УДК 681.3.06(075) DOI: 10.18698/0236-3933-2017-2-86-110

ГЕНЕРАТОР РАВНОМЕРНЫХ СЛУЧАЙНЫХ ВЕЛИЧИН ПО ТЕХНОЛОГИИ ПОЛНОГО ВИХРЕВОГО МАССИВА

А.Ф. Деон[1] deonalex@mail.ru
Ю.А. Меняев[2] yamenyaev@uams.edu

[1] **МГТУ им. Н.Э. Баумана, Москва, Российская Федерация**
[2] **Институт исследования рака им. Уинтропа Рокфеллера, Арканзас, США**

Аннотация

Генераторы равномерно распределенных случайных величин активно используются в различных приложениях, начиная от математики, моделирования радиоэлектронных и технических конструкций вплоть до медицинских и биологических исследований. Предложен новый подход к генерации случайных величин на основе объединения возможностей начального конгруэнтного массива с глобальным вихрем кольцевой технологии для полных стохастических последовательностей. Экспериментально подтверждено, что для полных последовательностей такой тип генерации обеспечивает равномерное распределение случайных величин. Предлагаемое программное обеспечение допускает методы настройки технологии генерации, где случайные величины могут принимать любую битовую длину. Рассмотрено автоматическое переключение параметров генератора, таких как начальные значения и конгруэнтные константы, что позволяет увеличивать число вариантов генерируемых последовательностей. Приведенные результаты тестирования подтверждают абсолютную равномерность распределения без каких-либо повторений или пропусков в генерируемых последовательностях случайных величин

Ключевые слова

Генератор псевдослучайных величин, случайные последовательности, конгруэнтная величина, вихревой генератор

Поступила в редакцию 13.12.2016

Введение. Псевдогенераторы случайных величин (PRNG) представляют собой хорошо известную технологию с широким приложением в различных областях, начиная от криптографии [1–3], моделирования случайных процессов [4, 5], полного тестирования технических систем [6, 7], до медицинских [8] и биологических [9, 10] исследований. Значительные успехи достигнуты в таких направлениях, как линейные конгруэнтные генераторы [11, 12] и последующие вихревые алгоритмические генераторы, в которых обычно используются числа Мерсена [13, 14]. Важные результаты были получены в применении чисел Фибоначчи [15, 16], алгоритма BBS [17] и др. Однако вопросы повторяемости элементов и полноты наборов элементов таких генераторов по-прежнему актуальны.

Согласно технологии конгруэнтной генерации, каждая следующая случайная величина x_{i+1} создается на основе текущей случайной величины x_i с помощью ограниченного функционального преобразования

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

Модуль $mod\ m$ определяет интервал $\left[\overline{0, m-1}\right]$ генерируемых величин. Исторически функция $f(x_i)$ выбиралась как линейное алгебраическое преобразование

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

Константные коэффициенты *a* и *c* подбираются согласно свойствам проектируемого генератора. Поясним это на примере технологии генерации нескольких $N = 8$ случайных величин из небольшого интервала $\left[\overline{0, 7}\right]$. Выбор малой длины интервала не нарушает общности исследования, но позволяет представлять результаты более очевидно в простой наглядной форме. В качестве начального значения возьмем $x_0 \in \left[\overline{1, 7}\right] \subset \left[\overline{0, 7}\right]$. Также примем во внимание все случайные последовательности при различных $a \in A = \left[\overline{1, 7}\right]$ и $c \in C = \left[\overline{0, 7}\right]$. Таким образом, общее число возможных комбинаций *a, c, x* обозначим как

$$NN = card(A) \cdot card(C) \cdot (N-1) = 7 \cdot 8 \cdot 7 = 392.$$

Теперь добавим в компьютерную программу функцию *Repeating()*, которая возвращает истинное значение *true*, если любой элемент *x* из 0, 1, …, 7 имеет повторение в случайной последовательности из восьми чисел. Этот подход помогает определить конгруэнтную равномерную полноту, реально перебрав все возможные значения:

$$CC = r / NN,$$

где *r* — число равномерных случайных последовательностей, а *NN* — общее число последовательностей.

Далее приведен программный код на диалекте C# [18] среды Visual Studio 2015 Microsoft, хотя подобный код можно было бы изложить на историческом C (диалект Win32) или C++ (диалект CLR). В любом варианте результат одинаковый.

```
static void Main( string[] args )
{  int N = 8;                              // длина последовательности
   Console.WriteLine("N = {0}", N);
   int NN = 0;                     // общее число последовательностей
   int m = N;                                  // модуль конгруэнтности
   int r = 0;                // номер равномерной последовательности
   int[] x = new int[N];            // случайная последовательность
   for ( int a = 1; a < N; a++ )
      for ( int c = 0; c < N; c++ )
         for ( int x0 = 1; x0 < N; x0++ )
         {  x[0] = x0;                  // начало последовательности
```

```
            for ( int i = 1; i < N; i++ )
               x[i] = ( a * x[i - 1] + c ) % m;
            NN++;              // общее число последовательностей
            if ( Repeating(x, N) ) continue;            // повтор
            Console.WriteLine( "a = {0}  c = {1}", a, c );
            r++;                       // номер последовательности
            Console.Write( "r = {0,4}  ", r );
            qWrite("x = ", x, N, true);
         }
         Console.WriteLine( "Finish" );
         Console.WriteLine( "NN = {0}", NN );
         double CC = (double)r / NN;      // конгруэнтная полнота
         Console.WriteLine("CC = r / NN = {0:F4}", CC);
         Console.ReadKey();                // просмотр результата
}
//---------------------------------------------------------------
static bool Repeating ( int[] x, int N )
{  for ( int i = 1; i < N; i++ )
      for ( int j = 0; j < i; j++ )
         if ( x[i] == x[j] ) return true;           // повторение
   return false;                                 // повторений нет
}
//---------------------------------------------------------------
static void qWrite ( string text, int[] x, int N,
                     bool newstr )
{  Console.Write( text );
   for ( int i = 0; i < N; i++ )
      Console.Write( "{0,3}", x[i] );
   if ( newstr ) Console.WriteLine();
}
```

После выполнения этой программы на мониторе появляется следующий листинг, который приводим с некоторыми сокращениями, ставя прочерк на месте пропущенных строк

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

Листинг показывает, что наблюдалось только 56 равномерных последовательностей из возможных 392 конгруэнтных последовательностей во всех диапазонах

допустимых значений параметров *a*, *c*, *x*. Другими словами, мы имеем не полную возможность равномерных последовательностей, а лишь *CC* = 56/392 = = 0,1429 = 14,29 % полного конгруэнтного списка. Показатель невелик.

При генерации конгруэнтных случайных последовательностей принимается следующая технология: операцию модуля *mod* в (1) можно заменить на операцию & битовой конъюнкции. Такой вариант действий допускается, если генерируемое двоичное число *x* длиной *w* бит принадлежит интервалу $x \in \left[\overline{0, 2^w - 1}\right]$.

Технология вихревых генераторов использует в качестве основы битовый сдвиг двоичных чисел в случайной последовательности. Частный случай такого подхода использовался в классических работах японских исследователей Матсумото и Нишимура [13, 14, 19] на основе ранних исследований Леви [20]. Они разработали несколько генераторов, включая хорошо известный генератор MT19937 (или MT19937-64 для выполнения с 64-битными словами), которые, по оценкам авторов, имеют большую величину повторяемости $2^{19937} - 1$, что довольно значительно в некоторых специальных случаях.

В общем случае суть кольцевого сдвига или глобального вихря заключается в следующем. Если взять два числа x_i и x_{i+1} одинаковой битовой длины *w*, то новые значения создаются согласно правилу: значения бит, взятые из числа x_{i+1}, успешно сдвигаются влево в число x_i; в то же время высвобождающиеся биты слева из числа x_i циклично присоединяются один за другим к правой части числа x_{i+1}.

В качестве примера покажем вихревой сдвиг в двоичной форме при $w = 3$ для последовательности чисел $x_0 = 5_{10} = 101_2$, $x_1 = 3_{10} = 011_2$, $x_2 = 6_{10} = 110_2$. Структуру такого подхода можно представить так же, как на диаграмме (рис. 1),

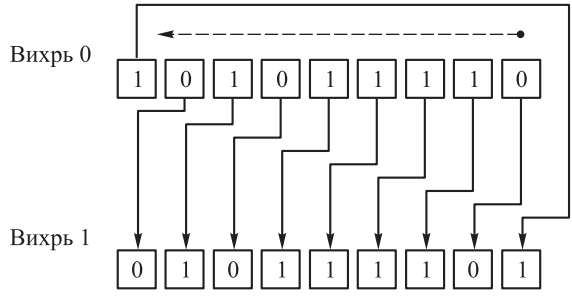

Рис. 1. Диаграмма кольцевого вихря

где рассмотрены первые две строки. Вихрь 0 является начальной последовательностью конгруэнтной генерации. Вихрь 1 является результатом глобального сдвига влево на 1 бит. В свою очередь старший бит исходной последовательности совершает кольцевое движение в последнюю позицию справа в следующей последовательности. Таким образом, исходная последовательность 101 011 110_2 = = 5 3 6_{10} превращается в вихревую 010 111 101_2 = 2 7 5_{10}. Этот алгоритм называется вихревой технологией, или кольцевым вихрем, или глобальным кольцевым вих-

рем, поскольку используется одновременный битовый сдвиг всех чисел без потери любого бита.

Следующий листинг показывает 10 последовательностей левого вихря. Для удобства восприятия в конце каждой строки выводятся их десятичные эквиваленты. Восемь строк, со второй по девятую, являются результатом сдвига влево на 1 бит с кольцевым переносом старшего левого бита в младший правый бит последовательности. Последняя десятая последовательность повторяет первую строку, подчеркивая кольцо вихря.

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

В этом листинге можно видеть, что число 5 повторяется 9 раз, число 3 — 3 раза, число 6 — 3 раза, возникающее число 7 — 6 раз. Этот результат далек от идеального случая, когда все генерируемые числа распределены равномерно. Все это не соответствует предположению об общей равномерной генерации чисел.

В теории методов вычислений случайных величин существуют также другие принципы генерации равномерно распределенных чисел [21–23].

Рассмотрим кратко два основных направления. В первом случае в большинстве технологий применяют сложные алгебраические формулы или многошаговые математические преобразования, что занимает много времени процесса и, таким образом, подлежит обсуждению, прежде чем включать их в практику реального времени даже для современных быстрых компьютеров.

Во втором случае простейшие искусственные математические решения, подобные среднеквадратическому методу Нейманна [24], имеют некоторые ограничения коротких циклов, которые могут привести к нулю. Хотя простота и высокое быстродействие достигаются, их выход имеет низкое качество [7]. Главное, оба направления не имеют 100 %-ной полноты множеств неповторяемых элементов и, более того, они хуже, чем вихревые случайные генераторы для таких же задач.

Итак, задачей настоящего исследования является поиск дополнительных возможностей вихревой технологии для получения полных равномерных стохастических последовательностей, в которых нет отсутствующих или повторяющихся случайных величин.

Теория. В технологии глобального вихря положительно то, что возможно появление новых значений случайных величин, которых не было в исходной последовательности. В предыдущем разделе результат последнего примера показывает два новых числа 010 и 111 после кольцевого сдвига трех начальных

чисел 101, 011 и 110. В то же время отрицательным результатом является то, что во всех полученных последовательностях числа повторяются неравномерно. Ситуацию можно исправить, если учитывать следующее ограничение.

Чтобы добиться вихря без повторений, необходимо воспользоваться конгруэнтным генератором $x_{i+1} = (ax_i + c)\ \&\left(2^w - 1\right)$ на полной последовательности чисел, каждое длиной w бит. Полные последовательности содержат $N = 2^w$ чисел, т. е. каждая случайная величина $x \in \left[\overline{0, N-1}\right] = \left[\overline{0, 2^w - 1}\right]$. Поскольку в вихре сдвиг выполняется w раз, то объединение начальной конгруэнтной последовательности с остальными вихревыми последовательностями дает $w \cdot N = w \cdot 2^w$ последовательностей. Возникает вопрос, сколько уникальных (неповторяющихся) последовательностей можно создать, объединяя технологии конгруэнтной и вихревой генерации?

Далее приведен программный код, позволяющий с помощью функции *MatrixAdd()* расположить генерируемые конгруэнтные и вихревые последовательности в вспомогательной матрице *MS*. В каждой строке находится одна последовательность и три вспомогательных элемента. Затем функция *MatrixCheck()* сравнивает последовательность в каждой строке матрицы со всеми остальными строками. Если последовательность в текущей строке совпадает с любой предыдущей строкой, то номер предыдущей строки заносится в первый вспомогательный элемент. Второй и третий вспомогательные элементы содержат константы конгруэнтности *a* и *c*. Когда все строки будут проверены, тогда распечатка матрицы позволит обнаружить повторяющиеся последовательности, поскольку их первый вспомогательный элемент будет отличен от нуля.

```
static void Main ( string[] args )
{  int w = 3;                      // битовая длина числа
   int N = 8;                 // длина последовательности
   int maskW = 0x7;                        // маска числа
   int maskU = 0x4;               // маска старшего бита
   Console.WriteLine ( "w = {0}  N = {1}", w, N);
   int[] x = new int[N]; // случайная последовательность
   int[,] MS = new int[2000, N + 3];          // матрица
   int M = 0;// количество последовательностей в матрице
   int k = 0;         // номер полной последовательности
   int a = 5, c = 1;           // конгруэнтные константы
   for ( int x0 = 1; x0 < N; x0++ )
   {  Cong_Start( x, N, a, c, x0, maskW );
      if ( Repeating ( x, N ) ) continue;  // повторение
      MatrixAdd ( MS, ref M, x, N, a, c );
      Console.WriteLine ( "a = {0}  c = {1}", a, c );
      k++;                    // номер последовательности
       Console.Write ( "k = {0,4}  ", k );
      qWrite ( "x = ", x, N, true );
      for ( int i = 1; i < 24; i++ )
      {  Twist ( x, w, N, maskW, maskU );       // вихрь
         if (Repeating(x, N)) continue;
         MatrixAdd ( MS, ref M, x, N, a, c );
```

```
          k++;
          Console.Write ( "k = {0,4}  ", k );
          qWrite ( "x = ", x, N, true );
       }
   }
   MatrixCheck ( MS, M, N, 1 );   // проверка совпадений
   Console.WriteLine ( "Matrix of unique sequences" );
   MatrixWrite ( MS, M, N );          // монитор матрицы
   Console.ReadKey();              // просмотр результат
}
//-----------------------------------------------------
static void MatrixAdd ( int[,] MS, ref int M,
                        int[] x, int N, int a, int c )
{  for ( int j = 0; j < N; j++ ) MS[M, j] = x[j];
   MS[M, N] = 0;                       // номер совпадения
   MS[M, N + 1] = a;                        // параметр a
   MS[M, N + 2] = c;                        // параметр c
   M++;      // количество последовательностей в матрице
}
//-----------------------------------------------------
static void MatrixCheck ( int[,] MS, int M,
                          int N, int bMS )
{  for ( int i = bMS; i < M; i++ )
      for ( uint k = 0; k < i; k++ )
      {  uint j = 0;
         for ( ; j < N; j++ )
            if ( MS[i, j] != MS[k, j] ) break;
         if ( j == N )                // совпадение есть
         {  MS[i, N] = k + 1;        // номер совпадения
            break;
         }
      }
 }
//-----------------------------------------------------
static void MatrixWrite ( int[,] MS, int M, int N )
{  for ( int i = 0; i < M; i++ )
   {  Console.Write ( "i = {0,5} |", i + 1 );
      for ( int j = 0; j < N; j++ )
         Console.Write ( "{0,3}", MS[i, j] );
      Console.Write ( " | " );
      Console.Write ( "{0,3}", MS[i, N] );
      Console.Write ( "{0,5}", MS[i, N + 1] );
      Console.WriteLine ( "{0,5}", MS[i, N + 2] );
   }
}
```

После выполнения этой программы на мониторе появляется следующий листинг, который приведен с некоторыми сокращениями — прочерк на месте пропущенных строк. Общее число последовательностей, сгенерированных по конгруэнтной и вихревой технологиям, составляет $7 \cdot 24 = 168$. Каждая содержит 8 различных чисел. Эти последовательности были занесены в матрицу *MS* последовательно. Вторая часть листинга представляет собой результат проверки. Если строка содержит нуль в первом дополнительном элементе, то это озна-

чает, что последовательность является уникальной. Их общее число составляет $w \cdot N = 3 \cdot 8 = 24$ и они находятся в первых 24-х строках второй части листинга. Последовательности в строках с 25 по 168 не являются уникальными, поскольку они повторяют предыдущие 24 строки в различных комбинациях.

```
w = 3  N = 8
a = 5  c = 1
k =    1   x =   1  6  7  4  5  2  3  0
k =    2   x =   3  5  7  1  2  4  6  0
- - - - -
k =    6   x =   3  6  2  5  1  4  0  7
- - - - -
k =   16   x =   2  3  0  1  6  7  4  5
- - - - -
k =  167   x =   5  7  1  2  4  6  0  3
k =  168   x =   3  6  2  5  1  4  0  7
Matrix of unique sequences
k =    1 |  1  6  7  4  5  2  3  0 |  0  5  1
k =    2 |  3  5  7  1  2  4  6  0 |  0  5  1
n- - - - -
k =   16 |  2  3  0  1  6  7  4  5 |  0  5  1
- - - - -
k =   24 |  0  7  3  6  2  5  1  4 |  0  5  1
k =   25 |  2  3  0  1  6  7  4  5 | 16  5  1
- - - - -
k =  168 |  3  6  2  5  1  4  0  7 |  6  5  1
```

Последовательность в строке 25 повторяет вихревую последовательность в строке 16, а последняя вихревая последовательность в строке 168 совпадает с последовательностью в строке 6.

Итак, последний результат показывает, что конгруэнтная технология генерации случайных последовательностей с полным набором исходных чисел является частью вихревой технологии. Это также подтверждается моделированием с произвольной битовой длиной *w*.

Параметры. Выявленные в предыдущих разделах особенности конгруэнтной и вихревой генераций позволяют перейти к вопросам проектирования конкретного генератора случайных равномерных последовательностей. Одним из основных параметров настройки генератора является битовая длина *w* получаемых чисел. В реальных технических вопросах нужны генераторы с длиной чисел 15, 16, 24, 32, 64 бит.

Другой параметр — это количество чисел в каждой последовательности. Для равномерной генерации выгодно придерживаться зависимой длины $N = 2^w$, поскольку это позволяет использовать вихревую технологию для каждой последовательности [25].

Теперь о формуле создания чисел в последовательности. Обычно последовательность получают конгруэнтным способом по линейной формуле $x_{i+1} = (ax_i + c) mod\ m$. Для полных последовательностей операцию *mod m* можно заменить конъюнкцией & с битовой длиной *w*. Таким образом, допустимо выражение $x_{i+1} = (ax_i + c)\ \&\left(2^w - 1\right)$.

Исходное число x_0 начала генерации можно задать либо явно, либо автоматически, используя таймер компьютера. Для полных последовательностей интерес представляют все числа только из интервала $\left[\overline{0, N-1}\right] = \left[\overline{0, 2^w - 1}\right]$.

В конгруэнтной линейной формуле (2) параметр *a* выбирается по условию, чтобы получающаяся последовательность была полной при заданной длине чисел *w* бит. Известно, что равномерные последовательности с $N = 2^w$ чисел создаются тогда, когда $(a-1) mod 4 = 0$. Например, если длина чисел $w = 3$, то последовательность полная и содержит $N = 2^w = 2^3 = 8$ чисел. В этом интервале $\left[\overline{0, 2^w - 1}\right] = \left[\overline{0, 7}\right]$ параметр *a* следует выбирать как *a* = 1 или 5. Если $w = 4$, то *a* может принимать значения только 1, 5, 9 или 13 из интервала $a \in \left[\overline{1, 2^4 - 1}\right] = \left[\overline{1, 15}\right]$.

Параметр *c* в формуле (2) выбирается с тем же условием, чтобы получаемая последовательность была полной. Это достигается в тех случаях, когда параметр *c* принимает нечетные значения в диапазоне $c \in \left[\overline{1, 2^w - 1}\right]$.

Далее приведен код программы, подтверждающей рекомендации по выбору параметров *a* и *c* при максимально возможной генерации последовательностей чисел длиной $w = 4$ бита по вихревой технологии с матричной проверкой уникальности последовательностей. Функции *MatrixAdd()*, *MatrixCheck()* и *MatrixWrite()* взяты из раздела **Теория.**

```
static void Main(string[] args)
{    int w = 4;                              // битовая длина числа
     int N = 16;                          // длина последовательности
     Console.WriteLine("w = {0}   N = {1}", w, N);
     int maskW = (int)(0xFFFFFFFF >> (32 - w));          // маска
     int maskU = 1 << (w - 1);                       // старший бит
     Console.WriteLine("maskW = {0:X}   maskU = {1:X}",
                       maskW, maskU);
     int[] x = new int[N];        // случайная последовательность
     int[,] MS = new int[3000, N + 3];                 // матрица
     int M = 0;       // количество последовательностей в матрице
     for (int a = 1; a < N; a += 4)
         for (int c = 1; c < N; c += 2)
         {    int x0 = 1;
              Cong_Start(x, N, a, c, x0, maskW);
              if (Repeating(x, N)) continue;        // повторения
              MatrixAdd(MS, ref M, x, N, a, c);
              for (int i = 1; i < w * N; i++)
              {   Twist(x, w, N, maskW, maskU);          // вихрь
                  if (Repeating(x, N)) continue;
                  MatrixAdd(MS, ref M, x, N, a, c);
              }
         }
         MatrixCheck(MS, M, N, 1);         // проверка совпадений
         Console.WriteLine("Matrix of unique sequences");
         MatrixWrite(MS, M, N);                // монитор матрицы
```

```
        Console.WriteLine("Finish");
        Console.ReadKey();                // просмотр результата
}
```

После выполнения этой программы на мониторе появляется следующий листинг, который приведен с некоторыми сокращениями (прочерк на месте пропущенных строк). В конце каждой строки первый дополнительный элемент является результатом проверки, а два следующих показывают конгруэнтные константы *a* и *c*. Общее количество каждой пачки последовательностей оценивается как $w \cdot N = 4 \cdot 16 = 64$. Всего $w \cdot N \cdot 4_a \cdot 8_c = 4 \cdot 16 \cdot 4 \cdot 8 = 2048$ уникальных последовательностей.

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

Этот листинг показывает, что функция *Repeating()* не зафиксировала ни одной последовательности с повторяющимися элементами из всех возможных 2048 последовательностей. Первый дополнительный элемент в каждой строке указывает на это, поскольку содержит ноль для каждой сгенерированной последовательности. Другими словами, этот результат означает, что теперь можно генерировать все уникальные последовательности, используя такой тип параметра *a*. Как было указано ранее, этот результат достигается благодаря нечетным значениям конгруэнтной константы *c*. Таким образом, при любых допустимых значениях *a* и нечетных значениях *c* все сгенерированные случайные последовательности уникальны.

Конструкция и результаты. От рассмотренных в предыдущих разделах специальных аспектов перейдем к практической реализации. Далее представлено пространство имен *nsDeonYuliTwist28DA*, в котором класс *cDeonYuliTwist28DA* содержит необходимые параметры и функции генерации. Использование класса *cDeonYuliTwist28DA* показано в следующей программе *P020401*, представленной после этого программного кода.

```
namespace nsDeonYuliTwist28DA
{   class cDeonYuliTwist28DA
    {   public int w = 16;                  // битовая длина числа
        public int N = 0;              // длина последовательности
```

```
        public int N1 = 0;                  // максимальное число
        int[] x;       // конгруэнтно-вихревая последовательность
        public int x0 = 1;           // начало последовательности
        public double abf = 0.39;       // относительное начало a
        public double aef = 0.39;    // относительное окончание a
        public int a1b = 1, a1e = 0;               // интервал a1
        int a1s = 0;                     // состояние интервала a1
        public int a2b = 1, a2e = 0;               // интервал a2
        int a2s = 0;                     // состояние интервала a2
        int a1 = 5;                 // константа для интервала a1
        int a2 = 5;                 // константа для интервала a2
        int nA = 1;                  // номер константы a1 или a2
        public int a = 5;          // текущее значение константы a
        public double cbf = 0.1;        // относительное начало c
        public double cef = 0.9;     // относительное окончание c
        public int cb = 1, ce = 0;                  // интервал c
        public int c = 1;             // конгруэнтная константа c
        public int st = 1;                // состояние генератора
        int nW = 0;                     // номер вихревого сдвига
        int nT = 0;                         // номер вихря nT в N
        int nV = 0;                         // номер элемента в x
        uint maskW = 0U;                           // маска числа
        uint maskU = 0U;                   // маска старшего бита
//---------------------------------------------------------------
        public cDeonYuliTwist28DA()
        {   N = 1 << w;               // длина последовательности
            N1 = N - 1;                     // максимальное число
            x0 = N1 / 7;                      // начало генерации
        }
//---------------------------------------------------------------
        public int Next()
        {   bool flagW = true;                     // вечный цикл
            while (flagW)          // путешествие по состояниям st
            {   switch (st)          // переключение по состояниям
                {   case 1:            // инициализация генератора
                        nA = 1;     // генерация начинается на a1
                        a1s = 1;         // создать вихрь 0 на a1
                        a1 = a1e;       // окончание интервала a1
                        a = a1;            // текущая константа a
                        a2s = 0;        // a2 еще не используется
                        a2 = a2b - 4;// левее начала интервала a2
                        c = cb;             // начало интервала c
                        st = 2;            // конгруэнция вихря 0
                        break;
                    case 2:                  // начальный вихрь 0
                        DeonYuli_Cong(a,c);    // инициализация x
                        nW = 0;         // номер вихревого сдвига
                        nT = 0;            // номер вихря nT в nW
                        nV = 0;      // номер начального значения
                        st = 101;         // массив x подготовлен
                        break;
                    case 101:             // выборка из массива x
                        if (nV <= N1) flagW = false;// брать из x
                        else st = 102;          // заменить вихрь
                        break;
```

```
                case 102: // следующий вихрь с прежними a и c
                    nW++;     // следующий бит сдвига в слове
                    if (nW < w) { st = 103; break; }
                    nT++;//начало вихря со следующей величины
                    nW = 0;             // условно без сдвига
                    if (nT < N) st = 103;  // следующий вихрь
                    else st = 201;   // следующая константа c
                    break;
                case 103:                   // следующий вихрь
                    DeonYuli_Twist();      // следующий вихрь
                    nV = 0;      // номер начального значения
                    st = 101;         // выборка из массива x
                    break;
                case 201:              // заменить вихрь по c
                    c += 2;          // следующая константа c
                    if (c <= ce) st = 2;       // новый вихрь
                    else st = 202;   // следующая константа a
                    break;
                case 202:           // заменить интервал по a
                    c = cb;           // начальное значение c
                    if (nA == 1) nA = 2; else nA = 1;//замена
                    if (nA == 1) st = 203;     // интервал a1
                    else st = 204;             // интервал a2
                    break;
                case 203:             // новое значение из a1
                    a1 -= 4;
                    a = a1;
                    if (a1b <= a1) { a1s = 1; st = 2; }
                    else { a1s = 2; st = 205; }// a1 исчерпан
                    break;
                case 204:             // новое значение из a2
                    a2 += 4;
                    a = a2;
                    if (a2 <= a2e) { a2s = 1; st = 2; }
                    else { a2s = 2; st = 205; }// a2 исчерпан
                    break;
                case 205:        // один из a1 или a2 пройден
                    if (a2s != 2) st = 204;
                    else if (a1s != 2) st = 203;
                    else st = 1;              // общее начало
                    break;
            }  // switch
        }      // while
        return x[nV++];                // случайная величина
    }
//-------------------------------------------------------------
    void DeonYuli_Cong(int a, int c)
    {   x[0] = x0;              // начало последовательности
        for (int i = 1; i < N; i++)
            x[i] = (int)( (a * x[i - 1] + c) & maskW );
    }
//-------------------------------------------------------------
    void DeonYuli_Twist()
    {   uint z = (uint)((x[0] & maskU) >> (w - 1));  // левый
        for (int j = 0; j < N - 1; j++)
        {   uint g = (uint)((x[j + 1] & maskU) >> (w - 1));
```

```
                x[j] = (int)(((x[j] << 1) & maskW) | g);
            }
            x[N - 1] = (int)(((x[N - 1]<<1) & maskW)|z);// кольцо
        }
//---------------------------------------------------------------
        public void Start()
        {   N = 1 << w;               // длина последовательности
            N1 = N - 1;                       // максимальное число
            maskW = 0xFFFFFFFF >> (32 - w);        // маска числа
            maskU = (uint)(0x1 << (w - 1));// маска старшего бита
            DeonYuli_SetA();    // установить границы для a1 и a2
            DeonYuli_SetC();          // установить границы для c
            x0 &= (int)maskW;
            x = new int[N];          // массив последовательности
            st = 1;                    // инициализация генератора
        }
//---------------------------------------------------------------
        public void TimeStart()
        {   x0 = (int)DateTime.Now.Millisecond;   // миллисекунды
            Start();                            // старт генератора
        }
//---------------------------------------------------------------
        public void SetW(int sw)
        {   w = Math.Abs(sw);              // битовая длина числа
            if (w < 3) w = 3;                // минимальная длина
            else if (w > 28) w = 28;        // максимальная длина
        }
//---------------------------------------------------------------
        public void SetA(double sab, double sae)
        {   abf = Math.Abs(sab);
            aef = Math.Abs(sae);
            if (abf > 1.0) abf = 1.0;
            if (aef > 1.0) aef = 1.0;
            if (abf > aef) aef = abf;
        }
//---------------------------------------------------------------
        void DeonYuli_SetA()
        {   a1b = (int)(N1 * abf);         // нижняя грань для a1
            a1b = DeonYuli_PlusA(a1b);     // начало интервала a1
            a2e = (int)(N1 * aef);        // верхняя грань для a2
            a2e = DeonYuli_MinusA(a2e); // окончание интервала a2
            int r = a2e - a1b;
            if (a1b >= a2e)          // интервал a стянут в точку
            {   a1e = a1b;           // a1 состоит из одной точки
                a2b = a1b;          // интервал a2 совпадает с a1
                a2e = a2b;           // a2 состоит из одной точки
                return;
            }
            if (r == 4)                   // одноточечные a1 и a2
            {   a1e = a1b;           // a1 состоит из одной точки
                a2b = a2e;           // a2 состоит из одной точки
                return;
            }
            if (r == 8)      // a1 имеет 2 точки, a2 - одну точку
            {   a1e = a1b + 4;                    // окончание a1
                a2b = a2e;                           // начало a2
```

```
                return;
            }
            a1e = (a1b + a2e) / 2;              // середина для a
            a2b = a1e;
            a1e = DeonYuli_MinusA(a1e);      // слева от середины
            a2b = a1e + 4;                  // справа от середины
        }
//-------------------------------------------------------------
        int DeonYuli_PlusA(int a)
        {   if (a < 1) { a = 1; return a; }
            int z = a;                       // нижняя граница для a
            for (int i = 0; i < 3; i++)
                if (a % 4 != 0) a--;     // условие равномерности
                else break;
            a++;               // правильное значение константы a
            if (a < z) a += 4;        // справа от нижней границы
            if (a >= N1) a -= 4;      // слева от верхней границы
            return a;
        }
//-------------------------------------------------------------
        int DeonYuli_MinusA(int a)
        {   if (a < 1) { a = 1; return a; }
            int z = a;                       // нижняя граница для a
            for (uint i = 0; i < 3; i++)
                if (a % 4 != 0) a--;     // условие равномерности
                else break;
            a++;               // правильное значение константы a
            if (a > z) a -= 4;        // слева от верхней границы
            return a;
        }
//-------------------------------------------------------------
        public void SetC(double scb, double sce)
        {   cbf = Math.Abs(scb);
            cef = Math.Abs(sce);
            if (cbf > 1.0) cbf = 1.0;
            if (cef > 1.0) cef = 1.0;
            if (cbf > cef) cef = cbf;
        }
//-------------------------------------------------------------
        void DeonYuli_SetC()
        {   cb = (int)(N1 * cbf);           // нижняя грань для c
            if (cb % 2 == 0) cb += 1;        // только нечетное c
            if (cb > N1) cb = N1;        // максимальное значение
            ce = (int)(N1 * cef);          // верхняя грань для c
            if (ce % 2 == 0) ce -= 1;        // только нечетное c
            if (ce > N1) ce = N1;        // максимальное значение
            if (cb > ce) ce = cb;
            c = cb;            // начало конгруэнтной константы c
        }
//-------------------------------------------------------------
        public void SetX0(double xs)
        {   x0 = (int)(N1 * Math.Abs(xs));
        }
//=============================================================
    }
}
```

В классе *cDeonYuliTwist28DA* зарезервировано несколько переменных, которые можно настраивать с помощью инкапсулированных функций. В качестве первого примера воспользуемся значениями параметров по умолчанию для решения простейшей задачи генерации нескольких случайных величин, каждая длиной $w = 16$ бит из диапазона $\left[\overline{0, 2^w - 1}\right] = \left[\overline{0, 2^{16} - 1}\right] = \left[\overline{0, 65535}\right]$. Далее приведен код для решения этой задачи. Имена *P020401* и *cP020401* выбраны произвольно.

```
using nsDeonYuliTwist28DA;                  // вихревой генератор
namespace P020401
{   class P020401
    {   static void Main(string[] args)
        {   cDeonYuliTwist28DA CT = new cDeonYuliTwist28DA();
            CT.Start();                          // старт генератора
            for (int j = 0; j < 8; j++)
            {   int z = CT.Next();
                Console.Write("{0,7} ", z);            // монитор
            }
            Console.ReadKey();             // просмотр результата
        }
    }
}
```

После выполнения программы на мониторе появляется следующий результат:

9362 36699 52924 2805 8774 14575 51504 13129.

При генерации очередной новой исходной последовательности (вихрь 0) в классе *cDeonYuliTwist28DA* конгруэнтная константа *a* поочередно выбирается из двух разных интервалов, как показано на рис. 2.

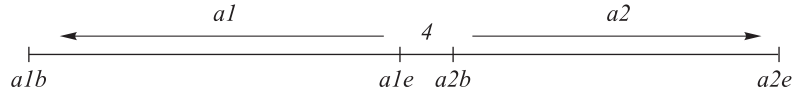

Рис. 2. Схема реализации интервала конгруэнтной константы *a*

Значение *a1e* находится слева от $N/2$, значение $a2b = a1e + 4$ — справа от $N/2$. Движение *a* в интервале *a1* производится справа налево от *a1e* к *a1b* с шагом –4. Движение *a* в интервале *a2* выполняется слева направо от *a2b* к *a2e* с шагом +4. Такой выбор конгруэнтной константы *a* предпринят искусственно, чтобы обеспечить лучшую перемешиваемость производимой генерации. Можно выбрать любой другой алгоритм, если есть необходимость в строгом управлении процесса генерации. Например, если использовать старт по датчику времени *TimeStart()*, то начальное значение последовательности определяется компонентой миллисекунд текущего времени.

Значение *a1b* задается параметром *b*, значение *a2e* — параметром *e* в интерфейсе их установки с помощью функции *SetA()*. При полной генерации кон-

груэнтный параметр *a* проходит все возможные значения в интервале $\left[\overline{1,N-3}\right]$ с шагом ±4. В этом случае обращение к функции настройки выглядит как *SetA(0.0, 1.0)*.

Далее представлен программный код, позволяющий настраивать генератор на различные значения *w*, *x0*, *a1b*, *a2e*. Конгруэнтный параметр *c* проходит все нечетные значения из интервала $\left[\overline{1,N-1}\right]$, начиная с единицы и заканчивая справа от $N-1$. Установка параметра *c* задается с помощью функции *SetC()*. Как было показано, первое число можно задать непосредственно, а следующие числа — по константе *c*. Технология вихря на полных последовательностях все равно обеспечит все неповторяющиеся числа в каждой новой последовательности. Итак, для примера в следующей программе: $w=4$, $N=2^w=16$, $a1b=1$, $a2e=13$. С каждым значением $a=5,9,1,13$ конгруэнтная константа $c=1,3,5,7,9,11,13,15$. *Имена P020403* и *cP020403* выбраны произвольно.

```
using nsDeonYuliTwist28DA;          // класс вихревого генератора
namespace P020403
{   class P020403
    {   static void Main(string[] args)
        {   cDeonYuliTwist28DA CT = new cDeonYuliTwist28DA();
            CT.SetW(4);
            CT.SetA(0.0, 1.0);           // полный диапазон для a
            CT.SetC(0.0, 1.0);           // полный диапазон для c
            CT.Start();
            Console.WriteLine("w = {0}   N = {1}", CT.w, CT.N);
            Console.WriteLine("a1b = {0}  a1e = {1}",
                              CT.a1b, CT.a1e);
            Console.WriteLine("a2b = {0}  a2e = {1}",
                              CT.a2b, CT.a2e);
            Console.WriteLine("cb = {0}  ce = {1}", CT.cb, CT.ce);
            int k = 0;                 // номер последовательности
            int NN = 0;           // количество случайных величин
            for (int nw = 0; nw < CT.w; nw++)
                for (int nt = 0; nt < CT.N; nt++)
                    for (int na = 1; na <= 4; na++)
                        for (int nc = 1; nc <= 8; nc++)
                        {   Console.Write("k={0,4} | ", ++k);
                            for (int i = 0; i < CT.N; i++)
                            {   Console.Write("{0,3}", CT.Next());
                                NN++;
                            }
                            Console.WriteLine(" a={0,2} c ={1,2}",
                                              CT.a, CT.c);
                            if (k % 250 == 0) Console.ReadKey();
                        }
            Console.WriteLine("Finish");
            Console.WriteLine("NN = {0}", NN);
            Console.ReadKey();
        }
    }
}
```

После выполнения этой программы на мониторе появляется следующий листинг, который приведем с некоторыми сокращениями, ставя прочерк на месте пропущенных строк.

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

Всего создано $4 \cdot 16 \cdot 4 \cdot 8 = 2048$ последовательностей. Поскольку каждая последовательность содержит 16 неповторяющихся случайных величин, то общее число сгенерированных величин составляет $2048 \cdot 16 = 32768$.

Обсуждение. В общем виде N_s сгенерированных полных неповторяющихся последовательностей зависит от следующих составляющих:

- битовой длины *w* случайных величин;
- количества $N = 2^w$ чисел в одной последовательности;
- количества $N_T = w \cdot N = w \cdot 2^w$ вихрей для каждой пары конгруэнтных констант *a* и *c*;
- числа возможных вариантов $N_a = N/4$ конгруэнтной константы *a*, числа $N_c = N/2$ возможных вариантов конгруэнтной константы *c*.

Собирая вместе эти выражения, получаем

$$N_s = N_T \cdot N_a \cdot N_c = w \cdot 2^w \cdot \frac{2^w}{4} \cdot \frac{2^w}{2} = w \frac{2^{3w}}{2^3} = w \cdot 2^{3w-3} = w \cdot 2^{3(w-1)}.$$

Количество N_{ns} генерируемых чисел во всех полных последовательностях определяется как

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

Битовая длина неповторения N_{bs} определяется как количество бит во всех числах всех неповторяющихся последовательностей

$$N_{bs} = w \cdot N_{ns} = w \cdot N \cdot N_s = w \cdot 2^w \cdot w \cdot 2^{3(w-1)} = w^2 \cdot 2^{4w-3}.$$

Вычисляя значение $N_s(w=4)$, получаем $N_s(w=4) = 4 \cdot 2^{3 \cdot 3} = 4 \cdot 512 = 2048$, что непосредственно подтверждается счетчиком *NN* прямого моделирования в программе *P020403* в разделе **Конструкция и результаты**.

В качестве примера рассмотрим часто используемые в практических разработках случайные величины длиной 16 бит. Это двухбайтные числа из диапазона $\left[\overline{0,\ 65535}\right]$. Число случайных величин в одной полной последовательности составляет $N\left(w=16\right)=2^w=2^{16}=65536$. Общее число неповторяющихся числовых последовательностей определяется как $N_s\left(w=16\right)=w\cdot 2^{3w-3}=2^4\cdot 2^{3\cdot 2^4-3}=$ $=2^{3\cdot 16+1}=2^{49}$. Тогда количество всех генерируемых чисел определяется как $N_{ns}\left(w=16\right)=N\cdot N_{ns}=2^4\cdot 2^{49}=2^{51}$. В этом случае битовая длина будет $N_{bs}\left(w=16\right)=w\cdot N_{ns}=2^4\cdot 2^{51}=2^{55}$.

Этот результат можно сравнить с оценкой для генераторов семейства MT19937 [13]. Просматривая программный код этого генератора, обнаруживаем, что битовая длина генерируемых случайных величин составляет $w_{32}=32$. Длина массива начальной конгруэнтной генерации задана как $MT_{32}=624$ элемента. Глобальный вихрь не использует кольцевой перенос старшего бита последовательности. Эти данные говорят о том, что полное число генерируемых случайных величин составляет

$$MT_{ns32}=w_{32}\cdot MT_{32}-31=32\cdot 624-31=19968-31=19937.$$

Вычитание константы 31 связано с тем, что вихрь не учитывает кольцо избыточных бит. Гипотетически в этих 19 937 переменных можно получить неповторяющуюся битовую последовательность в 2^{19937} бит. Однако это только гипотетически, поскольку конгруэнтный генератор вихря 0 с последующими однобитными вихрями позволяют получать только $MT_{ns}=19\,937$ чисел. Чтобы получить больше чисел, необходимо вручную задавать новое исходное число x_0, поскольку конгруэнтные константы *a* и *c* заданы стационарно и не подлежат автоматическому изменению.

Продолжим обсуждение настройки рассмотренного вихревого генератора. Необходимо явно убедиться, что все создаваемые случайные величины в генераторе *nsDeonYuliTwist28DA* повторяются одинаковое число раз. Поскольку условие равномерности предполагает, что все величины в полной генерации присутствуют одинаковое число раз, то в простейшем варианте в одной такой последовательности все числа встречаются лишь один раз. Следовательно, во множестве перестановок любое число встречается столько раз, сколько самих перестановок, так как в каждой равномерной перестановке любое число встречается только один раз. Это является прекрасным средством для подтверждения равномерности генераторов.

Далее приведен код, в котором каждые элементы массива *xC* являются соответствующими счетчиками случайных величин, т. е. индекс счетчика равен случайной величине. В программе проверяется равномерность вихря 0, который соответствует начальной конгруэнтной генерации. По умолчанию битовая длина каждой случайной величины принимается $w=16$ бит. Тогда длина полной последовательности составит $N=2^w=2^{16}=65\,536$ случайных величин.

Имена *P020501* и *cP020501* выбраны произвольно.

```
using nsDeonYuliTwist28DA;          // класс вихревого генератора
namespace P020501
{   class P020501
    {   static void Main(string[] args)
        {   cDeonYuliTwist28DA CT = new cDeonYuliTwist28DA();
            CT.Start();                       // запуск генератора
            Console.WriteLine("w = {0}  N = {1}", CT.w, CT.N);
            int[] cX = new int[CT.N];         // массив счетчиков
            for (int i = 0; i < CT.N; i++) cX[i] = 0;
            for (int n = 0; n < CT.N; n++)
            {   int z = CT.Next();
                cX[z]++;            // счетчик случайной величины
            }
            int count0 = 0;     // количество пропущенных величин
            int count1 = 0;     // количество однократных величин
            int count2 = 0;      // количество двукратных величин
            for (int i = 0; i < CT.N; i++)
            {   if (cX[i] == 1) count1++;          // однократные
                else if (cX[i] == 2) count2++;      // двукратные
                else if (cX[i] == 0) count0++;         // никогда
            }
            Console.Write("count0 = {0}   ", count0);
            Console.Write("count1 = {0}   ", count1);
            Console.WriteLine("count2 = {0}   ", count2);
            Console.ReadKey();             // просмотр результата
        }
    }
}
```

После выполнения программы следующие две строки появляются на мониторе:

```
w = 16   N = 65566
count0 = 0   count1 = 65536   count2 = 0
```

Чтобы убедиться в равномерности одного полного вихря при заданных по умолчанию конгруэнтных константах a и c, необходимо сгенерировать $nwN = w \cdot N = 16 \cdot 65536 = 1048576$ случайных величин. Следующий код выполняет такую работу. При этом каждая равномерно распределенная случайная величина должна появиться $w = 16$ раз. Имена *P020502* и *cP020502* выбраны произвольно.

```
using nsDeonYuliTwist28DA;          // класс вихревого генератора
namespace P020502
{   class P020502
    {   static void Main(string[] args)
        {   cDeonYuliTwist28DA CT = new cDeonYuliTwist28DA();
            CT.Start();                       // старт генератора
            int nwN = CT.w * CT.N;//число вихрей одной пары a и c
            Console.WriteLine("w = {0}  N = {1}  nwN = {2}",
                              CT.w, CT.N, nwN);
            int[] cX = new int[CT.N];         // массив счетчиков
            for (int i = 0; i < CT.N; i++) cX[i] = 0;
```

```
            for (int n = 0; n < nwN; n++)
            {   int z = CT.Next();
                cX[z]++;            // счетчик случайной величины
            }
            int count16 = 0;   // количество 16-тикратных величин
            for (int i = 0; i < CT.N; i++)
                if (cX[i] == 16) count16++;       // 16-тикратные
            Console.WriteLine("count16 = {0}   ", count16);
            Console.ReadKey();             // просмотр результата
        }
    }
}
```

После выполнения программы на мониторе появляется следующий результат:

```
w = 16   N = 65566  nwN = 1048576
count16 = 65536
```

Задача тестирования завершена, хотя исследование вихревого генератора требует специальных ресурсов, таких как мощные процессоры, дополнительная оперативная память, устройства внешней памяти и др. Однако принципы верификации остаются прежними.

Заключение. Первоначально авторы настоящей работы исходили из того факта, что конгруэнтный способ генерации случайных величин не может дать равномерное распределение для всех конгруэнтных констант и начальных значений по линейной конгруэнтной функции $x_{i+1} = (ax_i + c) mod\, m$. Однако результат может получиться равномерным, если рассматривать полные последовательности со случайными величинами в количестве $N = m$. В этом случае последовательности становятся полными с равномерным однократным распределением генерируемых случайных величин. Ради увеличения скорости вычислений операцию модуля *mod* на полных последовательностях можно заменить на битовую конъюнкцию & с маской, содержащей $w = \log_2 N$ битовых единиц. Вводя кольцевой перенос старшей битовой единицы при реализации левого глобального вихря, удалось получить $w \cdot N$ уникальных последовательностей, содержащих N неповторяющихся однократно равномерных случайных величин. При этом экспериментально с помощью матричной диагностики было выявлено, что всевозможное сочетание пар конгруэнтных констант a и c учитывает всевозможное задание исходных начальных величин x_0. Эти фундаментальные свойства позволили перейти к реализации настройки вихревых генераторов, указывая диапазоны конгруэнтных констант. Максимальный диапазон констант на полных последовательностях обеспечивает максимально возможную по количеству генерацию вихревых последовательностей и вихревых случайных величин. Приведенная конструкция вихревого генератора на основе исходного конгруэнтного массива обеспечивает равномерность создаваемых полных последовательностей случайных величин. Это может быть достаточно полезно для многих приложений, и в первую очередь для задач в таких областях, как информационные технологии, криптография, инженерное проектирование, биология, медицина и др.

ЛИТЕРАТУРА

1. *Cryptanalysis* of DES implemented on computers with cache / Y. Tusnoo, T. Saito, T. Suzaki, M. Shigeri, H. Miyauchi // Proc. Cryptographic Hardware and Embedded Systems — CHES 2003. 5th International Workshop. 2003. P. 62–76. DOI: 10.1007/978-3-540-45238-6_6
URL: http://link.springer.com/chapter/10.1007%2F978-3-540-45238-6_6

2. *Ozturk E., Sunar B., Savas E.* Low-power elliptic curve cryptography using scaled modular arithmetic // Proc. Cryptographic Hardware and Embedded Systems — CHES 2004. 6th International Workshop. P. 92–106. DOI: 10.1007/978-3-540-28632-5_7
URL: http://link.springer.com/chapter/10.1007%2F978-3-540-28632-5_7

3. *Panneton F., L'Ecuyer P., Matsumoto M.* Improved long-period generators based on linear recurrences modulo 2 // ACM TOMS. 2006. Vol. 32. No.1. P. 1–16.
DOI: 10.1145/1132973.1132974 URL: http://dl.acm.org/citation.cfm?doid=1132973.1132974

4. *Deon A.F., Menyaev Yu.A.* The complete set simulation of stochastic sequences without repeated and skipped elements // Journal of Universal Computer Science. 2016. Vol. 22. No. 8. P. 1023–1047. DOI: 10.3217/jucs-022-08-1023
URL: http://www.jucs.org/jucs_22_8/the_complete_set_simulation

5. *Entacher K.* Bad subsequences of well-known linear congruential pseudorandom number generators // ACM TOMACS. 1998. Vol. 8. No.1. P. 61–70. DOI: 10.1145/272991.273009
URL: http://dl.acm.org/citation.cfm?doid=272991.273009

6. *Leeb H., Wegenkittl S.* Inversive and linear congruential pseudorandom number generators in empirical tests // ACM TOMACS, 1997. Vol. 7. P. 272–286. DOI: 10.1145/249204.249208
URL: http://dl.acm.org/citation.cfm?doid=249204.249208

7. *Park S.K., Miller K.W.* Random number generators: good ones are hard to find // Communication of the ACM. 1988. Vol. 31. No. 10. P. 1192–1201. DOI: 10.1145/63039.63042
URL: http://dl.acm.org/citation.cfm?doid=63039.63042

8. *Optical* clearing in photoacoustic flowcytometry / Yu.A. Menyaev, D.A. Nedosekin, M. Sarimollaoglu, M.A. Juratli, E.I. Galanzha, V.V. Tuchin, V.P. Zharov // Biomed. Opt. Express. 2013. Vol. 4. No. 12. P. 3030–3041. DOI: 10.1364/BOE.4.003030
URL: https://www.osapublishing.org/boe/abstract.cfm?uri=boe-4-12-3030

9. *Wiese K.C., Hendriks A., Deschenes A., Youssef B.B.* The impact of pseudorandom number quality on P-RnaPredict, a parallel genetic algorithm for RNA secondary structure prediction // GECCO '05. Proc. 7th Annual Conf. on Genetic and Evolutionary Computation. 2005. P. 479–480. DOI: 10.1145/1068009.1068089
URL: http://dl.acm.org/citation.cfm?doid=1068009.1068089

10. *Leonard P., Jackson D.* Efficient evolution of high entropy RNGs using single node genetic programming // GECCO '15. Proc. of the 2015 Annual Conf. on Genetic and Evolutionary Computation. 2015. P. 1071–1078. DOI: 10.1145/2739480.2754820
URL: http://dl.acm.org/citation.cfm?doid=2739480.2754820

11. *Niederreiter H.* Some linear and nonlinear methods for pseudorandom number generation // WSC '95. Proc. of the 27th Conf. on Winter Simulation. 1995. P. 250–254.
DOI: 10.1145/224401.224611 URL: http://dl.acm.org/citation.cfm?doid=224401.224611

12. *Entacher K.* Parallel streams of linear random numbers in the spectral test // ACM TOMACS. 1999. Vol. 9. No.1. P. 31–44. DOI: 10.1145/301677.301682
URL: http://dl.acm.org/citation.cfm?doid=301677.301682

13. *Matsumoto M., Nishimura T.* Mersenne twister: a 623-dimensionnally equidistributed uniform pseudorandom number generator // ACM TOMACS. 1998. Vol. 8. No. 1. P. 3–30.
DOI: 10.1145/272991.272995 URL: http://dl.acm.org/citation.cfm?doid=272991.272995

14. *Nishimura T.* Tables of 64-bit Mersenne twisters // ACM TOMACS. 2000. Vol. 10. No. 4. P. 348–357. DOI: 10.1145/369534.369540
URL: http://dl.acm.org/citation.cfm?doid=369534.369540

15. *Makino J.* Lagged-Fibonacci random number generators on parallel computers // Parallel Comput. 1994. Vol. 20. No. 9. P. 1357–1367. DOI: 10.1016/0167-8191(94)90042-6
URL: http://www.sciencedirect.com/ science /article/pii/0167819194900426?via%3Dihub

16. *Aluru S.* Lagged Fibonacci random number generators for distributed memory parallel computers // J. Parallel Distr. Com. 1997. Vol. 45. No. 1. P. 1–12. DOI: 10.1006/jpdc.1997.1363
URL: http://www.sciencedirect.com/science/article/ pii/S0743731597913630?via%3Dihub

17. *Blum L., Blum M., Shub M.* A Simple unpredictable pseudo-random number generator // SIAM Journal on Computing.1986. Vol. 15. No. 2. P. 364–383. DOI: 10.1137/0215025
URL: http://epubs.siam.org/doi/10.1137/0215025

18. *Schildt H.* C# 4.0: the complete reference. New York: The McGraw-Hill Companies, 2010. 949 p.

19. *Saito M., Matsumoto M.* SIMD-oriented fast Mersenne twister: a 128-bit pseudorandom number generator // Monte Carlo and quasi-Monte Carlo methods 2006. Heidelberg, Berlin: Springer, 2008. P. 607–622. DOI: 10.1007/978-3-540-74496-2_36
URL: http://link.springer.com/chapter/10.1007%2F978-3-540-74496-2_36

20. *Lewis T.G., Payne W.H.* Generalized feedback shift register pseudorandom number algorithm // J. ACM. 1973. Vol. 20. No. 3. P. 456–486. DOI: 10.1145/321765.321777
URL: http://dl.acm.org/citation.cfm?doid=321765.321777

21. *Chandrasekaran S., Amira A.* High performance FPGA implementation of the Mersenne twister // DELTA. IEEE International Workshop on Electronic Design, Test and Applications. 2008. P. 482–485. DOI: 10.1109/DELTA.2008.113 URL: http://ieeexplore.ieee.org/document/4459598

22. *Pellicer-Lostao C., Lopez-Ruiz R.* Pseudo-random bit generation based on 2D chaotic maps of logistic type and its applications in chaotic cryptography // Proc. Computational Science and its Applications — ICCSA 2008 Int. Conf. 2008. Part. II. Vol. 5073. P. 784–796.
DOI: 10.1007/978-3-540-69848-7_62
URL: http://link.springer.com/chapter/10.1007%2F978-3-540-69848-7_62

23. *Bos J.W., Kleinjung T., Lenstra A.K., Montgomery P.L.* Efficient SIMD arithmetic modulo a Mersenne number // ARITH '11. Proc. IEEE 20th Symposium on Computer Arithmetic. 2011. P. 213–221. DOI: 10.1109/ARITH.2011.37 URL: http://ieeexplore.ieee.org/document/5992129

24. *Rahimov H., Babaie M., Hassanabadi H.* Improving middle square method RNG using chaotic map // Appl. Math. 2011. Vol. 2. P. 482–486. DOI: 10.4236/am.2011.24062
URL: http://file.scirp.org/pdf/AM20110400016_12226152.pdf

25. *Deon A.F., Menyaev Yu.A.* Parametrical tuning of twisting generator // Journal of Computer Science. 2016. Vol. 12. No. 8. P. 363–378. DOI: 10.3844/jcssp.2016.363.378
URL: http://thescipub.com/abstract/10.3844/jcssp.2016.363.378

Деон Алексей Федорович — канд. техн. наук, доцент кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана (Российская Федерация, 105005, Москва, 2-я Бауманская ул., д. 5).

Меняев Юлиан Алексеевич — канд. техн. наук, сотрудник Института исследования рака им. Уинтропа Рокфеллера, Арканзасский Университет Медицинских Наук (Литл-Рок, Арканзас, США).

Просьба ссылаться на эту статью следующим образом:

Деон А.Ф., Меняев Ю.А. Генератор равномерных случайных величин по технологии полного вихревого массива // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2017. № 2. С. 86–110. DOI: 10.18698/0236-3933-2017-2-86-110

UNIFORM RANDOM QUANTITY GENERATOR USING COMPLETE TWISTER ARRAY TECHNOLOGY

A.F. Deon[1] deonalex@mail.ru
Yu.A. Menyaev[2] yamenyaev@uams.edu

[1] Bauman Moscow State Technical University, Moscow, Russian Federation
[2] Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Science, Little Rock, AR, USA

Abstract

Uniformly distributed random quantity generators are widely used in various applications ranging from mathematics, radio-electronic and technical designing to medical and biological research. This paper proposes a novel approach to generating random quantities by combining the initial congruent array with a global circular vortex for complete stochastic sequences. We experimentally confirmed that for the complete sequences generation of this type provides random quantity uniform distribution. The proposed software includes the generation technique tuning methods where random quantities may take any bit length. Moreover, we examined the automatic switching of such generator parameters as initial values and congruent constants, which allowed us to increase the amount of the options of the generated sequences. Findings of the research confirm the absolute uniformity of distribution without any repeated or skipped elements in generated sequences of random quantities

Keywords

Pseudorandom quantity generator, random sequences, congruent quantity, vortex generator

REFERENCES

[1] Tusnoo Y., Saito T., Suzaki T., Shigeri M., Miyauchi H. Cryptanalysis of DES implemented on computers with cache. *Proc. Cryptographic Hardware and Embedded Systems — CHES 2003. 5th Int. Workshop*, 2003, pp. 62–76. DOI: 10.1007/978-3-540-45238-6_6
Available at: http://link.springer.com/chapter/10.1007%2F978-3-540-45238-6_6

[2] Ozturk E., Sunar V., Savas E. Low-power elliptic curve cryptography using scaled modular arithmetic. *Proc. Cryptographic Hardware and Embedded Systems — CHES 2004. 6th Int. Workshop*, 2004, pp. 92–106. DOI: 10.1007/978-3-540-28632-5_7
Available at: http://link.springer.com/chapter/10.1007%2F978-3-540-28632-5_7

[3] Panneton F., L'Ecuyer R., Matsumoto M. Improved long-period generators based on linear recurrences modulo 2. *ACM TOMS,* 2006, vol. 32, no. 1, pp. 1–16. DOI: 10.1145/1132973.1132974
Available at: http://dl.acm.org/citation.cfm?doid=1132973.1132974

[4] Deon A.F., Menyaev Yu.A. The complete set simulation of stochastic sequences without repeated and skipped elements. *Journal of Universal Computer Science*, 2016, vol. 22, no. 8, pp. 1023–1047. DOI: 10.3217/jucs-022-08-1023
Available at: http://www.jucs.org/jucs_22_8/the_complete_set_simulation

[5] Entacher K. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM TOMACS*, 1998, vol. 8, no. 1, pp. 61–70. DOI: 10.1145/272991.273009
Available at: http://dl.acm.org/citation.cfm?doid=272991.273009

[6] Leeb H., Wegenkittl S. Inversive and linear congruential pseudorandom number generators in empirical tests. *ACM TOMACS*, 1997, vol. 7, pp. 272–286. DOI: 10.1145/249204.249208
Available at: http://dl.acm.org/citation.cfm?doid=249204.249208

[7] Park S.K., Miller K.W. Random number generators: good ones are hard to find. *Communication of the ACM*, 1988, vol. 31, no. 10, pp. 1192–1201. DOI: 10.1145/63039.63042
Available at: http://dl.acm.org/citation.cfm?doid=63039.63042

[8] Menyaev Yu.A., Nedosekin D.A., Sarimollaoglu M., Juratli M.A., Galanzha E.I., Tuchin V.V., Zharov V.P. Optical clearing in photoacoustic flowcytometry. *Biomed. Opt. Express*., 2013, vol. 4, no. 12, pp. 3030–3041. DOI: 10.1364/BOE.4.003030
Available at: https://www.osapublishing.org/boe/abstract.cfm?uri=boe-4-12-3030

[9] Wiese K.C., Hendriks A., Deschenes A., Youssef V.V. The impact of pseudorandom number quality on P-RnaPredict, a parallel genetic algorithm for RNA secondary structure prediction. *GECCO '05. Proc. 7th Annual Conf. on Genetic and Evolutionary Computation*, 2005, pp. 479–480. DOI: 10.1145/1068009.1068089
Available at: http://dl.acm.org/citation.cfm?doid=1068009.1068089

[10] Leonard P., Jackson D. Efficient evolution of high entropy RNGs using single node genetic programming. *GECCO '15. Proc. of the 2015 Annual Conf. on Genetic and Evolutionary Computation*, 2015, pp. 1071–1078. DOI: 10.1145/2739480.2754820
Available at: http://dl.acm.org/citation.cfm?doid=2739480.2754820

[11] Niederreiter H. Some linear and nonlinear methods for pseudorandom number generation. *WSC '95. Proc. of the 27th Conf. on Winter Simulation*, 1995, pp. 250–254.
DOI: 10.1145/224401.224611 Available at: http://dl.acm.org/citation.cfm?doid=224401.224611

[12] Entacher K. Parallel streams of linear random numbers in the spectral test. *ACM TOMACS*, 1999, vol. 9, no. 1, pp. 31–44. DOI: 10.1145/301677.301682
Available at: http://dl.acm.org/citation.cfm?doid=301677.301682

[13] Matsumoto M., Nishimura T. Mersenne twister: a 623-dimensionnally equidistributed uniform pseudorandom number generator. *ACM TOMACS*, 1998, vol. 8, no. 1, pp. 3–30.
DOI: 10.1145/272991.272995 Available at: http://dl.acm.org/citation.cfm?doid=272991.272995

[14] Nishimura T. Tables of 64-bit Mersenne twisters. *ACM TOMACS*, 2000, vol. 10, no. 4, pp. 348–357. DOI: 10.1145/369534.369540
Available at: http://dl.acm.org/citation.cfm?doid=369534.369540

[15] Makino J. Lagged-Fibonacci random number generators on parallel computers. *Parallel Comput*., 1994, vol. 20, no. 9, pp. 1357–1367. DOI: 10.1016/0167-8191(94)90042-6
Available at: http://www.sciencedirect.com/ science/article/pii/0167819194900426?via%3Dihub

[16] Aluru S. Lagged Fibonacci random number generators for distributed memory parallel computers. *J. Parallel Distr. Com*., 1997, vol. 45, no. 1, pp. 1–12. DOI: 10.1006/jpdc.1997.1363
Available at: http://www.sciencedirect.com/ science/article/pii/S0743731597913630?via%3Dihub

[17] Blum L., Blum M., Shub M. A Simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 1986, vol. 15, no. 2, pp. 364–383. DOI: 10.1137/0215025
Available at: http://epubs.siam.org/doi/10.1137/0215025

[18] Schildt H. C# 4.0: the complete reference. New York, The McGraw-Hill Companies, 2010, 949 p.

[19] Saito M., Matsumoto M. SIMD-oriented fast Mersenne twister: a 128-bit pseudorandom number generator. In: Monte Carlo and quasi-Monte Carlo methods 2006. Heidelberg, Berlin, Springer, 2008, pp. 607–622. DOI: 10.1007/978-3-540-74496-2_36
Available at: http://link.springer.com/chapter/10.1007%2F978-3-540-74496-2_36

[20] Lewis T.G., Payne W.H. Generalized feedback shift register pseudorandom number algorithm. *J. ACM.,* 1973, vol. 20, no. 3, pp. 456–486. DOI: 10.1145/321765.321777
Available at: http://dl.acm.org/citation.cfm?doid=321765.321777

[21] Chandrasekaran S., Amira A. High performance FPGA implementation of the Mersenne twister. *DELTA. IEEE Int. Workshop on Electronic Design, Test and Applications*, 2008, pp. 482–485. DOI: 10.1109/DELTA.2008.113
Available at: http://ieeexplore.ieee.org/document/4459598

[22] Pellicer-Lostao C., Lopez-Ruiz R. Pseudo-random bit generation based on 2D chaotic maps of logistic type and its applications in chaotic cryptography. *Proc. Computational Science and its Applications — ICCSA 2008 Int. Conf. Part. II*. 2008, vol. 5073, pp. 784–796.
DOI: 10.1007/978-3-540-69848-7_62
Available at: http://link.springer.com/chapter/10.1007%2F978-3-540-69848-7_62

[23] Bos J.W., Kleinjung T., Lenstra A.K., Montgomery P.L. Efficient SIMD arithmetic modulo a Mersenne number. *ARITH '11. Proc. IEEE 20th Symp. on Computer Arithmetic*, 2011, pp. 213–221. DOI: 10.1109/ARITH.2011.37
Available at: http://ieeexplore.ieee.org/document/5992129

[24] Rahimov H., Babaie M., Hassanabadi N. Improving middle square method RNG using chaotic map. *Appl. Math.,* 2011, vol. 2, pp. 482–486. DOI: 10.4236/am.2011.24062
Available at: http://file.scirp.org/pdf/AM20110400016_12226152.pdf

[25] Deon A.F., Menyaev Yu.A. Parametrical tuning of twisting generator. *Journal of Computer Science*, 2016, vol. 12, no. 8, pp. 363–378. DOI: 10.3844/jcssp.2016.363.378
Available at: http://thescipub.com/abstract/10.3844/jcssp.2016.363.378

Deon A.F. — Cand. Sc. (Eng.), Assoc. Professor of Computer Software and Information Technology Department, Bauman Moscow State Technical University (2-ya Baumanskaya ul. 5, Moscow, 105005 Russian Federation).
Menyaev Yu.A. — Cand. Sc. (Eng.), works in Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Science (Little Rock, AR, USA).

Please cite this article in English as:

Deon A.F., Menyaev Yu.A. Uniform Random Quantity Generator using Complete Twister Array Technology. *Vestn. Mosk. Gos. Tekh. Univ. im. N.E. Baumana, Priborostr.* [Herald of the Bauman Moscow State Tech. Univ., Instrum. Eng.], 2017, no. 2, pp. 86–110.
DOI: 10.18698/0236-3933-2017-2-86-110