but unfortunately some well-known algorithmic generators do not take this important feature into account. This case provides a discussible issue: should the realization process be named as white noise generation? Moreover, how to take into account the statistical independence checking of autovectors for a generator during the realization of white noise process?

Thus, the purpose of this article is to propose instrumental algorithmic tools for generating statistically independent white noise signals. This method allows a phase signal generator to be created with an improved matrix of autocorrelation coefficients, while maintaining the property of absolutely equivalent intensities at the eigen Fourier frequencies.

2. White Noise Autocorrelation Matrix

The white noise autocorrelation matrix plays a fundamental role in analyzing the statistical independence of white noise signals [1–4,42–44]. Let us demonstrate this utilization the example of the matrix of autovectors ${}^0Z$ from the previous section.

Using the matrix ${}^0Z$, it is possible to calculate the paired scalar multiplications of autovectors ${}^0Z_0$, ${}^0Z_1, \cdots {}^0Z_{n-1}$. To simplify the notation of these autovectors, let us omit the upper left index 0, i.e., instead of ${}^0Z_i$ the indication Z_i is considered further. For autovectors Z_i and Z_k the scalar multiplication (Z_i, Z_k) is determined by the sum of multiplications of countings as follows:

$$(Z_i, Z_k) = \sum_{m=0}^{n-1} z_{im} \cdot z_{km} \tag{5}$$

In Formula (5), the sum of multiplications of the corresponding countings is calculated, provided that these autovectors are in an orthogonal coordinate system. Formally, the geometric length or norm $\|Z_i\|$ of the autovector Z_i in an n-dimensional orthogonal coordinate system is calculated by the following scalar multiplication:

$$\|Z_i\| = \sqrt{\sum_{m=0}^{n-1} x_{im} \cdot x_{im}} = \sqrt{\sum_{m=0}^{n-1} x_{im}^2} \tag{6}$$

Formula (6) formally coincides with the Pythagorean Theorem in a multidimensional linear geometric space. From analytical geometry it is known (or it could be counted directly) that cosine of the angle between two linear vectors is determined by the scalar multiplication (5) between them and their norms (6). So, using Expressions (5) and (6), the following Formula (7) for the angle between the autovectors Z_i and Z_k can be obtained:

$$\cos \varphi_{ik} = \frac{(Z_i, Z_k)}{\|Z_i\| \cdot \|Z_k\|} \tag{7}$$

In applied signal analysis the statistical estimates are very important and could be easily obtained by means of computer technologies. In this sense, it is of significant interest to evaluate the linear connections of autovectors, which for $\cos \varphi_{ik} = 0$ are usually called

statistically independent. In this case such autovectors have a correlation coefficient equal to zero. For autovectors Z_i and Z_k the correlation coefficient is calculated as follows:

$$r_{ik} == \frac{\sum_{m=0}^{n-1}(z_{im} - E_1(Z_i))\cdot(z_{km} - E_1(Z_k))}{\sqrt{\sum_{m=0}^{n-1}(z_{im} - E_1(Z_i))^2}\sqrt{\sum_{k=0}^{n-1}(z_{km} - E_1(Z_k))^2}} \tag{8}$$

In Expression (8) the correlation coefficient r_{ik} deals with centered autovectors relative to the first statistical moments $E_1(Z_i)$ and $E_1(Z_k)$, which are the following:

$$E_1(Z_i) = \frac{1}{n}\sum_{m=0}^{n-1} Z_{im}, \qquad E_1(Z_k) = \frac{1}{n}\sum_{m=0}^{n-1} Z_{km} \tag{9}$$

Further, using the Expressions (9) let us determine the autovector V_i, which is statistically centered for the autovector Z_i after corresponding adjustment of each its counting z_{im}:

$$v_{im} = z_{im} - E_1(Z_i), \qquad m \in \overline{[0:n-1]} \tag{10}$$

Finally, the angle φ_{ik} between the autovectors V_i and V_k is defined by using Expressions (5)–(10) in the following form:

$$\cos\varphi_{ik} = r_{ki} = \frac{(V_i, V_k)}{\|V_i\|\cdot\|V_k\|} \tag{11}$$

From Expressions (8) and (11) it follows that the autovectors Z_i, Z_k and the autovectors V_i, V_k have equal angles to each other. Thus, the initial signal S_0 is statistically independent, since all pairs of its autovectors are orthogonal.

The centered autovectors V_i could be located in the matrix of autovectors V having the size $n \times n$. Below is a program *P070102*, in which the matrix V contains the autovectors V_i obtained from the autovectors Z_i of the matrix Z after using the transformation (10).

```
namespace P070201
{  class cP070201
   {  static void Main(string[] args)
      {   const int NS = 33; // signal counter quantity
          Console.WriteLine("NS = {0}", NS);
          Random rdm = new Random(0); // integer random generator
          double max = (double)0x7FFFFFFF;
          double[] s0 = new double[NS]; // initial signal s0
          for (int i = 0; i < NS; i++)
               s0[i] = 4.0 * ((double)rdm.Next() / max - 0.5); // matrix z for vectors
          double[,] z = new double[NS, NS];
          MatrixZ(z, s0, rdm, NS, max); // matrix v for vectors
          double[,] v = new double[NS, NS];
          MatrixV(v, z, NS);
          Console.WriteLine("V = ");
          for (int i = 0; i < NS; i++)
          {   Console.Write("{0,8:F3}", v[i, 0]);
              Console.Write("{0,8:F3}", v[i, 1]);
              Console.Write("{0,8:F3}", v[i, 2]);
              Console.Write(" - - - -");
              Console.Write("{0,8:F3}", v[i, NS - 2]);
              Console.WriteLine("{0,8:F3}", v[i, NS - 1]);
          }
          Console.ReadKey();                                    // result viewing
      }
```

```
//-------------------------------------------------------------
        static void MatrixV(double[,] v, double[,] z, int NS)
        {   double dNS = (double)NS;
            for (int i = 0; i < NS; i++)
             {  double zE1 = 0.0;
                for (int j = 0; j < NS; j++)
                    zE1 += z[i, j];
                zE1 /= dNS;
                for (int j = 0; j < NS; j++)
                    v[i, j] = z[i, j] - zE1;
            }
        }
//-------------------------------------------------------------
            Function MatrixZ from previous program P070101
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ ~~~~~~~ ~~~~~~~~~~
 }
}
```

After launching the program *P070201*, the following result appears. The omitted values are substituted by a dash.

```
NS = 33
V =
   0.595   0.959   0.762 - - - - −0.497   −1.122
   0.927   0.730  −0.110 - - - - −1.153    1.612
   0.751  −0.088  −1.497 - - - - - 1.633    0.250
  −0.088  −1.496  −0.085 - - - - - 0.250    0.731
  −1.432  −0.021   1.368 - - - - - 0.795   −2.135
- - - - -
  −0.414  −1.039   1.727 - - - - - 1.3544    1.358
  −0.995   1.771   0.388 - - - - - 1.403   −1.837
```

Next, by using the matrix of autovectors V it is possible to obtain the autocorrelation matrix A. For this, the multiplication of the matrix V and its transposed variant V^T is used as follows:

$$A = V \cdot V^T \tag{12}$$

The calculation of the autocorrelation matrix A in Expression (12) is the main key tool for further determining the statistical independence of the signals $S_i \in \mathbb{P}$ in the stochastic process of white noise $\mathbb{P}$.

Using the following formula, it is possible to verify directly that each element $a_{ij} \in A$ is a scalar multiplication of centered autovectors V_i:

$$a_{ij} = \left(V_i, V_j\right) = \sum_{m=0}^{n-1} v_{im} v_{jm} \tag{13}$$

If the autovectors Z_i are orthogonal, then the elements of the autocorrelation matrix A in Expression (13) are zeros, and on the main diagonal there is a minimal dispersion σ^2:

$$A = \sigma^2 I \tag{14}$$

In this Expression (14) the matrix I is the identity matrix, which means that this $n \times n$ square matrix contains the meanings of ones on the main diagonal and zeros elsewhere. Formula (14) is a fundamental tool of verifying whether a stochastic process $\mathbb{P}$ can be considered as white noise.

If the signals are not orthogonal, then in the matrix R the elements $r_{ij} \in R$ show the cosines of the angles between the autovectors Z_i and Z_j for the corresponding elements a_{ij} (according to Formulas (8)–(11)). The program *P070202* below calculates the autocorrelation matrix A and autocorrelation coefficients R in matrix V.

```
namespace P070202
{   class cP070202
    {   static void Main(string[] args)
        {  const int NS = 33;                                  // signal counter quantity
           Console.WriteLine("NS = {0}", NS);
           Random rdm = new Random(0);                        // integer random generator
           double max = (double)0x7FFFFFFF;
           double[] s0 = new double[NS];                          // initial signal s0
           for (int i = 0; i < NS; i++)
           s0[i] = 4.0 * ((double)rdm.Next() / max - 0.5);        // matrix z for vectors
           double[,] z = new double[NS, NS];
           MatrixZ(z, s0, rdm, NS, max);                          // matrix v for vectors
           double[,] v = new double[NS, NS];
           MatrixV(v, z, NS);                                  // autocorrelation matrix A
           double[,] a = new double[NS, NS];
           MatrixA(a, v, NS);
           Console.WriteLine("A = ");
           for (int i = 0; i < NS; i++)
           {   Console.Write("{0,8:F3}", a[i, 0]);
               Console.Write("{0,8:F3}", a[i, 1]);
               Console.Write("{0,8:F3}", a[i, 2]);
               Console.Write(" - - - -");
               Console.Write("{0,8:F3}", a[i, NS - 2]);
               Console.WriteLine("{0,8:F3}", a[i, NS - 1]);
           }                                          // autocorrelation coefficient matrix R
           double[,] r = new double[NS, NS];
           MatrixR(r, a, v, NS);
           Console.WriteLine("R = ");
           for (int i = 0; i < NS; i++)
           {   Console.Write("{0,8:F3}", r[i, 0]);
               Console.Write("{0,8:F3}", r[i, 1]);
               Console.Write("{0,8:F3}", r[i, 2]);
               Console.Write(" - - - -");
               Console.Write("{0,8:F3}", r[i, NS - 2]);
               Console.WriteLine("{0,8:F3}", r[i, NS - 1]);
           }
           Console.ReadKey();                                     // result viewing
        }
//-------------------------------------------------------------------
       static void MatrixR(double[,] r, double[,] a,
                           double[,] v, int NS)
        {   for (int i = 0; i < NS; i++)
               for (int j = i; j < NS; j++)
               {   double iE2 = 0.0;
                   double jE2 = 0.0;
                   for (int m = 0; m < NS; m++)
                   {   iE2 += v[i, m] * v[i, m];
                       jE2 += v[j, m] * v[j, m];
                   }
```

```
                r[i, j] = a[i, j] / Math.Sqrt(iE2 * jE2);
                r[j, i] = r[i, j];
            }
        }
//---------------------------------------------------------------
        static void MatrixA(double[,] a, double[,] d, int NS)
        {   for (int i = 0; i < NS; i++)
                for (int j = i; j < NS; j++)
                {   a[i, j] = 0.0;
                    for (int m = 0; m < NS; m++)
                        a[i, j] += d[i, m] * d[j, m];
                    a[j, i] = a[i, j];
                }
        }
//---------------------------------------------------------------
 Function MatrixV from previous program P070201
//---------------------------------------------------------------
 Function MatrixZ from previous program P070101
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    }
}
```

After executing the program *P070202*, the autocorrelation matrix *A* and the matrix of autocorrelation coefficients *R* appear on the monitor. The omitted values are changed by a dash.

```
NS = 33
A =
   44.628  −4.879  −1.959  - - - -   5.740  −2.823
  −4.879   46.943  −5.032  - - - - −6.393   2.941
  −1.959  −5.032   46.121  - - - - −3.466  −6.470
  −12.693 −1.203  −5.562  - - - - −9.785  −4.531
   2.965  −16.188 −1.669  - - - -   5.198  −5.778
- - - - -
   5.740  −6.393  −3.466  - - - -  39.945   0.869
  −2.823   2.941  −6.470  - - - -   0.869  43.248
R =
    1.000  −0.107  −0.043  - - - -   0.136  −0.064
  −0.107   1.000  −0.108  - - - - −0.148   0.065
  −0.043  −0.108   1.000  - - - - −0.081  −0.145
  −0.280  −0.026  −0.121  - - - -   0.228  −0.101
   0.062  −0.332  −0.034  - - - -   0.115  −0.123
- - - - -
   0.136  −0.148  −0.081  - - - -   1.000   0.021
  −0.064   0.065  −0.145  - - - -   0.021   1.000
```

In the analysis of signals a set of consecutive n countings makes it possible to form or detect discrete spectra of multiple internal Fourier frequencies. By the nature of the frequency distribution there are white noise signals with a uniform distribution of intensities, a normal Gaussian distribution, and many others. This article focuses on white noise with a uniform intensity distribution at Fourier frequencies, i.e., all frequencies are of the same intensity.

In a random process, the signals of uniform white noise are characterized by the following properties:

(1) Uniform distribution of intensities at all internal frequencies of the signal;
(2) Zero value of mathematical expectation (the first moment) of the values of countings in the signal;
(3) Autocorrelation matrix has a diagonal form with equal meanings of dispersion (the second moment) along the main diagonal and zero values for all other elements.

From this designation it follows that white noise signals should be orthogonal in an environment of independent countings with equal amplitudes at all internal frequencies. Therefore, property (1) forces us to consider the set of exactly n countings in the signals S with the further use of Fourier frequency analysis.

The simulation results in the above presented programs *P070101*, *P070201* and *P070202* show that the *Random* generator creates the countings of signals with low quality for white noise process. Their autocorrelation matrix A and the corresponding matrix of autocorrelation coefficients R are far from the desirable zero values. In line with all of this, in this article we offer a new congruential generator which has better quality of generating the uniform white noise.

3. Theory

Consider a model, in which the values of countings of white noise random process are present at a finite observation interval T. This process starts at point t_0. In the interval $[t_0,\ t_1] = [t_0,\ t_0 + T]$ there are N_S countings of the initial signal S_0; in the interval $[t_1,\ t_2] = [t_1,\ t_1 + T]$ there are N_S countings of the signal S_1, and so on.

The designation N_S for the number of countings in the signal exactly corresponds to n in the previous sections above, i.e., $N_S = n$. Suppose that in all signals the countings are located with a constant step:

$$\tau = \frac{T_S}{N_S - 1} \tag{15}$$

In each signal $S_0,\ S_1,\ \cdots, S_{n-1}$ a certain spectrum of frequencies is fixed. If among these frequencies there is one with a period T_S, then it is denoted as the initial frequency ω_1. Typically, in trigonometric studies the interval $[-\pi,\ +\pi]$ is used. An isomorphic transition of an observation point $t \in [0, T_S]$ to an isomorphic point $x \in [-\pi, +\pi]$ is performed as follows:

$$x = -\pi + 2\pi \frac{t}{T_S} \tag{16}$$

On the interval $[-\pi, +\pi]$ the frequency ω_1 is equal to one, i.e., $\omega_1 = 1$. Frequencies $\omega_k = k{\cdot}\omega_1$, integral multiples of the initial frequency ω_1, refer to the discrete Fourier spectrum, in which N_F is the quantity of frequencies. These frequencies make it possible to organize an orthonormal system of sine-cosine coordinates in Euclidean space $E_0[-\pi, +\pi]$:

$$\frac{1}{\sqrt{2\pi}},\ \frac{\cos\omega_1 x}{\sqrt{\pi}},\ \frac{\sin\omega_1 x}{\sqrt{\pi}},\ \frac{\cos 2\omega_1 x}{\sqrt{\pi}},\ \frac{\sin 2\omega_1 x}{\sqrt{\pi}},\ \cdots,\ \frac{\cos N_F\omega_1 x}{\sqrt{\pi}},\ \frac{\sin N_F\omega_1 x}{\sqrt{\pi}} \tag{17}$$

In the trigonometric space with the coordinate system (17) the countings are observed at the points $x_i \in [-\pi, +\pi]$, $i \in \overline{[0 : N_s - 1]}$. By the property of this space (17), the values of countings $f(x_i)$ could be determined by the following Fourier polynom:

$$f(x_i) = a_0 + \sum_{k=1}^{N_F} (a_k \cos k\omega_1 x_i + b_k \sin k\omega_1 x_i) \tag{18}$$

It is assumed that if the information signal is closely approximated to the white noise signal, then it should have the same amplitudes at all frequencies $\omega_k = k\omega_1$. However, in Expression (18), each frequency is accompanied by two amplitudes a_k and b_k with possibly separate distributions. This approach leads to a significant complication of the algorithm for their calculations. To get around this obstacle, let us replace the calculation of each pair

of intensities a_k, b_k with the generation of uniform intensity A_k at multiple frequencies $k\omega_1$ with a phase shift. Let us analyze how this could be done in more detail.

It is known from trigonometric transformations that the spectral sine of the sum of two angles γ_k and ψ_k can be calculated using the following expression:

$$A_k \sin(\gamma_k + \psi_k) = A_k \sin\gamma_k \cdot \cos\psi_k + A_k \cos\gamma_k \cdot \sin\psi_k \tag{19}$$

Considering Expressions (18) and (19) together, the following estimates are obtained:

$$\begin{gathered} \gamma_k = k\omega_1 x_i, \\ a_k = A_k \sin\psi_k, \\ b_k = A_k \cos\psi_k \end{gathered} \tag{20}$$

Using the Formulas (20), the relationship between the values A_k and the Fourier coefficients a_k, b_k can be established as follows:

$$\begin{gathered} a_k^2 + b_k^2 = A_k^2(\sin\psi_k)^2 + A_k^2(\cos\psi_k)^2 = A_k^2 \\ A_k = \sqrt{a_k^2 + b_k^2} \end{gathered} \tag{21}$$

The value of the angle ψ_k is calculated using the Formulas (20) as well:

$$\psi_k = \arcsin\frac{a_k}{A_k} \; or \; \psi_k = \arccos\frac{b_k}{A_k} \tag{22}$$

Thus, Expression (18) is equivalent to the following sine ratio:

$$f(x_i) = a_0 + \sum_{k=1}^{N_\omega} A_k \sin(k\omega_1 x_i + \psi_k) \tag{23}$$

A similar theoretical result can be obtained by using the cosine of the sum of two angles:

$$A_k \cos(\gamma_k + \varphi_k) = A_k \cos\gamma_k \cos\varphi_k - A_k \sin\gamma_k \sin\varphi_k \tag{24}$$

Considering Expressions (18) and (24) together, the following estimates are derived:

$$\begin{gathered} \gamma_k = k\omega_1 x_i, \\ a_k = A_k \cos\varphi_k, \\ b_k = -A_k \sin\varphi_k \end{gathered} \tag{25}$$

Using the Formulas (25), the relationship between the values A_k and the Fourier coefficients a_k, b_k can be established as follows:

$$\begin{gathered} a_k^2 + b_k^2 = A_k^2(\cos\varphi_k)^2 + A_k^2(\sin\varphi_k)^2 = A_k^2 \\ A_k = \sqrt{a_k^2 + b_k^2} \end{gathered} \tag{26}$$

The value of the angle ψ_k is calculated using the Formulas (25) as well:

$$\varphi_k = \arccos\frac{a_k}{A_k} \; or \; \varphi_k = -\arcsin\frac{b_k}{A_k} \tag{27}$$

Thus, Expression (18) is equivalent to the following cosine ratio:

$$f(x_i) = a_0 + \sum_{k=1}^{N_\omega} A_k \cos(k\omega_1 x_i + \varphi_k) \tag{28}$$

The value a_0 is not a randomly organized constant. For a white noise generator it could be set as 0 or any other number. Since the values of the amplitudes A_k have to be constant in white noise, they can be chosen based on natural tests, or set equal to the universal meaning such as value one, for example:

$$A_1 = A_2 = \cdots = A_{N_\omega} = const_A = 1 \tag{29}$$

Frequency components $k\omega_1 x_i$ are also not random variables in Fourier space. Therefore, only stochastic phases ψ_k of sine (23) or φ_k cosine (28) signals can provide stochastic capabilities of a white noise signal at Fourier frequencies. This article discusses stochastic phases ψ_k and φ_k which are uniformly distributed in the interval $[-\pi/2, +\pi/2]$, since the functions *arcsine* and *arccosine* are used in Formulas (22) and (27).

Now it is necessary to assign the corresponding generator of uniformly distributed values for the stochastic phases ψ_k and φ_k. In our previous articles [45–50], we thoroughly explored the capabilities of new congruential and twister generators, which ensure absolute completeness and uniformity of integer random variable distribution. Based on the main principles outlined in [45–50], further here we have established a new generator, which is specially designed to implement the white noise process with the above mentioned properties. In accordance with this, below is the basic generator *cDeonYuliCongBase62* and on its basis the derived congruential generator *cDeonYuliCongSequence62*. Together they provide the absolute completeness of sequences of uniform integer random variables of arbitrary size.

To design these generators, it is necessary, first of all, to choose the size or number of bits in an integer random variable. Below is presented the base class *cDeonYuliCongBase62*, in which the number of bits w of random integer variables is specified. Their amount can be arbitrary in the range $2 \le w \le 62$. The quantity of random variables N in one sequence is $N = 2^w$. Class *cDeonYuliCongBase62* is made in the *C#* programming language in *Microsoft Visual Studio*. This class is located in a separate namespace file *nsDeonYuliCongBase62*.

```
namespace nsDeonYuliCongBase62
{   class cDeonYuliCongBase62
    {   public int w;                                  // bit length of random variable
        public bool wFlag;                                    // flag of w setting
        public long N;                          // quantity of variables in the sequence
        public bool NFlag;                                    // flag of N setting
//-------------------------------------------------------------------
        public cDeonYuliCongBase62()
        {   wFlag = false;                                          // w disable
            NFlag = false;                                          // N disable
        }
//-------------------------------------------------------------------
        public void SetW (int rw)
        {   w = rw;                                 // the bit length of random variable
            if (w < 2) w = 3;
            if (w > 63) w = 63;                                   // maximal bit length
            wFlag = true;                                         // the bit length is set
            NFlag = false;                                      // to verify the bit length
            VerifyWN();                               // to verify w and N parameters
        }
//-------------------------------------------------------------------
        public void VerifyWN()
        {   if ( !wFlag && !NFlag )
            {   w = 4;                                    // the bit length by default
                wFlag = true;                                     // the bit length is set
                N = 1L << w; // the sequence length
```

```
            NFlag = true;                                   // the sequence is set
            return;
        }
        if (wFlag && !NFlag)
        {  N = 1L << w; // the sequence length
            NFlag = true; // the sequence is set
            return;
        }
        if (!wFlag && NFlag)
        {  long r = 1L;
            w = 0;
            while (r < N) { r <<= 1; w++; }
            wFlag = true; // the bit length is set
            N = 1L << w; // the sequence length
            NFlag = true; // the bit length is set
            return;
        }
      }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  }
}
```

This base class is the basis for creating the uniform sequences in the derived class *cDeonYuliCongSequence62* having congruential parameters a and c. In a congruential sequence of N random elements the adjacent random variables x_i and x_{i+1} are calculated using the following formula:

$$x_{i+1} = (ax_i + c)\ mod\ N \tag{30}$$

The parameter a in (30) has the following property:

$$(a-1)\ mod\ 4 = 0 \tag{31}$$

The parameter c is the odd number in (30):

$$c\ mod\ 2 \neq 0 \tag{32}$$

For congruential generation in accordance with Formula (30), compliance with properties (31) and (32) is mandatory. Below is a derived *class cDeonYuliCongSequence62*, in which these properties are checked and subsequent congruential random variables are generated. This class is located in a separate namespace file *nsDeonYuliCongSequence62*.

```
using nsDeonYuliCongBase62;                          //congruential base class
namespace nsDeonYuliCongSequence62
{  class cDeonYuliCongSequence62 : cDeonYuliCongBase62
   {  public long a;                                 // multiplicative constant
      public bool aFlag;                             // setting flag of parameter a
      public long c;                                 // additive constant
      public bool cFlag;                             // setting flag of parameter c
      public long x0Beg;                             // initial setting of a
      public long x0;                                // sequence beginning
      public bool x0Flag;                            // setting flag of x0
      bool x0TimeFlag;                               // true – setting x0 by timer
      public bool xeFlag;                            // sequence end flag
      public long x;                                 // current random variable
```

```
    public long xCounter;                                   // random variable counter
    public long sCounter;                                    // counter of sequences
//-----------------------------------------------------------------
    public cDeonYuliCongSequence62()
    {  aFlag = false;                                        // parameter a is not set
       cFlag = false;                                        // parameter c is not set
       xeFlag = false;                                     // there is no sequence end
       x0Flag = false;                                   // there is no sequence beginning
       x0TimeFlag = false;                                   // setting x0 by not timer
    }
//-----------------------------------------------------------------
    public void SeqStart()
    {  if (!wFlag || !NFlag) SetW(4);                                      // by default
       if (!aFlag) a = N / 2L;                                  // parameter a by default
       SeqVerifyA();                                  // congruential verification for a
       aFlag = true;                                              // parameter a is set
       if (!cFlag) c = 3L;                                      // default parameter a
       SeqVerifyC();                                  // congruential verification for c
       cFlag = true;                                              // parameter c is set
       x0 = x0Beg;                                          // initial congruential value
       x = x0;                                       // congruential sequence beginning
       xCounter = 0L;                                           // random counter value
       sCounter = 1L;                                             // sequence counter
    }
//-----------------------------------------------------------------
    public void SeqTimeStart()
    {  x0TimeFlag = true;                                          // start by timer
       x0 = (long)DateTime.Now.Millisecond;                                  // msec
       x0 = x0 % N;                                         // initial random variable
       x0Flag = true;                                                 // x0 is set
       SeqStart();                                     // random variable generation
    }
//-----------------------------------------------------------------
    public long SeqNext()
    {  if (0L < xCounter && xCounter < N)                                  // x counter
       {  x = SeqCong(x);                              // random variable generation
          xCounter++;                                                 // x counter
          return x;                                             // random variable x
       }
       if (xCounter == 0L)
       {  x = x0;                                              // sequence beginning
          xCounter = 1L;                                              // x counter
        return x;                                               // random variable x
       }
       if (x0Flag == false) x0 = (x0 + 1L) % N;
       else x0 = SeqCong(x0);                                // congruential variable x0
       x = x0;                                                  // random variable
       xCounter = 1L;                                              // x counter
       if (sCounter < N) sCounter++;
       else { x0 = x0Beg; x = x0; sCounter = 1L; }
       return x;                                                 // random variable
    }
//-----------------------------------------------------------------
    void SeqTimeInit()
```

```
    {  long xt = (long)DateTime.Now.Millisecond;
       x = xt % N;
    }
//------------------------------------------------------------------
    public long SeqCong(long xz)
    {   return (a * xz + c) % N;                          // congruential variable
    }
//------------------------------------------------------------------
    public void SetAC(long ra, long rc)
    {   a = ra;                                        // multiplicative constant a
       SeqVerifyA();                                                 // to verify a
       c = rc;                                                // additive constant c
       SeqVerifyC(); // to verify c
    }
//------------------------------------------------------------------
    public void SetA(long ra)
    {   a = ra; // multiplicative parameter a
       SeqVerifyA();                                                 // to verify a
       aFlag = true;                                        // parameter a is set
    }
//------------------------------------------------------------------
    public void SetC(long rc)
    {   c = rc;                                            // additive parameter c
       SeqVerifyC();                                                 // to verify c
       cFlag = true;                                  // additive parameter c is set
    }
//------------------------------------------------------------------
    public void SetX0(long rx0, bool flag)
    {   x0 = rx0;
       SeqVerifyX0();                                        // to verify initial value
       x0Beg = x0;                                             // initial x0 setting
       x0Flag = flag;                              // true – x0 beginning of sequence
    }
//------------------------------------------------------------------
    public void SeqVerifyA()
    {   if (a < 1L) a = 1L;
       if (a >= N) a = N - 1;
       for (int i = 0; i < 3; i++)
           if ((a - 1) % 4L == 0) break;
           else a–;
       aFlag = true;                                        // parameter a is set
    }
//------------------------------------------------------------------
    public void SeqVerifyC()
    {   if (c < 0L) c = 1L;
       if (c >= N) c = N - 1L;
       if (c % 2L == 0L) c–;
       cFlag = true;
       return;
    }
//------------------------------------------------------------------
    public void SeqVerifyX0()
    {   if (x0 < 0L) x0 = 0L;
       if (x0 >= N) x0 = N - 1L;
```

```
            x0Flag = true;                                  // sequence beginning is set
        }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    }
}
```

These two instrumental classes are sufficient to ensure the computation of uniformly distributed phases in stochastic spectra.

4. Construction and Results

When converting a discrete signal into the sum of Fourier frequencies, the following fundamental relationship is realized between the minimum amount of countings N_S and the maximum one N_F of Fourier frequencies on a circle of unit radius 2π long:

$$N_S \geq 2 \cdot N_F + 1 \tag{33}$$

As an example, let us set arbitrarily the quantity of frequencies in white noise equal to $N_F = 16$. Then, by condition (33), each signal can contain the following amount of countings $N_S = 2 \cdot N_F + 1 = 2 \cdot 16 + 1 = 33$.

According to Expressions (23) or (28), each spectral frequency $\omega_k = k\omega_1$ has its own phase $\psi_k, \varphi_k \in [-\pi/2k, +\pi/2k]$, respectively. To generate random phases, let us take stochastic sequences consisting of $N = N_F = 2^w$ integer congruential variables $x_{cong} \in \overline{[0, N_F - 1]} = \overline{[0 : 2^w - 1]}$. In binary form, each integer random variable x_{cong} has the following number w of bits:

$$w = \log_2 N_F \tag{34}$$

It should be emphasized that the congruential generator only works with an integer number of bits. From Expression (34) it follows that a quantity of Fourier frequencies should correspond to the following power function:

$$N_F = 2^w \tag{35}$$

The phase interval $[-\pi/2, +\pi/2]$ of length π/k is divided into N_F subintervals of length d_φ each:

$$d_\varphi = \frac{\pi}{kN_F} \tag{36}$$

Using Expressions (35) and (36), the random phases ψ_k or φ_k are determined by the congruential technology [45–50] using corresponding integer random variable $x_{cong\ k}$:

$$\psi_k = \varphi_k = x_{cong\ k} \cdot d_\varphi - \frac{\pi}{2k} \tag{37}$$

Stochastic phases (37) set the random nature of values for the countings in the white noise generator.

The stochastic values of countings together with phases for the white noise are computed in the derived class *cDeonYuliCongPhase62A* below, using base classes *cDeonYuliCongSequence62* and *cDeonYuliCongBase62* from the previous section. Joint testing of these classes will be carried out here later using sine based technology (23) in the *P070401* program.

```
using nsDeonYuliCongSequence62;                     // congruential generator
namespace nsDeonYuliCongPhase62A                    // congruential phase generator
{   class cDeonYuliCongPhase62A : cDeonYuliCongSequence62
    {   public long NS;                             // counter quantity in signal
        public long NF;                             // Fourier frequency quantity
        public double constA =1.0;                  // uniform frequency amplitude
```

```
    public double w1f;                                  // initial Fourier frequency
    public long[] cong;                                 // congruential sequence
    public double[] psi;                                // phase frequencies
    public long iNS;                                    // counter number
    public double dxs;                                  // counter point step
    public double xsWN;                                 // counter value
//---------------------------------------------------------------------
  public cDeonYuliCongPhase62A (long _NF, long _NS)
   {  NS = _NS;                                         // counter quantity in signal
      if (NS < 17L) NS = 17L;                           // default counter quantity
      iNS = -1L; // counter number
      NF = _NF;                                         // frequency quantity in counter
      if (NF < 4) NF = 4L;                              // default frequency quantity
      w1f = 1.0;                                        // default initial Fourier frequency
      dxs = 2.0 * Math.PI / NS;                         // counter point step
      int wf = 0;                                       // initial bit length of random variable
      for (long nf = 1L; nf < NF; nf *= 2L) wf += 1;
      w = wf;                                           // bit length of random variables
      SetW(w);                                          // set w
   }
//---------------------------------------------------------------------
  public void SetACX( long _a, long _c, long _x0)
   {  SetAC(_a, _c);                                    // congruential parameters
      SetX0(_x0, true);                                 // beginning of congruential sequence
   }
//---------------------------------------------------------------------
  public void SetAmplitude( double _A)
   {  constA = _A;                                      // amplitude of all frequencies
   }
//---------------------------------------------------------------------
  public void PhaseStart()
   {  SeqStart();                                       // congruential generator start
      cong = new long[N+1];                             // congruential sequence
      psi = new double[N+1];                            // frequency phases
      PhaseCong();                                      // congruential sequence of phases
   }
//---------------------------------------------------------------------
  void PhaseCong()
   {  for (int k = 1; k <= N; k++)
      {  cong[k] = SeqNext();                           // random variable
         double pi2k = Math.PI / (double)k / 2.0;
         double dpsik = pi2k / (double)N;               // phase shift
         psi[k] = dpsik * (double)cong[k] - pi2k / 2.0;
      }
   }
//---------------------------------------------------------------------
  public double PhaseSinNext( double x)
   {  iNS++;                                            // counter point number
      if (iNS == NS) { iNS = 0; PhaseCong(); }
      PhaseSinWN(x);                                    // calculation in point x
      return xsWN;                                      // white noise in point x
   }
//---------------------------------------------------------------------
  void PhaseSinWN(double x)
```

```
    {  double f = 0.0; // frequency value sum
       for (long k = 1; k <= NF; k++)
       {  double wk = (double)k*w1f;                    // spectrum frequency
          f += constA * Math.Sin(wk * x + psi[k]);
       }
       xsWN = f;                                        // counter value in point x
    }
//-------------------------------------------------------------------
   public double PhaseCosNext( double x)
    {  iNS++;                                           // counter point number
       if (iNS == NS) { iNS = 0; PhaseCong(); }
       PhaseCosWN(x);                                   // calculation in point x
       return xsWN;                                     // white noise in point x
}
//-------------------------------------------------------------------
   void PhaseCosWN(double x)
    {  double f = 0.0; // counter value
       for (long k = 1; k <= NF; k++)
       {  double wk = (double)k*w1f;
                                                        // spectrum frequency
          f += constA * Math.Cos( wk * x + psi[k]);
       }
       xsWN = f; // counter value in point x
    }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  }
}
```

Below, in the program *P070401* the values of countings are calculated by Formula (37) using the congruential phase generator *nsDeonYuliCongPhase62A*. Autovectors $Z_0, \cdots, Z_{32}$ are determined on the basis of the initial signal S_0 with the addition of the next counting of the random process by analogy with Formulas (2) and (3). These autovectors are located in the matrix Z. For example, each autovector $Z_0, \cdots, Z_{32}$ is in the corresponding row $m \in \boxed{0:32}$ of the matrix Z. The original signal S_0 is generated using congruential constants $a = 5$ and $c = 3$. The intensities A_k of all internal frequencies are arbitrarily set equal to $A_k = 0.7$.

```
using nsDeonYuliCongPhase62A;                   // congruential phase generator
namespace P070401
{  class cP070401
   {  static void Main(string[] args)
      {  const long NS = 33L;                           // signal counter quantity
         const long NF = 16L;                    // frequency quantity in a counter
         Console.WriteLine("NS = {0} NF = {1}", NS, NF);
         cDeonYuliCongPhase62A PH =
                        new cDeonYuliCongPhase62A(NF, NS);
         Console.WriteLine("tau = {0:F6}", PH.dxs);
         double constA = 0.7;                       // amplitude of all frequencies
         PH.SetAmplitude(constA);
         Console.WriteLine("constA = {0:F2}", PH.constA);
         PH.SetACX(5L, 3L, 2L);                     // congruential parameters
         Console.WriteLine("a = {0} c = {1} Cong(x0) = {2}",
                           PH.a, PH.c, PH.x0);
         PH.PhaseStart();                                   // phase generator start
         Console.WriteLine("cong =");
```

```
            for (int i = 1; i <= NF; i++)
            {  Console.Write("{0,4}", PH.cong[i]);
               if (i % 12 == 0) Console.WriteLine();
            }
            Console.WriteLine();
            Console.WriteLine("psi =");
            for (int i = 1; i <= NF; i++)
            {   Console.Write("{0,8:F3}", PH.psi[i]);
                if (i % 6 == 0) Console.WriteLine();
            }
            Console.WriteLine();
            double[] s0 = new double[NS];                          // initial signal s0
            double t = 0.0;                                         // counter time
            for (int i = 0; i < NS; i++)
            {  s0[i] = PH.PhaseSinNext(t);                          // counter value
               t += PH.dxs;
            }
            Console.Write("S0 =");
            for (int i = 0; i < NS; i++)
            {  if (i % 6 == 0) Console.WriteLine();
               Console.Write("{0,8:F3}", s0[i]);
            }
            Console.WriteLine();                                    // matrix z for vectors
            t = 0.0;                                               // next period beginning
            double[,] z = new double[NS, NS];
            MatrixZ(z, s0, PH, NS, t);
            Console.WriteLine("Z = ");
            for (int i = 0; i < NS; i++)
            {  Console.Write("{0,8:F3}", z[i, 0]);
               Console.Write("{0,8:F3}", z[i, 1]);
               Console.Write("{0,8:F3}", z[i, 2]);
               Console.Write(" - - - -");
               Console.Write("{0,8:F3}", z[i, NS - 2]);
               Console.WriteLine("{0,8:F3}", z[i, NS - 1]);
            }
            Console.ReadKey();                                      // result viewing
        }
    //-----------------------------------------------------------------
        static void MatrixZ(double[,] z, double[] s0,
                        cDeonYuliCongPhase62A PH, long ns, double t)
        {  for (int j = 0; j < ns; j++) z[0, j] = s0[j];
           for (int k = 1; k < ns; k++)                             // autovector shift
           {  for (int j = 0; j < ns - 1; j++)
                  z[k, j] = z[k - 1, j + 1];
              z[k, ns - 1] = PH.PhaseSinNext(t);
              t += PH.dxs;
           }
        }
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    }
}
```

After starting the program *P070401*, a complete uniform congruential sequence *cong* of integers appears on the monitor (presented in the listing below). Next in listing is a

stochastic sequence of congruential phases *psi*, which is derived from uniform integers *cong*. This sequence is used to compute the countings of signal *S0* using a technique of sine harmonics (23). Further, on the basis of the initial signal *S0* the matrix *Z* of autovectors is created. Each autovector is located on the corresponding row of the matrix *Z*. To shorten this listing, missing numbers and matrix rows have been replaced with a dash.

```
NS = 33      NF = 16
tau = 0.190400
constA = 0.70
a = 5   c = 3   Cong(x0) = 2
cong =
    2   13   4   7   6   1   8   11   10   5   12   15
    14  9    0   3
psi =
 −0.589   0.245  −0.131  −0.025  −0.039  −0.115
  0.000   0.037   0.022  −0.029   0.036   0.057
  0.045   0.007  −0.052  −0.031
S0 =
 −0.371   6.938  −0.398   2.317  −0.415   1.159
 −0.540   0.626  −0.283   0.545  −0.396   0.479
 −0.117   0.736   0.200   0.980   0.238   0.919
  0.215   0.836  −0.125   0.483  −0.566   0.146
 −0.779   0.226  −1.147  −0.087  −1.497  −0.077
 −2.373  −0.201  −7.674
Z =
 −0.371   6.938  −0.398  - - - -  −0.201  −7.674
  6.938  −0.398   2.317  - - - -  −7.674   0.087
 −0.398   2.317  −0.415  - - - -   0.087   7.450
  2.317  −0.415   1.159  - - - -   7.450   0.015
 - - - - -
 −0.077  −2.373  −0.201  - - - -   0.436  −1.075
 −2.373  −0.201  −7.674  - - - -  −1.075   0.432
 −0.201  −7.674   0.087  - - - -   1.000   0.021
 −7.674   0.087   7.450  - - - -  −1.923   0.169
```

The result of this listing allows the movement from the matrix of autovectors *Z* to centering them with the same line-by-line arrangement in the matrix *V*. This is fulfilled by analogy with Formulas (9) and (10) regarding the mathematical expectation for each autovector separately. Below is program *P070402*, which presents how to implement this.

```
using nsDeonYuliCongPhase62A;                          // congruential phase generator
namespace P070402
{  class cP070402
   {  static void Main(string[] args)
      {  const long NS = 33L;                                  // signal counter quantity
         const long NF = 16L;                          // frequency quantity in a counter
         Console.WriteLine("NS = {0} NF = {1}", NS, NF);
         cDeonYuliCongPhase62A PH =
                        new cDeonYuliCongPhase62A(NF, NS);
         double constA = 0.7;                             // amplitude of all frequencies
         PH.SetAmplitude(constA);
         PH.SetACX(5L, 3L, 2L);                          // congruential parameters
         PH.PhaseStart();                                   // phase generator start
         double[] s0 = new double[NS];                            // initial signal s0
```

```
            double t = 0.0;                                  // the beginning of signal counters
            for (int i = 0; i < NS; i++)
            {  s0[i] = PH.PhaseSinNext(t);                                  // counter value
               t += PH.dxs;                                               // next counter time
            }
            Console.Write("S0 =");
            for (int i = 0; i < NS; i++)
            {  if (i % 6 == 0) Console.WriteLine();
               Console.Write("{0,8:F3}", s0[i]);
            }
            Console.WriteLine();                                          // autovector matrix z
            t = 0.0;                                                     // next period beginning
            double[,] z = new double[NS, NS];
            MatrixZ(z, s0, PH, NS, t);                                    // autovector matrix v
            double[,] v = new double[NS, NS];
            MatrixV(v, z, NS);
            Console.WriteLine("V = ");
            for (int i = 0; i < NS; i++)
            {  Console.Write("{0,8:F3}", v[i, 0]);
               Console.Write("{0,8:F3}", v[i, 1]);
               Console.Write("{0,8:F3}", v[i, 2]);
               Console.Write(" - - - -");
               Console.Write("{0,8:F3}", v[i, NS - 2]);
               Console.WriteLine("{0,8:F3}", v[i, NS - 1]);
            }
            Console.ReadKey();                                            // result viewing
        }
    //-----------------------------------------------------------------------
        static void MatrixV(double[,] v, double[,] z, long ns)
        {  double dns = (double)ns;
           for (int i = 0; i < ns; i++)
           {   double zE1 = 0.0;
               for (int j = 0; j < ns; j++)
                   zE1 += z[i, j];
               zE1 /= dns;
                   for (int j = 0; j < ns; j++)
                       v[i, j] = z[i, j] - zE1;
           }
        }
    //-----------------------------------------------------------------------
     Function MatrixZ from previous program P070401
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      }
    }
```

After executing the program *P070402*, the matrix V of centered autovectors appears on the monitor. To shorten the listing the skipping values have been substituted with a dash.

```
NS = 33     NF = 16
S0 =
 −0.371   6.938   −0.398   2.317   −0.415    1.159
 −0.540   0.626   −0.283   0.545   −0.396    0.479
 −0.117   0.736    0.200   0.980    0.238    0.919
  0.215   0.836   −0.125   0.483   −0.566    0.146
```

```
 −0.779   0.226   −1.147  −0.087  −1.497  −0.077
 −2.373  −0.201   −7.674
V =
 −0.371   6.938   −0.398  - - - -  −0.201  −7.674
  6.924  −0.411    2.303  - - - -  −7.688   0.073
 −0.427   2.288   −0.445  - - - -   0.057   7.421
  2.275  −0.457    1.117  - - - -   7.408  −0.027
 −0.427   1.102   −0.597  - - - -  −0.043   2.767
 - - - - -
 −0.022  −2.318   −0.146  - - - -   0.491  −1.020
 −2.333  −0.161   −7.635  - - - -  −1.035   0.472
 −0.175  −7.648    0.113  - - - -   0.458  −1.897
 −7.660   0.102    7.465  - - - -  −1.908   0.184
```

The matrix V of centered autovectors allows the calculation of the autocorrelation matrix A and the matrix R of the corresponding autocorrelation coefficients by analogy with Formulas (11) and (12) from the program *P070202*. In the following program *P070403*, the corresponding calculations are performed.

```
using nsDeonYuliCongPhase62A;                    // congruential phase generator
namespace P070403
{  class cP070403
   {  static void Main(string[] args)
      {  const long NS = 33L;                              // signal counter quantity
         const long NF = 16L;                       // frequency quantity in a counter
         Console.WriteLine("NS = {0} NF = {1}", NS, NF);
         cDeonYuliCongPhase62A PH =
                        new cDeonYuliCongPhase62A(NF, NS);
         double constA = 0.7;                          // amplitude of all frequencies
         PH.SetAmplitude(constA);
         PH.SetACX(5L, 3L, 2L);                        // congruential parameters
         PH.PhaseStart();                                // phase generator start
         double[] s0 = new double[NS];                          // initial signal s0
         double t = 0.0;                          // the beginning of signal counters
         for (int i = 0; i < NS; i++)
         {  s0[i] = PH.PhaseSinNext(t);                        // counter value
            t += PH.dxs;                                  // next counter time
         }
         t = 0.0;                                        // next signal beginning
         double[,] z = new double[NS, NS];               // autovector matrix z
         MatrixZ(z, s0, PH, NS, t);
         double[,] v = new double[NS, NS];               // autovector matrix v
         MatrixV(v, z, NS);                       // autocorrelation matrix A
         double[,] a = new double[NS, NS];
         MatrixA(a, v, NS);
         Console.WriteLine("A = ");
         for (int i = 0; i < NS; i++)
         {  Console.Write("{0,8:F3}", a[i, 0]);
            Console.Write("{0,8:F3}", a[i, 1]);
            Console.Write("{0,8:F3}", a[i, 2]);
            Console.Write(" - - - -");
            Console.Write("{0,8:F3}", a[i, NS - 2]);
            Console.WriteLine("{0,8:F3}", a[i, NS - 1]);
         }                                  // autocorrelation coefficient matrix R
```

```
            double[,] r = new double[NS, NS];
            MatrixR(r, a, v, NS);
            Console.WriteLine("R = ");
            for (int i = 0; i < NS; i++)
            {   Console.Write("{0,8:F3}", r[i, 0]);
                Console.Write("{0,8:F3}", r[i, 1]);
                Console.Write("{0,8:F3}", r[i, 2]);
                Console.Write(" - - - -");
                Console.Write("{0,8:F3}", r[i, NS - 2]);
                Console.WriteLine("{0,8:F3}", r[i, NS - 1]);
            }
            Console.ReadKey();                                          // result viewing
        }
    //-----------------------------------------------------------------
         static void MatrixA(double[,] a, double[,] d, long ns)
                                 double[,] d, long ns)
        {   for (int i = 0; i < ns; i++)
                for (int j = i; j < ns; j++)
                {   double iE2 = 0.0;
                    double jE2 = 0.0;
                    for (int m = 0; m < ns; m++)
                    { iE2 += d[i, m] * d[i, m];
                      jE2 += d[j, m] * d[j, m];
                    }
                    r[i, j] = a[i, j] / Math.Sqrt(iE2 * jE2);
                    r[j, i] = r[i, j];
                }
        }
    //-----------------------------------------------------------------
         static void MatrixA(double[,] a, double[,] d, long ns)
        {   for (int i = 0; i < ns; i++)
                for (int j = i; j < ns; j++)
                {   a[i, j] = 0.0;
                    for (int m = 0; m < ns; m++)
                        a[i, j] += d[i, m] * d[j, m];
                        a[j, i] = a[i, j];
                }
        }
    //-----------------------------------------------------------------
     Function MatrixV from previous program P070402
    //-----------------------------------------------------------------
     Function MatrixZ from previous program P070401
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      }
}
```

After launching the program *P070403* the autocorrelation matrix *A* and the matrix *R* of the autocorrelation coefficients appear. The omitted values of the listing below are substituted by a dash.

```
NS = 33    NF = 16
A =
   129.360  −7.555   −8.062  - - - -  −10.614    −8.202
  −7.555   129.224   −4.349  - - - -  −11.034   −10.667
  −8.062   −4.349   136.564  - - - -  −6.537    −8.376
```

```
 −8.394   −8.227   −1.509  - - - -    −5.415    −6.619
 −9.268   −7.316   −3.298  - - - -    −2.212    −4.479
 - - - - -
 −7.584   −4.653    1.473  - - - -    −2.729    −5.097
 −8.709   −8.208   −2.251  - - - -    −3.444    −3.180
 −9.999   −8.682   −4.441  - - - -    −0.558    −3.358
−10.614  −11.034   −6.537  - - - -   136.653    −1.339
 −8.202  −10.667   −8.376  - - - -    −1.339   136.656
R =
  1.000   −0.058   −0.061  - - - -    −0.080    −0.062
 −0.058    1.000   −0.033  - - - -    −0.083    −0.080
 −0.061   −0.033    1.000  - - - -    −0.048    −0.061
 −0.063   −0.062   −0.011  - - - -    −0.040    −0.048
 −0.069   −0.055   −0.024  - - - -    −0.016    −0.033
 - - - - -
 −0.056   −0.035    0.011  - - - -    −0.020    −0.037
 −0.065   −0.061   −0.016  - - - -    −0.025    −0.023
 −0.075   −0.065   −0.032  - - - -    −0.004    −0.024
 −0.080   −0.083   −0.048  - - - -     1.000    −0.010
 −0.062   −0.080   −0.061  - - - -    −0.010     1.000
```

The analysis of the results received above shows that even such a limited listing provides evidence that the resulting matrix R is closer to the statistical independence of white noise $R = I$ than the same matrix obtained earlier in the program *P070101* using the standard function *Random.Next()*. At the same time, the main advantage of this outcome is that the developed congruential phase generator in the program *P070403* creates anew the Fourier frequency spectrum with equal intensities at the stochastic phases. The data presented in the last listing demonstrate that the uniform white noise was indeed achieved with an almost independent autocorrelation matrix for the autovectors of the original signal.

5. Discussion

After obtaining the correlation matrices, the first thing which should be analyzed is whether both experiments in the programs *P070101* and *P070403* ensure the realization of the first fundamental property of uniform white noise (considered here earlier in subsection *White Noise Autocorrelation Matrix*) by the equality of all intensities of the internal Fourier spectrum. Let us discuss this issue further in more detail.

When considering the frequency properties of discrete information signals, usually the Fourier polynom (11) is used, with the number of countings N_S and the quantity of internal frequencies N_F in the original signal S_0. The amplitudes of the cosine a_k and sine b_k components are calculated from the meanings of countings $f(x_i)$ using the following Euler–Fourier formulas below:

$$a_0 = \frac{1}{N_S} \sum_{i=0}^{N_S-1} f(x_i), \tag{38}$$

$$a_k = \frac{2}{N_S} \sum_{i=0}^{N_S-1} f(x_i) \cos k\omega_1 x_i, \tag{39}$$

$$b_k = \frac{2}{N_S} \sum_{i=0}^{N_{S-1}} f(x_i) \sin k\omega_1 x_i \tag{40}$$

The process of white noise generation could be considered successful if the obtained countings $f(x_i)$ of the signal admit transformations (38)–(40) into the Euler–Fourier coefficients a_k and b_k. It should be checked now whether the intensities A_k are the same at all

frequencies $\omega_k = k\omega_1$. This is required by the first property in the designation of white noise, i.e., the demand of uniform distribution of intensities at all internal frequencies of the signal. This check should be carried out both for the white noise of the function *Random.Next()* in the program *P070101* and for the congruential white noise in the program *P070401*.

Below is the program *P070501*, which uses the random process generated earlier in the program *P070101* for the white noise signal S_0 using the function *Random.Next()*. The derivable spectral amplitudes A_k are calculated with help from sine technique using the function *Fourier()*, which is composed according to the Euler–Fourier Formulas (38)–(40) for the coefficients a_k and b_k.

```
namespace P070501
{  class cP070501
   {  static void Main(string[] args)
      {  const int NS = 33;                                  // signal counter quantity
         const int NF = 16;                                  // frequency quantity in a counter
         Console.WriteLine("NS = {0} NF = {1}", NS, NF);
         double[] s0 = new double[NS]
         {
           0.905,    1.269,   1.072,    0.233, −1.176,    0.236,
           1.624,  −0.231,    1.910,  −0.905, −0.832,  −0.131,
           0.531,  −0.122,    1.929,  −1.879,   1.449,  −1.981,
           0.709,  −0.742,    1.268,    1.392,   1.968,  −1.869,
           0.800,    0.105,   1.736,    0.750,   0.187,  −1.676,
         −1.252,  −0.187,  −0.811,
         };
         Console.WriteLine("S0 = ");
         for (int i = 1; i <= NS; i++)
         {  Console.Write("{0,8:F3}", s0[i-1]);
            if (i % 6 == 0) Console.WriteLine();
         }
         Console.WriteLine();
         double[] AF = new double[NF + 1];
         double[] phiF = new double[NF + 1];
         Fourier(NS, NF, s0, AF, phiF);
         Console.WriteLine("AFourier =");
         for (int i = 1; i <= NF + 1; i++)
         {  Console.Write("{0,8:F3}", AF[i - 1]);
            if (i % 6 == 0) Console.WriteLine();
         }
         Console.WriteLine();
         Console.ReadKey();                                  // result viewing
      }
//--------------------------------------------------------------------
       static void Fourier(int NS, int NF, double[] s,
                           double[] AF, double[] phiF)
      {  double a, b;
         for (long k = 1; k <= NF; k++)
         {  a = 0.0;                                         // cosine coefficients
            b = 0.0;                                         // sine coefficients
            double w1 = 1.0;
            double dx = 2.0 * Math.PI / NS;
            for (long i = 0; i < NS; i++)
            {  double x = i * dx;
```

```
                a += s[i] * Math.Cos(k * w1 * x);
                b += s[i] * Math.Sin(k * w1 * x);
            }
            a = a * 2.0 / NS;
            b = b * 2.0 / NS;
            AF[k] = Math.Sqrt(a * a + b * b);
            phiF[k] = Math.Asin(a / AF[k]);
        }
    }
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  }
}
```

After executing the program *P070501* the following outcome shows up.

```
NS = 33   NF = 16
S0 =
   0.905    1.269   1.072    0.233  −1.176    0.236
   1.624  −0.231    1.910  −0.905  −0.832  −0.131
   0.531  −0.122    1.929  −1.879    1.449  −1.981
   0.709  −0.742    1.268    1.392    1.968  −1.869
   0.800    0.105   1.736    0.750    0.187  −1.676
 −1.252  −0.187  −0.811
AFourier =
   0.000    0.283   0.406    0.221    0.339    0.638
   0.309    0.593   0.406    0.338    0.309    0.213
   0.580    0.310   0.265    0.338    0.081    0.717
```

The listing of the results of this program begins with specifying the number of countings $NS = 33$ and Fourier frequencies $NF = 16$. The values of countings are taken for the original signal S_0 from the result of the program *P070101* in the subsection *Introduction*. Then there are lines *AFourier* with the amplitudes of the internal phase sine frequencies. Their analysis suggests that the function *Random.Next()* does not satisfy the first property about equality of the amplitudes of the internal frequency spectrum of white noise. Thus, taking into account the limited level of the corresponding matrix of autocorrelation coefficients R (presented in the subsection *Introduction*), and also the lack of equality of amplitudes in the internal spectrum of the signal frequencies, it becomes apparent that the standard function *Random.Next()* generates sequences which are relatively far from satisfactory quality of the white noise process.

Next, it is time to check the amplitudes of the internal frequency spectrum of the congruential white noise generator, which has been proposed in the current article. Below is the program *P070502*, which also uses the same number of N_S countings. They were created earlier in the program *P070401* (in subsection *Construction and Results*) by using congruential technology [45–50] in the generator *cDeonYuliCongPhase62A*.

```
namespace P070502
{  class cP070502
   {  static void Main(string[] args)
      {  const long NS = 33L;                              // signal counter quantity
         double dNS = (double)NS;
         const long NF = 16L;                          // frequency quantity in a counter
         Console.WriteLine("NS = {0} NF = {1}", NS, NF);
         double[] s0 = new double[]
         {
```

```
            −0.371,    6.938, −0.398,  2.317, −0.415,    1.159,
            −0.540,    0.626, −0.283,  0.545, −0.396,    0.479,
            −0.117,    0.736,  0.200,  0.980,  0.238,    0.919,
             0.215,    0.836, −0.125,  0.483, −0.566,    0.146,
            −0.779,    0.226, −1.147, −0.087, −1.497, −0.077,
            −2.373, −0.201, −7.674
        };
        Console.WriteLine("S0 =");
        for (int i = 1; i <= NS; i++)
        {  Console.Write("{0,8:F3}", s0[i - 1]);
           if (i % 6 == 0) Console.WriteLine();
        }
        Console.WriteLine();
        double[] AF = new double[NF + 1];
        double[] phiF = new double[NF + 1];
        int NNS = (int)NS;
        int NNF = (int)NF;
        Fourier(NNS, NNF, s0, AF, phiF);
        Console.WriteLine("AFourier =");
        for (int i = 1; i <= NF; i++)
        {  Console.Write("{0,8:F3}", AF[i]);
           if (i % 6 == 0) Console.WriteLine();
        }
        Console.WriteLine();
        Console.WriteLine("psiFourier =");
        for (int i = 1; i <= NF; i++)
        {  Console.Write("{0,8:F3}", phiF[i]);
           if (i % 6 == 0) Console.WriteLine();
        }
        Console.WriteLine();
        Console.ReadKey();                                    // result viewing
    }
//-------------------------------------------------------------------
 Function Fourier from previous program P070501
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  }
}
```

After launching the program *P070502* the following outcome shows up as well.

```
NS = 33    NF = 16
S0 =
  −0.371    6.938 −0.398  2.317 −0.415    1.159
  −0.540    0.626 −0.283  0.545 −0.396    0.479
  −0.117    0.736  0.200  0.980  0.238    0.919
   0.215    0.836 −0.125  0.483 −0.566    0.146
  −0.779    0.226 −1.147 −0.087 −1.497 −0.077
  −2.373 −0.201 −7.674
AFourier =
   0.700    0.700    0.700    0.700    0.700    0.700
   0.700    0.700    0.700    0.700    0.700    0.700
   0.700    0.700    0.700    0.700    0.700    0.700
psiFourier =
  −0.589    0.245 −0.131 −0.025 −0.039 −0.115
   0.000    0.037    0.022 −0.029    0.036    0.057
```

$$0.045 \quad 0.007 \quad -0.052 \quad -0.031$$

The listing of the results demonstrates that the signal contains $N_s = 33$ countings and $N_F = 16$ Fourier frequencies. The values of countings are in the array *s0*, which are taken from the signal S_0 obtained earlier (subsection *Construction and Results*) using the congruential phase generator *cDeonYuliCongPhase62A* in the program *P070401*. Further, the listing includes the lines *AFourier* with the derived intensities A_k of the internal phase sine frequencies. These intensities were calculated using the Euler–Fourier Formulas (38)–(40) with the subsequent application of the elementary transformation $A_k = \sqrt{a_k^2 + b_k^2}$. All meanings received with the values 0.7 finely match to the first property of the uniform white noise process. In the last part of this listing, the phases *psiFourier* show coincidence with the congruential phases in program *P070403*. It should also be noted that the application of the Euler–Fourier transform (38)–(40) completely recovers the generation of white noise process received in the class *cDeonYuliCongPhase62A*.

Careful analysis of all results above satisfies that the generator *cDeonYuliCongPhase62A* does indeed provide the equal amplitudes of all internal phase frequencies, and that it ideally satisfies to the first property of the uniform white noise process. Thus, taking into account the better approximation of the matrix R of autocorrelation coefficients obtained in the program *P070403* to the same matrix of theoretical white noise, and also taking into consideration an ideal coincidence of the intensities of the internal phase frequencies, it should be recognized that the here proposed congruential phase generator does indeed ensure a sufficiently high quality of generation of the white noise signals, which closely approximate true natural white noise.

6. Conclusions

Analysis of the source material shows that the algorithms of the commonly used generators of white noise signals have a low stochasticity of countings in the given observation intervals. Based on this, in this article the instrumental algorithmic tools for generating statistically independent white noise signals have been proposed. The designed techniques allowed for the creation of a new phase signal generator with an improved matrix of autocorrelation coefficients. The mathematical expressions used confirm that at Fourier frequencies a single dimensional phase random variable could be obtained. As a result, the derivative *cDeonYuliCongPhase62A* phase generator made it possible to create information signals with a better approximation to the uniform white noise process. The simulation outcomes verify that the information signals received have the properties of white noise signals with equal amplitudes at all internal frequencies with uniformly distributed random phases. These results can be used in a huge number of applications where white noise processes are used.

Author Contributions: All the authors equally contributed to this work. All authors have read and agreed to the published version of the manuscript.

Funding: The authors have no support or funding to report.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: The authors are thankful to Robert Weingold and Julia Alex Watts (University of Arkansas for Medical Sciences, Little Rock, AR, USA) for the proofreading.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Hida, T. *Stochastic Analysis: Classical and Quantum: Perspectives of White Noise Theory*; World Scientific Publishing Company: Singapore, 2005; p. 300, ISBN 10:9812565264.

2. Hida, T.; Si, S. *Lectures on White Noise Functionals*; World Scientific Publishing Company: Singapore, 2008; p. 266, ISBN 10:9812560521.
3. Bernido, C.C.; Carpio, M.V. *Methods and Applications of White Noise Analysis in Interdisciplinary Sciences*; World Scientific Publishing Company: Singapore, 2014; p. 204. ISBN 10:9814569119.
4. Hida, T.; Streit, L. *Let Us Use White Noise*; World Scientific Publishing Company: Singapore, 2017; p. 232. ISBN 10:9813220937.
5. Howard, R.M. White noise: A time domain basis. In Proceedings of the 2015 International Conference on Noise and Fluctuations (ICNF), Xi'an, China, 2–6 June 2015; pp. 1–4. [CrossRef]
6. Hakeem, A.O. The q-Gamma white noise. *Tatra Mt. Math. Publ.* **2016**, *66*, 81–90. [CrossRef]
7. Suryawan, H.P. Gaussian white noise analysis and its application to Feynman path integral. *AIP Conf. Proc. 1707* **2016**, 030001. [CrossRef]
8. Balan, M.R.; Ndongo, C.B. Malliavin differentiability of solutions of SPDEs with Lévy white noise. *Int. J. Stoch. Anal.* **2017**, *2017*, 1–9. [CrossRef]
9. Croci, M.; Giles, M.B.; Rognes, M.E.; Farrell, P.E. Efficient white noise sampling and coupling for multilevel Monte Carlo with nonnested meshes. *SIAM ASA J. Uncertain. Quantif.* **2018**, *6*, 1630–1655. [CrossRef]
10. Zhu, D.; Beeby, S.P. A broadband electromagnetic energy harvester with a coupled bistable structure. *J. Phys.* **2013**, *476*, 012070. [CrossRef]
11. Kang, Y.; Belusic, D.; Smith-Miles, K. Detecting and classifying events in noisy time series. *J. Atmos. Sci.* **2014**, *71*, 1090–1104. [CrossRef]
12. Mitsuya, H.; Ashizawa, H.; Homma, H.; Hashiguchi, G.; Toshiyoshi, H. A method to determine the electret charge potential of MEMS vibrational energy harvester using pure white noise. In Proceedings of the 2019 IEEE 32nd International Conference on Microelectronic Test Structures (ICMTS), Fukuoka, Japan, 18–21 March 2019; pp. 171–174. [CrossRef]
13. Préaux, Y.; Boudraa, A. Statistical behavior of Teager-Kaiser energy operator in presence of white gaussian noise. *IEEE Signal Process. Lett.* **2020**, *27*, 635–639. [CrossRef]
14. Pralgauskaitė, S.; Palenskis, V.; Matukas, J.; Seliuta, D.; Kašalynas, I.; Valušis, G. White noise peculiarities in diode structures. In Proceedings of the 22nd International Conference on Noise and Fluctuations (ICNF), Montpellier, France, 24–28 June 2013; pp. 1–4. [CrossRef]
15. Paik, H.; Sastry, N.N.; SantiPrabha, I. Effectiveness of noise jamming with white gaussian noise and phase noise in amplitude comparison monopulse radar receivers. In Proceedings of the 2014 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT), Bangalore, India, 6–7 January 2014; pp. 1–5. [CrossRef]
16. Arslan, S.; Yildirim, B.S. A broadband microwave noise generator using zener diodes and a new technique for generating white noise. *IEEE Microw. Wirel. Compon. Lett.* **2018**, *28*, 329–331. [CrossRef]
17. Takada, A. White noise spectra obtained in a phase-locked loop operating like a Josephson junction. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 26–29 May 2019; pp. 1–5. [CrossRef]
18. Shi, X.; Cai, L.; Wang, G.; Liang, L. A new aircraft taxiing model based on filtering white noise method. *IEEE Access* **2020**, *8*, 10070–10087. [CrossRef]
19. Shen, Z.; Wu, Y. Mean square stabilization of multi-input discrete-time systems over stochastic multiplicative and additive white gaussian noise channels. *IEEE Access* **2020**, *8*, 111791–111801. [CrossRef]
20. Yildirim, A.; Kiranyaz, S. 1D convolutional neural networks versus automatic classifiers for known LPI radar signals under white gaussian noise. *IEEE Access* **2020**, *8*, 180534–180543. [CrossRef]
21. Oh, H.; Nam, H. Maximum rate scheduling with adaptive modulation in mixed impulsive noise and additive white gaussian noise environments. *IEEE Trans. Wirel. Commun.* **2021**, 1–13. [CrossRef]
22. Mukherjee, A.; Mandal, S.; Ghosh, D.; Biswas, B.N. Influence of additive white gaussian noise on the OEO output. *IEEE J. Quantum Electron.* **2021**, *57*, 1–10. [CrossRef]
23. Ohmori, K.; Amakawa, S. Direct white noise characterization of short-channel MOSFETs. *IEEE Trans. Electron. Devices* **2021**, 1–5. [CrossRef]
24. Stansfeld, S.A.; Berglund, B.; Clark, C.; Lopez-Barrio, I.; Fischer, P.; Ohrström, E.; Haines, M.M.; Head, J.; Hygge, S.; van Kamp, I.; et al. Aircraft and road traffic noise and children's cognition and health: A cross-national study. *Lancet* **2005**, *365*, 1942–1949. [CrossRef]
25. Prochnik, G. *Pursuit of Silence: Listening for Meaning in a World of Noise*, Reprint ed.; Anchor Books: New York, NY, USA, 2011; p. 352. ISBN 10:0767931211.
26. Jespers, P.G.A. *Integrated Converters: D to A and A to D Architectures, Analysis and Simulation*, Illustrated ed.; Oxford University Press: Oxford, UK, 2001; p. 280. ISBN 10:0198564465.
27. Olano, M. Modified noise for evaluation on graphics hardware. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware HWWS'05, Los Angeles, CA, USA, 30–31 July 2005; pp. 105–110. [CrossRef]
28. Sun, X. Optimal weighted state fusion white noise deconvolution estimator. In Proceedings of the 2013 International Conference on Advanced Mechatronic Systems, Luoyang, China, 25–27 September 2013; pp. 46–50. [CrossRef]
29. Liu, W.; Deng, Z. Robust weighted fusion white noise deconvolution estimators with uncertain-variance linearly correlated white noises and missing measurements. In Proceedings of the 20th International Conference on Information Fusion, Xi'an, China, 10–13 July 2017; pp. 1–8. [CrossRef]

30. Menyaev, Y.A.; Zharov, V.P. Experience in development of therapeutic photomatrix equipment. *Biomed. Eng.* **2006**, *40*, 57–63. [CrossRef]
31. Menyaev, Y.A.; Zharov, V.P. Experience in the use of therapeutic photomatrix equipment. *Biomed. Eng.* **2006**, *40*, 144–147. [CrossRef]
32. Menyaev, Y.A.; Nedosekin, D.A.; Sarimollaoglu, M.; Juratli, M.A.; Galanzha, E.I.; Tuchin, V.V.; Zharov, V.P. Optical clearing in photoacoustic flow cytometry. *Biomed. Opt. Express* **2013**, *4*, 3030–3041. [CrossRef]
33. Menyaev, Y.A.; Carey, K.A.; Nedosekin, D.A.; Sarimollaoglu, M.; Galanzha, E.I.; Stumhofer, J.S.; Zharov, V.P. Preclinical photoacoustic models: Application for ultrasensitive single cell malaria diagnosis in large vein and artery. *Biomed. Opt. Express* **2016**, *7*, 3643–3658. [CrossRef]
34. Wang, X.; Li, Y.; Wang, X. The stochastic stability of internal HIV models with gaussian white noise and gaussian colored noise. *Discret. Dyn. Nat. Soc.* **2019**, *2019*, 1–8. [CrossRef]
35. Karaduta, O.; Deon, A.; Menyaev, Y. Designing the uniform stochastic photomatrix therapeutic systems. *Algorithms* **2020**, *13*, 41. [CrossRef]
36. Karaduta, O.; Zaman, L. Shk-9: A new tool in approach of glycoprotein annotation. *SoftwareX* **2018**, *7*, 302–303. [CrossRef]
37. Söderlund, G.B.; Sikström, S.; Loftesnes, J.M.; Sonuga-Barke, E.J. The effects of background white noise on memory performance in inattentive school children. *Behav. Brain Funct.* **2010**, *6*, 55. [CrossRef]
38. Szalma, J.L.; Hancock, P.A. Noise effects on human performance: A meta-analytic synthesis. *Psychol. Bull.* **2011**, *137*, 682–707. [CrossRef] [PubMed]
39. Deng, Z.L.; Zhang, H.S.; Liu, S.J.; Zhou, L. Optimal and selftuning white noise estimators with approach to deconvolution and filtering problem. *Automatica* **1996**, *32*, 199–216. [CrossRef]
40. Ditlevsen, P.D.; Andersen, K.K.; Svensson, A. The DO-climate events are probably noise induced: Statistical investigation of the claimed 1470 years cycle. *Clim. Past* **2007**, *3*, 129–134. [CrossRef]
41. Wikipedia. White Noise. Available online: https://en.wikipedia.org/wiki/White_noise (accessed on 17 March 2021).
42. Maurer, U.M. A universal statistical test for random bit generators. *J. Cryptol.* **1992**, *5*, 89–105. [CrossRef]
43. Kolmogorov, A.N.; Fomin, S.V. *Elements of the Theory of Functions and Functional Analysis*; Dover Publication: Mineola, NY, USA, 1999; p. 128. ISBN 10:0486406830.
44. Gnedenko, B. *Theory of Probability*, 6th ed.; CRC Press: Boca Raton, FL, USA, 2020; p. 520. ISBN 10:0367579316.
45. Deon, A.; Menyaev, Y. The complete set simulation of stochastic sequences without repeated and skipped elements. *J. Univ. Comput. Sci.* **2016**, *22*, 1023–1047. [CrossRef]
46. Deon, A.; Menyaev, Y. Parametrical tuning of twisting generators. *J. Comput. Sci.* **2016**, *12*, 363–378. [CrossRef]
47. Deon, A.; Menyaev, Y. Twister generator of arbitrary uniform sequences. *J. Univ. Comput. Sci.* **2017**, *23*, 353–384. [CrossRef]
48. Deon, A.; Menyaev, Y. Uniform twister plane generator. *J. Comput. Sci.* **2018**, *14*, 260–272. [CrossRef]
49. Deon, A.; Menyaev, Y. Poisson twister generator by cumulative frequency technology. *Algorithms* **2019**, *12*, 114. [CrossRef]
50. Deon, A.; Menyaev, Y. Twister generator of random normal numbers by Box-Muller model. *J. Comput. Sci.* **2020**, *16*, 1–13. [CrossRef]