



Article Poisson Twister Generator by Cumulative Frequency Technology

Aleksei F. Deon ¹ and Yulian A. Menyaev ^{2,*}

- ¹ Department of Information Systems and Computer Science, Bauman Moscow State Technical University, 2nd Baumanskaya St., 5/1, 105005 Moscow, Russia; DeonAlex@mail.ru
- ² Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Sciences, 4301 W. Markham St., Little Rock, AR 72205, USA
- * Correspondence: YAMenyaev@uams.edu

Received: 6 April 2019; Accepted: 25 May 2019; Published: 28 May 2019



Abstract: The widely known generators of Poisson random variables are associated with different modifications of the algorithm based on the convergence in probability of a sequence of uniform random variables to the created stochastic number. However, in some situations, this approach yields different discrete Poisson probability distributions and skipping in the generated numbers. This article offers a new approach for creating Poisson random variables based on the complete twister generator of uniform random variables, using cumulative frequency technology. The simulation results confirm that probabilistic and frequency distributions of the obtained stochastic numbers completely coincide with the theoretical Poisson distribution. Moreover, combining this new approach with the tuning algorithm of basic twister generation allows for a significant increase in length of the created sequences without using additional RAM of the computer.

Keywords: pseudorandom number generator; stochastic sequences; Poisson distribution; twister generator

1. Introduction

Using generators of Poisson random variables realizes a stochastic process of creating integer random numbers $\eta \in H$ having the following probability distribution with respect to the real parameter α [1,2]:

$$P(\eta, \alpha) = \frac{\alpha^{\eta}}{\eta!} e^{-\alpha},\tag{1}$$

where η takes any integer values such as $0, 1, 2, ..., \infty$.

The Poisson model usually describes a scheme of rare events: under certain assumptions about the nature of a process with random events, the number of elements observed over a fixed time interval or in a fixed region of space is often a subject of Poisson distribution. Examples include the number of particles of radioactive decay registered by a counter for some period of time, the number of calls received to a telephone switching exchange during the designated time, the number of defects in a piece of cloth or in a tape of fixed length, etc. Thus, Poisson distribution simulates a random variable that represents the number of events that occurred over a fixed time. These events happened with some fixed average intensity and independently of each other. At the same time, Poisson distribution is discrete, which is one of the important limiting cases of a binomial distribution. Therefore, it gives a good approximation of a binomial distribution for both small and large values. In this case, Poisson distribution is intensively used in quality control cards, queuing theory, telecommunications, etc.

Poisson distribution has the remarkable properties of initial probabilistic moments with respect to mathematical expectation $m(\eta, \alpha) = E_1(\eta, \alpha) = \alpha$ and to dispersion $D(\eta, \alpha) = E_2(\eta, \alpha) = E_1(\eta^2, \alpha) = \alpha$. These and other properties make it possible to use Poisson distribution in theoretical and statistical

mathematics [3,4], in the study of physical phenomena [5,6], in radio engineering and nuclear physics [7,8], in informatics and information systems [9], in modeling of data transmission and networks [10,11], in economics and financial analysis [12], and in other areas up to biological studies [13–16] and research for medical physics and technics [17–19].

There are various methods for implementing the pseudorandom number generators based on Poisson distribution. The generator proposed by Knuth [20,21] is widely known and actively used by his followers. In Wikipedia [22] it is presented in the following form in an arbitrary pseudo-code language, where parameter α has the notation λ , and the random variable η is equal to k: *algorithm poisson random number (Knuth)*:

init: Let $L \leftarrow e - \lambda$, $k \leftarrow 0$ and $p \leftarrow 1$. do: $k \leftarrow k + 1$.

Generate uniform random number u in [0,1] and let $p \leftarrow p \times u$. while p > L.

return k - 1.

Below is the program code in C# for *Microsoft Visual Studio*. According to Equation (1), instead of λ , the parameter α is assigned here, which can be designated by any arbitrary value, for example $\alpha = 2.0$. The function *KnuthPoisson()* creates stochastic integer numbers *k*, which are analogous to the random variables η in (1). Their frequency distribution is stored in an array *nuK*. The function *Random.Next()* is used as a generator of uniform random variables. The function *PoissonP()* places the probabilities $P(\eta, \alpha)$ of stochastic events in an array *pEta*. To form the distribution, a total $N = 2^w = 2^{16} = 65536$ of uniform random variables are applied. Program names *P060102* and *cP060102* are chosen arbitrarily.

```
namespace P060102
{ class cP060102
{
  static uint gc = 0;
                           //quantity of uniform generation
   static void Main(string[] args)
   { int w = 16; //bit width of uniform integer variable
      long N = 1L \ll w;
                                //random variable quantity
      Console.WriteLine("w = \{0\} N = \{1\}", w, N);
      double Alpha = 2.0;
      Console.WriteLine("Alpha = {0:F2}", Alpha);
      Random G = new Random();
                                 //uniform generator p
      int wX = 200;
      int[] nuK = new int[wX];
                                          //Knuth frequency
      for (int i = 0; i < wX; i++) nuK[i] = 0;
      int maxK = 0;
                                      //distribution length
      for (int i = 0; i < N; i++)
        int k = KnuthPoisson(Alpha, N, G);
      {
         nuK[k]++;
                                          //Knuth frequency
         if (k > maxK) maxK = k;
                                     //distribution length
      }
      Console.WriteLine("maxK = {0}", maxK);
      double[] pEta = new double[wX];
                                              //probability
      long[] nuEta = new long[wX];
                                    //Poisson frequency
      int cEta = PoissonP(Alpha, N, pEta, nuEta);
      VerifyProbability(N, cEta, pEta, nuEta);
      Console.WriteLine("cEta = {0} ", cEta);
      long snuEta = 0;
                                //sum of Poisson frequency
      double spEta = 0.0; //sum of Poisson probability
```

```
int snuK = 0;
                                //sum of Knuth frequency
     double spK = 0.0;
                              //sum of Knuth probability
                          рК
     Console.Write("Eta
                                         nuK");
     Console.Write("
                         pEta
                                     nuEta");
     Console.WriteLine("
                           nuK - nuEta");
     int nEta = cEta > maxK ? cEta : maxK;
     for (int i = 0; i \le nEta; i++)
                                    //frequency tabling
     { double pK = (double)nuK[i]/(double)N;
        int dnu = (int)(nuK[i] - nuEta[i]);
        Console.Write("{0,2} {2,12:F10} {1,10}",
                      i, nuK[i], pK);
        Console.Write(" {0,12:F10} {1,10}",
                     pEta[i], nuEta[i]);
        Console.WriteLine(" {0,8}", dnu);
        snuK += nuK[i];
                               //sum of Knuth frequency
        spK += pK;
                              //sum of Knuth probability
        snuEta += nuEta[i];
                             //sum of Poisson frequency
        spEta += pEta[i];
                          //sum of Poisson probability
     }
     Console.Write("sum
                         spK
                                        snuK");
     Console.WriteLine("
                            spEta
                                        snuEta");
     Console.Write("
                     {0,12:F10} {1,10}",
                   spK, snuK);
     Console.WriteLine(" {0,12:F10} {1,10}",
                      spEta, snuEta);
     Console.WriteLine("gc = {0}", gc);
     Console.ReadKey();
                                       //result viewing
  }
//-----
  static int KnuthPoisson(double Lam, long N, Random G)
  { double L = Math.Exp(-Lam);
     double p = 1.0;
     double dN = (double)N;
     int k = 0;
     do
     { k++;
        long z = (long)G.Next(); //uniform variable
        z = z \& (N - 1);
        double u = (double)z/dN;
        p = p * u;
                    //global number of uniform generation
        gc++;
     } while (p > L);
     return k - 1;
  }
//-----
  static int PoissonP(double alpha, long N,
                     double[] pEta, long[] nuEta)
  { double emAlpha = Math.Exp(-alpha);
     double spEta = 0.0;
                                      //probability sum
     long snuEta = 0L;
                                        //frequency sum
```

} }

```
pEta [0] = 1.0 * emAlpha;
                               //Poisson probability p(0)
     spEta += pEta[0];
                                      //probability sum
     nuEta[0] = (long)Math.Round(pEta[0] * (double)N);
     snuEta += nuEta[0];
                                        //frequency sum
                                 //Tailor first summand
     double r = alpha;
     pEta[1] = r * emAlpha; //Poisson probability p(1)
     spEta += pEta[1];
                                      //probability sum
     nuEta[1] = (long)Math.Round(pEta[1] * (double)N);
     snuEta += nuEta[1];
                                        //frequency sum
     int Eta = 2;
                                 //random variable value
     do
     { r *= alpha/(double)Eta; //regular summand of exp
        double p = r * emAlpha; //probability p(Eta)
        long nu = (long)Math.Round(p * (double)N);
        long sd = snuEta + nu;
        if (nu == 0L || sd > N) break;
                                             //the tail
        pEta[Eta] = p;
                                   //probability p(Eta)
        spEta += p;
                                      //probability sum
        nuEta[Eta] = nu;
                                     //frequency nu(Eta)
        snuEta += nu;
                                        //frequency sum
        Eta++;
                             //next random variable Eta
     } while (snuEta < N);</pre>
     long d = N - snuEta;
                                   //tailing frequencies
     if (d == 0L) return Eta - 1;
     double d1N = (1.0 - spEta)/(double)d;
     do
     { pEta[Eta] = d1N;
                            //the tail event probability
        nuEta[Eta] = 1;
                                       //one-part event
        snuEta++;
                                        //frequency sum
        Eta++;
     } while (snuEta < N);</pre>
     return Eta - 1;
  }
//-----
  static void VerifyProbability(long N, int cEta,
                              double[] pEta, long[] nuEta)
  { double dN = (double)N;
     for (int i = 0; i \le cEta; i++)
        pEta[i] = (double)nuEta[i]/dN;
  }
//~~~~~~
```

After starting the program *P060102*, the following result may be seen on a monitor.

```
w = 16 N = 65536
Alpha = 2.00
maxK = 12
cEta = 10
Eta
                             pEta
                                        nuEta nuK – nuEta
    рК
                   nuK
```

| | 0 | 0.1352539063 | 8864 | 0.1353302002 | 8869 | -5 | |
|-------------|-----|--------------|-------|--------------|--------|------|--|
| | 1 | 0.2691345215 | 17638 | 0.2706756592 | 17739 | -101 | |
| | 2 | 0.2699890137 | 17694 | 0.2706756592 | 17739 | -45 | |
| | 3 | 0.1814422607 | 11891 | 0.1804504395 | 11826 | 65 | |
| | 4 | 0.0910491943 | 5967 | 0.0902252197 | 5913 | 54 | |
| | 5 | 0.0361175537 | 2367 | 0.0360870361 | 2365 | 2 | |
| | 6 | 0.0125427246 | 822 | 0.0120239258 | 788 | 34 | |
| | 7 | 0.0032806396 | 215 | 0.0034332275 | 225 | -10 | |
| | 8 | 0.0008850098 | 58 | 0.0008544922 | 56 | 2 | |
| | 9 | 0.0002441406 | 16 | 0.0001983643 | 13 | 3 | |
| | 10 | 0.0000457764 | 3 | 0.0000457764 | 3 | 0 | |
| | 11 | 0.0000000000 | 0 | 0.0000000000 | 0 | 0 | |
| | 12 | 0.0000152588 | 1 | 0.0000000000 | 0 | 1 | |
| | sum | spK | snuk | spEta | snuEta | | |
| | | 1.0000000000 | 65536 | 1.0000000000 | 65536 | | |
| gc = 197025 | | | | | | | |

In this listing, the columns *pK* and *nuK* show the values for frequency and probability obtained by Knuth algorithm. These values correspond well to the analogous ones in the columns *pEta* and *nuEta*, which are calculated by Poisson model (1). However, there are some peculiarities that should be addressed here.

The first drawback is due to the fact that string 11 shows 0 by Knuth algorithm. This means that the generator did not create the 11th random variable, although it did create 12th one. This case points out the skipping of the 11th variable. In the theoretical Poisson distribution, this situation is not allowed. For some applications that are not limited to strict constraints, this could be neglected. However, if the generator is used for the comprehensive modeling of real stochastic situations, it is better to avoid this.

The second disadvantage is apparent if Knuth generator is launched repeatedly. The monitoring of the values in columns pK and nuK registers their inconsistency. In probability theory by Kolmogorov axiomatics [2,23] a change in the probabilistic measure in a given space under any circumstances is categorically prohibited. The disturbance of the axioms of a space can make it difficult to interpret the results. This may be crucial when repeated tests are required, for example, in emergency situations, i.e. when it is necessary to reiterate the special cases.

The third limitation of Knuth algorithm is that counter gc, which summarizes the number of generated uniform random variables, turned out to be 197025. The cycle of Knuth generator is arbitrary; hence the value of gc should also be arbitrary. In the current case, value gc = 197025 is almost three times greater than the number N = 65536 of uniform random variables, from which Knuth algorithm generates Poisson stochastic variables. This entails uncontrolled repetitions and skipping of basic random variables together with a loss of their uniformity, which ultimately leads to an insufficient quality for the results obtained.

So, summing up all the aforementioned points, the aim of this article is to propose a generator of stochastic variables in strict accordance with the theory of Poisson distribution by having no excessive and intermediate generations of uniform variables. This is the next step in searching for better algorithms for Poisson stochastic generation and their optimization.

2. Theory

Poisson stochastic process operates with random variables that linearly depend on a continuous parameter. Usually, such a parameter is the observation time of a stochastic event, but other interpretations are possible as well. In this particular case, an interest is represented both by the time moment *t* and by the time interval τ following it. The randomness of events in two successive continuous time intervals [0, t] and $(t, t + \tau]$ implies [23] that random events in the quantity η could

occur during the total time duration $[0, t + \tau]$. If events in the quantity *k* are observed in the interval [0, t], then a diminution $\eta - k$ for these events should occur in the half-open interval $(t, t + \tau]$. The first axiomatic restriction is due to the fact that both intervals are independent, and also that the events in them separately occur as the substantive cases. The consequence of this restriction is that the probability of observing the events η on the common interval $[0, t + \tau]$ is the joint probability of independent events:

$$P_{\eta}([0,t+\tau]) = P_k([0,t]) \cdot P_{\eta-k}((t,t+\tau])).$$
(2)

In Poisson model the probability $P_{\eta-k}((t, t + \tau])$ has a number of serious limitations, which can be formulated as follows:

1. The probability of events in the time interval $(t, t + \tau]$ does not depend on its origin *t*:

$$P_{\eta-k}((t,t+\tau]) = P_{\eta-k}((0,\tau]).$$
(3)

2. The probability of one event in the time interval $(0, \tau]$ depends linearly on the length of the interval τ with given intensity λ ; and the probability $o(\tau)$ of observing other events is negligible:

$$P_1((0,\tau]) = P_1(\tau) = \lambda \cdot \tau + o(\tau). \tag{4}$$

In Equation (4) the notation $\alpha = \lambda t$ (1) is used, which in the theory of probabilities [2] and the theory of random processes is in common use. Equation (4) excludes an observation of two or more events simultaneously in the time interval τ . This makes it possible to simplify it without considering the events with an infinitesimal small probability of a higher order $o(\tau)$:

$$P_1(\tau) = \lambda \cdot \tau. \tag{5}$$

Equation (5) allows for reaching the determination of probability of the event absence in the time interval τ :

$$P_0(\tau) = 1 - P_1(\tau) = 1 - \lambda \cdot \tau.$$
 (6)

Combining together Equations (2) and (6), the probability of the event absence at the time moment *t* in the general interval $[0, t + \tau]$ could be obtained as follows:

$$P_0(t+\tau) = P_0(t) \cdot P_0(\tau) = P_0(t)(1-\lambda\tau).$$
(7)

This expression (7) leads to the definition of the derivative with respect to probability:

$$\frac{dP_0(t)}{d\tau} = \lim_{\tau \to 0} \frac{P_0(t+\tau) - P_0(t)}{\tau} = -\lambda P_0(t).$$
(8)

The solving of the differential Equation (8) determines the probability of the absence of events at the time moment *t*, in which the constant c = 1 is derived from the initial condition $P_0(0) = 1$. The result is the following:

$$P_0(t) = ce^{-\lambda t} = e^{-\lambda t}.$$
(9)

Equation (9) with allowance for Equation (2) and Constraints (5) and (6) admits calculating the probability of a single event $P_1([0, t + \tau])$ as follows:

$$P_1([0,t+\tau]) = P_0(t)P_1(\tau) + P_1(t)P_0(\tau) = e^{-\lambda t}\lambda\tau + P_1(t)(1-\lambda\tau).$$
(10)

By analogy with Transformations (7) and (8), Equation (10) leads to the following differential equation:

$$\frac{dP_1(t)}{d\tau} = \lambda e^{-\lambda t} - \lambda P_1(t).$$
(11)

The solving of Equation (11) determines $P_1(t)$ with the initial condition $P_1(0) = 0$:

$$P_1(t) = \lambda t e^{-\lambda t}.$$
(12)

Performing successively the Transformations (10)–(12) for all variables $\eta \in [0, \infty]$, the distribution of Poisson probabilities $P_{\eta}(\lambda t)$ with respect to quantity η of random events with intensity λ at the time moment t could be obtained as follows:

$$P_{\eta}(\alpha = \lambda t) = \frac{(\lambda t)^{\eta}}{\eta!} e^{-\lambda t} = \frac{\alpha^{\eta}}{\eta!} e^{-\alpha}.$$
(13)

Since the set of $\eta \in H = [0, \infty]$ of random events in Kolmogorov axiomatics contains σ -algebra, the probability measure $P_{\eta}(\alpha = \lambda t)$ (13) uniquely determines the cumulative function $F_{H}(\eta, \alpha = \lambda t)$ of the distribution probabilities:

$$F_{\rm H}(\eta, \alpha = \lambda t) = \sum_{k=0}^{\eta} \frac{(\lambda t)^k}{k!} e^{-\lambda t} = \sum_{k=0}^{\eta} \frac{\alpha^k}{k!} e^{-\alpha}.$$
(14)

Using a definition of number $e^x = \sum_{k=0}^{\infty} x^k / k!$ for Equation (14), it follows that $F_H(\eta, \lambda) \in [0:1]$. It should be noted here that a probability space guarantees uniqueness of the determination of the inverse probability distribution function. If any value *h* of the cumulative distribution function $F_H(\eta_h, \alpha = \lambda t) = h$ is given, then a value of the stochastic variable η_h can be obtained as the inverse transformation $\eta_h = F_H^{-1}(h)$. Therefore, by specifying the complete uniform random values $F_H^{-1}(\eta, \alpha = \lambda t) \in [0:1]$, it allows uniquely obtaining the stochastic values η of this distribution. This main mathematical model contains the bases for constructing the generators of random variables from given functions of their distribution. Let us use this statement for developing the generator of Poisson stochastic variables. For this it is necessary to get an absolutely complete and uniform generator, which has no repetitions and skipping of random variables. Let us use here the twister generator *nsDeonYuliTwist32D* [24–27], which is of such properties.

In discrete probability space, the number *N* of events is fixed. For each quantitative variable η , with the probability $p(\eta, \alpha = \lambda t)$, it corresponds the frequency $v(\eta, \alpha = \lambda t)$ of observation of the random events:

$$\nu(\eta, \alpha = \lambda t) = p(\eta, \alpha = \lambda t) \cdot N = P_{\eta}(\lambda t) \cdot N.$$
(15)

According to Kolmogorov axiomatics, the determination of probability $p(\eta, \alpha = \lambda t)$ takes precedence over the determination of frequency $v(\eta, \alpha = \lambda t)$. Therefore, in general for the values of frequency $v(\eta, \alpha = \lambda t)$ in (15), it is possible to use mathematical rounding in the fractional part for the multiplication $p(\eta, \alpha = \lambda t) \cdot N$.

Below is the program code, in which the twister generator *nsDeonYuliTwist32D* [26] creates complete set $[\overline{0:2^w}-1]$ of uniform integer random variables *z*, having the length of *w* bits. Function *PoissonD()* creates an array *pEta* of Poisson probabilities (13) and an array of corresponding frequencies *nuEta* (15). The last trailing single frequencies complement an array of frequency distributions *nuEta* to the completeness 2^w of basic uniform random variables. Therefore, the trailing probabilities *pEta* are complemented relying on the trailing single frequencies in *nuEta*. Such a negligible deviation in the rest of the distribution (13) allows for preserving the completeness of generation of Poisson stochastic variables based on initial generation of uniform random variables $z \in [\overline{0:2^w-1}]$. The values of the cumulative frequency function are located in an array of summarized frequencies *snuEta*. Inverse function $F_H^{-1}(\eta_h, \alpha = \lambda t)$ is created by using function *SearchEta()* in accordance with a searching algorithm for the index of element in an array of the cumulative frequencies *cnuEta*. Program names *P060202* and *cP060202* are taken by chance.

```
using nsDeonYuliTwist32D;
                                 //complete twister generator
                              //of integer uniform numbers
namespace P060202
{ class cP060202
{ static void Main(string[] args)
   { int w = 32; //bit width of uniform integer variable
                               //random variable quantity
      long N = 1L \ll w;
      Console.WriteLine("w = {0} N = {1}", w, N);
      double Alpha = 2.0;
      Console.WriteLine("Alpha = {0:F2}", Alpha);
      int wX = 200;
      double[] pEta = new double[wX]; //Poisson probability
      long[] nuEta = new long[wX];
                                       //Poisson frequency
      long[] cnuEta = new long[wX]; //cumulative frequency
      int cEta = PoissonDY(Alpha, pEta, nuEta, cnuEta, N);
      VerifyProbability(N, cEta, pEta, nuEta, cnuEta);
      Console.WriteLine("cEta = {0}", cEta);
      cDeonYuliTwist32D DYG = new cDeonYuliTwist32D();
      DYG.SetW(w);
                      //set bit width of uniform variable
      DYG.Start();
                                 //start uniform generator
      long[] nuDYG = new long[wX];//frequencies of generator
      for (int i = 0; i < wX; i++) nuDYG[i] = 0;
      for (long j = 0; j < N; j++)
      { long z = DYG.Next();
                                        //uniform variable
         int Eta = SearchEta(z, cnuEta, cEta);
        nuDYG[Eta]++;
                               //uniform variable counter
      }
      double spEta = 0.0;
                              //sum of Poisson probability
                                //sum of Poisson frequency
      long snuEta = 0;
      long snuDYG = 0;
                           //sum of variables by generator
      Console.Write("Eta
                             pEta
                                           nuEta");
      Console.WriteLine("
                              cnuEta
                                           nuDYG");
      for (int Eta = 0; Eta <= cEta; Eta++)</pre>
      { Console.WriteLine(
             "{0,2}
                     \{1, 12: F10\} \{2, 10\} \{3, 10\} \{4, 10\}",
             Eta, pEta[Eta], nuEta[Eta],
             cnuEta[Eta], nuDYG[Eta]);
         spEta += pEta[Eta];
         snuEta += nuEta[Eta];
         snuDYG += nuDYG[Eta];
      }
      Console.Write("Sum
                            spEta
                                          snuEta");
      Console.WriteLine("
                                          snuDYG");
      Console.Write("
                         {0,12:F10} {1,10}",
                    spEta, snuEta);
      Console.WriteLine("
                                     {0,10}", snuDYG);
      Console.ReadKey();
                                          //result viewing
   }
//-----
```

```
static int PoissonDY (double alpha, double[] pEta,
                        long[] nuEta, long[] cnuEta, long N)
  { double emAlpha = Math.Exp(-alpha);
     double spEta = 0.0;
                                      //sum of probability
     long snuEta = 0L;
                                        //sum of frequency
     pEta[0] = 1.0 * emAlpha; //Poisson probability p(0)
     spEta += pEta[0];
                                      //sum of probability
     nuEta[0] = (long)Math.Round(pEta[0] * (double)N);
     snuEta += nuEta[0];
                                        //sum of frequency
     cnuEta[0] = snuEta;
                             //cumulative frequency cnu(0)
     double r = alpha;
                                    //Tailor first summand
     pEta[1] = r * emAlpha;
                               //Poisson probability p(1)
     spEta += pEta[1];
                                      //sum of probability
     nuEta[1] = (long)Math.Round(pEta[1] * (double)N);
     snuEta += nuEta[1];
                                        //sum of frequency
     cnuEta[1] = snuEta;
                             //cumulative frequency cnu(1)
     int Eta = 2;
                                         //random variable
     do
     { r *= alpha/(double)Eta; //regular summand of exp
                                     //probability p(Eta)
        double p = r * emAlpha;
        long nu = (long)Math.Round(p * (double)N);
        long sd = snuEta + nu;
        if (nu == 0L || sd > N) break;
                                               //the tail
        pEta[Eta] = p;
                                      //probability p(Eta)
        spEta += p;
                                      //sum of probability
        nuEta[Eta] = nu;
                                       //frequency nu(Eta)
        snuEta += nu;
                                        //sum of frequency
        cnuEta[Eta] = snuEta;
                                   //cumulative frequency
        Eta++;
                       //next random variableследующаяЕta
     } while (snuEta < N);</pre>
     Eta--;
     long d = N - snuEta;
                                     //tailing frequencies
     if (d == 0L) return Eta;
     double d1N = (1.0 - spEta)/(double)d;
     do
     { Eta++;
                             //probability of a tail event
        pEta[Eta] = d1N;
        nuEta[Eta] = 1;
                                     //one-frequency event
        snuEta++;
                                        //sum of frequency
        cnuEta[Eta] = snuEta;
                                    //cumulative frequency
     } while (snuEta < N);</pre>
     return Eta;
  }
//-----
  static void VerifyProbability (long N, int cEta,
        double[] pEta, long[] nuEta, long[] cnuEta)
  { double dN = (double)N;
     for (int i = 0; i \leq cEta; i++)
        pEta[i] = (double)nuEta[i]/dN;
  }
```

After launching the program P060202 the following listing appears on the monitor.

| w = 32 N = 4294967296 | | | | | | | | | |
|-----------------------|--------------|------------|------------|------------|--|--|--|--|--|
| Alpha = 2.00 | | | | | | | | | |
| cEta = 16 | | | | | | | | | |
| Eta | pEta | nuEta | cnuEta | nuDYG | | | | | |
| 0 | 0.1353352831 | 581260615 | 581260615 | 581260615 | | | | | |
| 1 | 0.2706705665 | 1162521231 | 1743781846 | 1162521231 | | | | | |
| 2 | 0.2706705665 | 1162521231 | 2906303077 | 1162521231 | | | | | |
| 3 | 0.1804470443 | 775014154 | 3681317231 | 775014154 | | | | | |
| 4 | 0.0902235222 | 387507077 | 4068824308 | 387507077 | | | | | |
| 5 | 0.0360894089 | 155002831 | 4223827139 | 155002831 | | | | | |
| 6 | 0.0120298029 | 51667610 | 4275494749 | 51667610 | | | | | |
| 7 | 0.0034370865 | 14762174 | 4290256923 | 14762174 | | | | | |
| 8 | 0.0008592717 | 3690544 | 4293947467 | 3690544 | | | | | |
| 9 | 0.0001909493 | 820121 | 4294767588 | 820121 | | | | | |
| 10 | 0.0000381898 | 164024 | 4294931612 | 164024 | | | | | |
| 11 | 0.0000069437 | 29823 | 4294961435 | 29823 | | | | | |
| 12 | 0.0000011572 | 4970 | 4294966405 | 4970 | | | | | |
| 13 | 0.0000001781 | 765 | 4294967170 | 765 | | | | | |
| 14 | 0.000000254 | 109 | 4294967279 | 109 | | | | | |
| 15 | 0.000000035 | 15 | 4294967294 | 15 | | | | | |
| 16 | 0.000000005 | 2 | 4294967296 | 2 | | | | | |
| Sum | spEta | snuEta | | snuDYG | | | | | |
| | 1.000000000 | 4294967296 | | 4294967296 | | | | | |

In this listing, the *pEta* column contains Poisson probabilities distribution (13). In the next column *nuEta*, there is a corresponding frequency distribution. The sum of all frequencies *snuEta* = 4294967296 has to be coincided with generation of the basic uniform random variables having w = 32 bits in the total quantity of $N = 2^w = 2^{32} = 4294967296$. The counters *nuDYG* of the last column confirm the complete coincidence of distribution of the generated random variables with the theoretical frequency distribution *nuEta*.

At this step, the theoretical issues are solved. With the currently reached result, the technology of the cumulative analysis provides an impeccable generation of the random variables with a frequency distribution according to Poisson probabilities. Testing of *P060202* with $w \in [3:32]$ and $\alpha \in [0.1:10]$ confirms an impeccability of received results.

3. Construction and Results

Below is class *nsDeonYuliCPoissonTwist32D*, in which the random variables are created in accordance with Poisson distribution (13). This class is derived over the base one *nsDeonYuliTwist32D*

of the twister generator of uniform random variables [24–27], which in DieHard Tests [28–32] shows an absolute uniform distribution. An example of generation of Poisson stochastic variables is given here later in program *P060302*.

```
using nsDeonYuliTwist32D;
                            //complete twister generator
                        //of integer uniform numbers
namespace nsDeonYuliCPoissonTwist32D
{ class cDeonYuliCPoissonTwist32D : cDeonYuliTwist32D
                       //quantity of uniform events
{ public long N;
                    //quantity of uniform events
  public double dN;
  public double Alpha = 2.0;
                                 //Alpha parameter
  double emAlpha;
                                     //exp(-Alpha)
  public int cEta;
                                     //maximal Eta
  public double[] pC; //probability distribution
  public long[] nuC; //frequency distribution
public long[] cnuC; //cumulative frequencies
//-----
  public cDeonYuliCPoissonTwist32D () {}
//-----
  public void CStart(double alpha)
  { Alpha = alpha;
                                 //Alpha parameter
                  //uniform twister generator
    base.Start();
    CStartInside();
  }
//-----
  public void CTimeStart(double alpha)
  { Alpha = alpha;
                                 //Alpha parameter
    base.TimeStart(); //uniform twister generator
    CStartInside();
  }
//-----
  void CStartInside()
  { int wX = 200;
    pC = new double[wX]; //probability distribution
    nuC = new long[wX];
                          //frequency distribution
    cnuC = new long[wX]; //cumulative frequencies
    emAlpha = Math.Exp(-Alpha);
                                     //exp(-Alpha)
    N = (long)N1 + 1L; //quantity of uniform events
    dN = (double)N; //quantity of uniform events
cEta = CPoissonDY(); //probability and frequency
    CVerifyProbability();
                        //probability verification
  3
//-----
  public int CNext()
  { uint z =base.Next(); //uniform random variable
    return CSearchEta(z); //Poisson random variable
//-----
  int CPoissonDY()
  { double spC = 0.0;
                                 //probability sum
    long snuC = 0L;
                                   //frequency sum
```

```
pC[0] = 1.0 * emAlpha;
                               //Poisson probability p(0)
     spC += pC[0];
                                       //probability sum
     nuC[0] = (long)Math.Round(pC[0] * dN);//frequency nu(0)
     snuC += nuC[0];
                                         //frequency sum
     cnuC[0] = snuC;
                            //cumulative frequency cnu(0)
                                   //Tailor first summand
     double r = Alpha;
     pC[1] = r * emAlpha;
                              //Poisson probability p(1)
     spC += pC[1];
                                       //probability sum
     nuC[1] = (long)Math.Round(pC[1] * dN);//frequency nu(1)
     snuC += nuC[1];
                                         //frequency sum
     cnuC[1] = snuC;
                           //cumulative frequency cnu(1)
     int Eta = 2;
                                       //random variable
     do
     { r *= Alpha/(double)Eta; //regular summand of exp
        double p = r * emAlpha;
                                    //probability p(Eta)
        long nu = (long)Math.Round(p * dN);
        if (nu == 0L) break;
                               //a tail zero frequencies
        long sd = snuC + nu;
        if (nu == 0L || sd > N) break;
                                              //the tail
        pC[Eta] = p;
                                    //probability p(Eta)
        spC += p;
                                       //probability sum
        nuC[Eta] = nu;
                                      //frquency nu(Eta)
        snuC += nu;
                                         //frequency sum
        cnuC[Eta] = snuC;
                                   //cumulative frequency
        Eta++;
                           //the next random variable Eta
     } while (snuC < N);</pre>
     Eta--;
     long d = N - snuC;
                                     //a tail frequencies
     if (d == 0L) return Eta;
     double d1N = (1.0 - spC)/(double)d;
     do
     { Eta++;
                              //a tail event probability
        pC[Eta] = d1N;
        nuC[Eta] = 1;
                                  //one-frequency event
        snuC++;
                                         //frequency sum
        cnuC[Eta] = snuC;
                                   //cumulative frequency
     } while (snuC < N);</pre>
     return Eta;
  }
//-----
  void CVerifyProbability()
  { for (int i = 0; i <= cEta; i++)
        pC[i] = (double)nuC[i]/dN;
  }
//-----
  int CSearchEta(uint z)
  { int Eta = 0;
     for (; Eta <= cEta; Eta++)</pre>
        if (z < cnuC[Eta]) break;</pre>
     return Eta;
```

As an example, let us use the following program code showing the complete generation of Poisson stochastic variables on a base space of uniform values having the bit length w = 7 (for arbitrary $w \le 32$ the listing of $N = 2^{w=32} = 4294967296$ random numbers is too long to present here). This allows demonstrating in a visual form the work of the twister generator *nsDeonYuliCPoissonTwist32D* with observance of Poisson distribution. Program names *P060302* and *cP060302* are assigned arbitrarily.

```
using nsDeonYuliCPoissonTwist32D; //Poisson twister generator
                  //by technology of cumulative frequencies
namespace P060302
{ class cP060302
{ static void Main(string[] args)
   { cDeonYuliCPoissonTwist32D PT =
                    new cDeonYuliCPoissonTwist32D();
      int w = 7;
                    //bit width of uniform random variable
                       //set bit width of uniform variable
      PT.SetW(w);
      double Alpha = 2.0;
                                         //Alpha parameter
      PT.CStart(Alpha);
                                         //start generator
11
        PT.CTimeStart(Alpha);//start generator using time value
      Console.WriteLine("w = \{0\} N = \{1\}", PT.w, PT.N);
      Console.WriteLine("Alpha = {0:F2}", Alpha);
      Console.WriteLine("cEta = {0}", PT.cEta);
      int wX = 200;
      int[] nuG = new int[wX]; //frequencies of generator
      for (int i = 0; i < wX; i++) nuG[i] = 0;
      for (int i = 0, j = 1; i < PT.N; i++, j++)
      { int Eta = PT.CNext();
                                       //Poisson variable
         Console.Write("{0,5}", Eta);
         if (j % 8 == 0) Console.WriteLine();
         nuG[Eta]++;
                        //counter of random variable
      }
      Console.WriteLine();
      double spEta = 0.0;
                                 //Poisson probability sum
      long snuEta = 0;
                                   //Poisson frequency sum
      int snuG = 0;
                             //frequency sum by generator
      Console.Write("Eta
                           pC
                                           nuC");
      Console.WriteLine("
                              cnuC
                                       nuDYG");
      for (int Eta = 0; Eta<= PT.cEta; Eta++)</pre>
      { Console.WriteLine(
              "{0,2} {1,12:F10} {2,8} {3,8} {4,8}",
               Eta, PT.pC[Eta], PT.nuC[Eta],
               PT.cnuC[Eta], nuG[Eta]);
         spEta += PT.pC[Eta];
         snuEta += PT.nuC[Eta];
         snuG += nuG[Eta];
      }
      Console.Write("Sum
                                          snuC");
                            spC
```

After executing the program *P060302* the following result appears on monitor.

| w = 7 N = 128 | | | | | | | | |
|----------------|----------------|-------|-----|------|------|-------|--------|--|
| Alpha = 2.00 | | | | | | | | |
| cEta = 6 | | | | | | | | |
| 1 | 6 | 3 | 1 | 3 | 1 | 1 | 2 | |
| 2 | 1 | 0 | 2 | 0 | 2 | 3 | 3 | |
| 3 | 2 | 1 | 4 | 2 | 3 | 0 | 0 | |
| 0 | 4 | 2 | 1 | 3 | 0 | 1 | 2 | |
| 1 | 1 | 5 | 2 | 0 | 1 | 2 | 3 | |
| 3 | 2 | 1 | 4 | 1 | 3 | 4 | 0 | |
| 0 | 4 | 2 | 1 | 2 | 0 | 1 | 1 | |
| 1 | 1 | 4 | 2 | 5 | 1 | 2 | 2 | |
| 2 | 2 | 1 | 3 | 1 | 2 | 4 | 5 | |
| 5 | 3 | 2 | 1 | 2 | 5 | 1 | 1 | |
| 1 | 0 | 3 | 2 | 4 | 1 | 2 | 2 | |
| 2 | 2 | 1 | 3 | 1 | 2 | 3 | 4 | |
| 4 | 3 | 2 | 0 | 2 | 4 | 0 | 1 | |
| 1 | 0 | 3 | 1 | 4 | 1 | 2 | 2 | |
| 2 | 1 | 0 | 3 | 1 | 2 | 3 | 3 | |
| 3 | 3 | 2 | 0 | 2 | 3 | 0 | 1 | |
| Eta pC | | | nuC | | cnuC | nuDYG | | |
| 0 0.1328125000 | | | 0 | 17 | | 17 | 17 | |
| 1 | 0.2734375000 | | | 35 | | 52 | 35 | |
| 2 | 0.2734375000 | | | 35 | | 87 | 35 | |
| 3 | 3 0.1796875000 | | | 23 | | 110 | 23 | |
| 4 | 4 0.0937500000 | | | 12 | | 122 | 12 | |
| 5 0.0390625000 | | | 0 | 5 | | 127 | 5 | |
| 6 0.0078125000 | | | 0 | 1 | | 128 | 1 | |
| sum spC | | | | snuC | | | snuDYG | |
| | 1.0000 | 00000 | 0 | 128 | | | 128 | |

This result shows a stochastic sequence of random variables generated by twister generator nsDeonYuliCPoissonTwist32D. The direct calculation by this listing confirms that all the elements including the random variables, their probabilities and frequencies do indeed satisfy Poisson distribution. Similar results could be obtained for other quantities as well, for example with a length of $w \leq 32$ bits.

4. Discussion

As mentioned at the beginning of this article, Poisson distribution has basic properties of initial probabilistic moments in terms of mathematical expectation and dispersion. The mathematical expectation for this type of a distribution characterizes the average number of successful results at any

interval. Usually, it is determined on a basis of the experimentally obtained data for a certain situation. Next, if the mathematical expectation is determined, then the dispersion is known also because of the distribution properties of Poisson probabilities. If it turns out that the values of the mathematical expectation and the dispersion are sufficiently close, then the hypothesis of the distribution of certain random variables in accordance with Poisson law is correct. However, if there is a meaningful difference in the obtained values of these characteristics, this would testify against the hypothesis of Poisson distribution of the given random variables.

Therefore, first of all, it is necessary to confirm two basic aforementioned properties of Poisson distribution on equality of parameter α to mathematical expectation and dispersion. Below is the program code that validates this. The generator in it creates the random variables in quantity $N = 2^w = 2^{32} = 4294967296$. Program names *P060402* and *cP060402* are chosen by chance.

```
using nsDeonYuliCPoissonTwist32D; //Poisson twister generator
                  //by technology of cumulative frequencies
namespace P060402
  class cP060402
{
{ static void Main(string[] args)
   { cDeonYuliCPoissonTwist32D PT =
                       new cDeonYuliCPoissonTwist32D();
      PT.SetW(32);
                             //bit width of random variable
      double Alpha = 2.0;
                                          //Alpha parameter
      PT.CStart(Alpha);
                                         //start generator
      Console.WriteLine("w = {0} N = {1}", PT.w, PT.N);
      Console.WriteLine("Alpha = {0:F2}", Alpha);
      double p1 = 1.0/(double)PT.N;
                                      //event probability
      double m = 0.0;
                                 //mathematical expectation
      double D = 0.0;
                                               //dispersion
      for (long i = 0; i < PT.N; i++)
        int Eta = PT.CNext();
                                          //random variable
      {
         m += Eta * p1;
                                //mathematical expectation
         D += Eta * Eta * p1;
                                               //dispersion
      }
      D = D - m * m;
      Console.WriteLine("m = {0:F10}", m);
      Console.WriteLine("D = {0:F10}", D);
      Console.ReadKey();
                                           //result viewing
   }
}
}
```

After running the program P060402, the following listing appears on the monitor.

```
w = 32 N = 4294967296
Alpha = 2.00
m = 2.0000000014
D = 2.0000000116
```

This listing shows the obtained values of mathematical expectation *m* and dispersion *D*. Their negligible difference from the value of parameter $\alpha = 2.0$ is related to the discreteness of Poisson model in probabilities and frequencies of the events.

Further, a general discussion should also supplement the estimates of possible generation of the random variables with respect to Poisson distribution. To do this, let us refer to the capabilities of

the basic twister generator *nsDeonYuliTwist32D*, which is used here for uniform random variables creation. For a given length of w bits it realizes several complete twisting sequences. Initial sequence contains $N = 2^w$ non-repeating numbers distributed uniformly and randomly in interval $[0:2^w - 1]$. For utilization, the constants a and c of the twister generation of the random variable $x_i = (ax_{i-1} + c)mod 2^w$ obtained from the value of the previous variable x_{i-1} are used. When the generator completes the creation of one series consisting of $N = 2^w$ random variables, it automatically proceeds to the creation of the next twisting series, in which $N = 2^w$ elements are obtained as well. So, for each pair of twister constants a and c, the twisting uniform sequences in quantity N_T could be created:

$$N_T = w \cdot 2^w. \tag{16}$$

Consequently, one complete twisting cycle (16) with unchanged *a* and *c* realizes the following quantity N_s for uniform random variables that allows the creation of the same number of Poisson stochastic variables:

$$N_s = N \cdot N_T = 2^w \cdot w \cdot 2^w = w \cdot 2^{2w}.$$
(17)

To obtain a complete cycle of all Poisson stochastic series with quantity $N_s = w \cdot 2^{2w}$ of the random variables in each complete twister (17), it is necessary to take into account the varieties of coefficients *a* and *c* in the twister transformation $x_i = (ax_{i-1} + c)mod \ 2^w$. The values of these coefficients must not exceed the interval limit $a, c \in [0: 2^w - 1]$ of all uniform variables in the single series. Quantity N_a of the coefficient *a* is defined as:

$$N_a = \frac{N}{4} = \frac{2^w}{2^2} = 2^{w-2}.$$
(18)

During the generation of uniform random variables, the values of the coefficients *c* have to be odd, i.e., *c* mod $2 \neq 0$. Their quantity N_c is defined as the following:

$$N_c = \frac{N}{2} = \frac{2^w}{2^1} = 2^{w-1}.$$
(19)

Collecting Equations (16)–(19) together, the estimate for the total number N_{sac} of the random variables is defined in the following manner:

$$N_{sac} = N_s \cdot N_a \cdot N_c = w 2^{2w} 2^{w-2} w^{w-1} = w \cdot 2^{4w-3}.$$
(20)

Finally, the information concerning non-repeatable cycle N_I has the following estimation:

$$N_I = 2^{wN_{sac}} = 2^{w^2 2^{4w-3}}.$$
(21)

As an example, let us get the real values of these estimations for the bit length w = 32. In this case, each series contains $N = 2^{32} = 4294967296$ Poisson stochastic variables, and in each of these series Poisson distribution is observed absolutely. In this case, the total number of generated Poisson stochastic variables is $N_{sac}(w = 32) = 32 \cdot 2^{4 \cdot 32 - 3} = 2^5 \cdot 2^{125} = 2^{130}$, and the value of the non-repeatable cycle (21) is defined as $N_I(w = 32) = 2^{2^5 2^{130}} = 2^{2^{135}}$.

Therefore, the complete discrete simulation of the random variables with Poisson distribution using the aforementioned basic twister generator confirms the important properties in terms of mathematical expectation and dispersion. Moreover, an automatic extension of the series of initial uniform random variables extends significantly the periods of non-repeatability (Equations (20) and (21)) for the series of Poisson stochastic variables. This notably exceeds the features of the known generators, which bases the probabilistic convergence algorithm.

5. Conclusions

An analysis of the source material shows that algorithms for generation of Poisson stochastic variables according to the probabilistic convergence technology lead to different realizations of

distributions. Moreover, the skipping of elements in such distributions may also occur, which is inappropriate in the theory of Poisson stochastic processes. Since the quality of the generators depends on the basic generations used for uniform random variables, it has been proposed here to use the twister generator *nsDeonYuliCPoissonTwist32D* to ensure the completeness of technology of the cumulative array of frequencies in Poisson space. This approach allows for reducing the time of operations and improving the quality of generation. In the discrete probabilistic Poisson space, the test results confirmed the complete coincidence of the distributions of the obtained stochastic variables with the ones received in theoretical modeling. Moreover, an automatic tuning of the parameters of the basic twister uniform generator *nsDeonYuliTwist32D* allows obtaining a significant increase in the overall period of non-repeatable generation of Poisson stochastic variables.

Author Contributions: Paper writing, conceptualization and methodology, programming and analysis, investigation and data curation, algorithm improvement were done by A.F.D. Paper writing, conceptualization, validation, original draft preparation, reviewing and editing, project administration and supervision were done by Y.A.M. All authors have read and approved the final manuscript.

Funding: This research received no external funding.

Acknowledgments: The authors are thankful to Matthew Vandenberg, J. Alex Watts, Jacqueline Nolan and Walter Harrington (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. Feller, W. *An Introduction to Probability Theory and Its Applications*, 3rd ed.; John Wiley & Sons: Hoboken, NJ, USA, 2008.
- 2. Gnedenko, B. *Theory of Probability*, 6th ed.; CRC Press: Boca Raton, FL, USA, 1998; p. 520.
- Zhang, H.; Li, B. Characterizations of discrete compound poisson distribution. *Commun. Stat.-Theory Method.* 2016, 45, 6789–6802. [CrossRef]
- 4. Guerriero, V. Power low distribution: method of multi-scale inferential statistics. *J. Mod. Math. Front.* **2012**, *1*, 21–28.
- 5. Arkani, M.; Khalafi, H.; Vosoughi, N. A flexible multichannel digital random pulse generator based on FPGA. *J. Nucl. Sci. Tech.* **2013**, *3*, 109–116. [CrossRef]
- 6. Rasoanaivo, A.N.; Horowitz, W.A. Medium-induced radiation beyond the Poisson approximation. *J. Phys. Conf.* **2017**, *878*. [CrossRef]
- Veiga, A.; Spinelli, E. A pulse generator with Poisson-exponential distribution for emulation of radioactive decay events. In Proceedings of the IEEE 7th Latin American Symposium on Circuits & Systems (LASCAS), Florianopolis, Brazil, 28 February–2 March 2016; pp. 31–34. [CrossRef]
- 8. Kirkpatrick, J.M.; Young, B.M. Poisson statistical methods for the analysis of low-count gamma spectra. *IEEE Trans. Nucl. Sci.* **2009**, *56*, 1278–1282. [CrossRef]
- 9. Marsaglia, G.; Tsang, W.W.; Wang, J. Fast generation of discrete random variables. *J. Stat. Software* **2004**, *11*, 1–11. [CrossRef]
- 10. Kumari, S.; Valarmathi, M.; Prince, S. Generation of pseudorandom binary sequence using shot noise for optical encryption. In Proceedings of the International Conference on Communication and Signal Processing (ICCSP), Melmaruvathur, India, 6–8 April 2016; pp. 0119–0122. [CrossRef]
- 11. Hosamo, M. A Study of the Source Traffic Generator Using Poisson Distribution for ABR Service. *Model. Simul. Eng.* **2012**, 2012, 1–6. [CrossRef]
- 12. Zhang, H.; Liu, Y.; Li, B. Notes on discrete compound poisson model with applications to risk theory. *Insur. Math. Econ.* **2014**, *59*, 325–336. [CrossRef]
- 13. Shanmugam, R. Informatics about fear to report rapes using bumped-up poisson model. *Am. J. Biostat.* **2013**, 3, 17–29. [CrossRef]
- 14. 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. Optic. Express* **2013**, *4*, 3030–3041. [CrossRef] [PubMed]

- 15. 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. Optic. Express* **2016**, *7*, 3643–3658. [CrossRef] [PubMed]
- Juratli, M.A.; Menyaev, Y.A.; Sarimollaoglu, M.; Melerzanov, A.V.; Nedosekin, D.A.; Culp, W.C.; Suen, J.Y.; Galanzha, E.I.; Zharov, V.P. Noninvasive label-free detection of circulating white and red blood clots in deep vessels with a focused photoacoustic prob. *Biomed. Opt. Express* 2018, *9*, 5667–5677. [CrossRef] [PubMed]
- 17. Sitek, A.; Celler, A.M. Limitations of Poisson statistics in describing radioactive decay. *Phys. Med.* **2015**, *31*, 1105–1107. [CrossRef] [PubMed]
- Menyaev, Y.A.; Zharov, V.P. Experience in development of therapeutic photomatrix equipment. *Biomed. Eng.* 2006, 40, 57–63. [CrossRef]
- 19. Menyaev, Y.A.; Zharov, V.P. Experience in the use of therapeutic photomatrix equipment. *Biomed. Eng.* **2006**, 40, 144–147. [CrossRef]
- 20. Knuth, D.E. Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd ed.; Addison-Wesle: Boston, MA, USA, 2014; p. 784.
- 21. Knuth, D.E. Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1, 1st ed.; Addison-Wesley: Boston, MA, USA, 2011; p. 912.
- 22. Wikipedia. Poisson Distribution. Available online: https://en.wikipedia.org/wiki/Poisson_distribution (accessed on 26 May 2019).
- 23. Kolmogorov, A.N.; Fomin, S.V. *Elements of the Theory of Functions and Functional Analysis*; Dover Publication: Mineola, NY, USA, 1974; p. 128.
- 24. Deon, A.F.; Menyaev, Y.A. The Complete Set Simulation of Stochastic Sequences without Repeated and Skipped Elements. *J. Univers. Comput. Sci.* **2016**, *22*, 1023–1047. [CrossRef]
- 25. Deon, A.F.; Menyaev, Y.A. Parametrical tuning of twisting generators. J. Comput. Sci. 2016, 12, 363–378. [CrossRef]
- 26. Deon, A.F.; Menyaev, Y.A. Twister generator of arbitrary uniform sequences. J. Univers. Comput. Sci. 2017, 23, 353–384. [CrossRef]
- 27. Deon, A.F.; Menyaev, Y.A. Uniform twister plane generator. J. Comput. Sci. 2018, 14, 260–272. [CrossRef]
- 28. Wikipedia. Diehard Tests. Available online: https://en.wikipedia.org/wiki/Diehard_tests (accessed on 26 May 2019).
- 29. The Marsaglia Random Number CDROM Including the Diehard Battery of Tests of Randomness. Available online: https://stat.fsu.edu/pub/diehard/ (accessed on 26 May 2019).
- Runs Test for Detecting Non-randomness. Available online: https://www.itl.nist.gov/div898/handbook/eda/ section3/eda35d.htm (accessed on 26 May 2019).
- 31. Sample 33092: Wald-Wolfowitz (or Runs) Test for Randomness. Available online: https://support.sas.com/ kb/33/092.html (accessed on 26 May 2019).
- 32. Alhakim, A.; Hooper, W. A non-parametric test for several independent samples. *J. Nonparametric Stat.* 2008, 20, 253–261. [CrossRef]



© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).