УДК 681.3.06 DOI: 10.18698/0236-3933-2018-3-51-69

# ГЕНЕРАТОР РАВНОМЕРНЫХ ВИХРЕВЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЦЕЛЫХ СЛУЧАЙНЫХ ВЕЛИЧИН БЕЗ ЗАПОМИНАЮЩЕГО 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**

Вихревые генераторы целых случайных величин могут привлекать массивы для внутренней реализации операций вихревого сдвига. Такая технология дает отличные результаты, когда используются массивы случайных величин в последовательностях малого и среднего размера. Однако в современных генераторах наблюдается тенденция к достаточно длинным последовательностям, массивы которых могут потребовать всю оперативную память компьютера. В таких вариантах прежняя технология применения массивов непригодна, поскольку тогда в компьютере не остается места для самих программ и операционной системы. Предложена новая технология генерации вихревых последовательностей без использования конгруэнтно-вихревых массивов. Это позволяет создавать вихревые равномерные целые случайные последовательности произвольного размера, не занимая оперативную память компьютера. Результаты моделирования подтверждают, что получаемые случайные величины имеют абсолютно равномерное распределение во множестве уникальных последовательностей. Кроме того, комбинирование этого нового подхода с алгоритмом настройки вихревой генерации позволяет существенно увеличить длину создаваемых последовательностей без использования дополнительной оперативной памяти компьютера

**Ключевые слова**

*Компьютерное моделирование, генераторы случайных величин, алгоритмы стохастических последовательностей*

Поступила в редакцию 05.02.2018


**Введение.** Равномерные последовательности целых случайных величин широко используют в криптографии [1–3], в тестировании технических систем [4–7], в анализе телетрафика [8, 9], в теоретическом моделировании природных процессов [10–12], в теоретической математике [13–16], в проверочных исследованиях биологии [17–19] и медицины [20, 21]. Ранее длина 15 или 16 бит для чисел из интервалов $[0:2^{15}-1]=[0:32767]$ или $[0:2^{16}-1]=[0:65535]$ соответственно была достаточной для используемых случайных величин. Однако современные исследования требуют длины 32…64 бита в качестве диапазонов случайных величин. В таких ситуациях использование технологий вихревой генерации на

основе конгруэнтных массивов [22–27] часто становится невозможным. Рассмотрим это подробнее.

Для того чтобы создать вихрь, необходима конгруэнтная генерация. Она определяет случайную величину $x_{i+1}$, следующую за текущей величиной $x_i$, используя функцию $f(x_i)$, ограниченную модулем $m$:

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

Исторически функция $f(x_i)$ линейно зависит от конгруэнтных констант $a$ и $c$:

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

В генераторе *MT19937* [22] реализация индекса $i$ сопровождается массивом *mt*, который необходимо разместить и инициализировать в оперативной памяти *RAM* компьютера с помощью следующего кода:

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

Элементы в массиве *mt* содержат случайные величины, созданные по линейной конгруэнтной формуле (2) с константами $a = 69069$ и $c = 0$. Качество получаемых величин генерации определяется ортогональным преобразованием матрицы [23, 24]. Интересно адресное пространство, которое требуется для 624 слов длиной 4 байта или 32 бита (в то время тип *long* обозначал 4 байта, позднее это стало 8 байт). Однако этот размер может оказаться незначительным для реализации некоторых задач в адресном пространстве $\log_2(624 \cdot 4) = \log_2 2496 < 12$ бит общей шины данных компьютера или микроконтроллера.

В работах [29, 30] было показано, что абсолютная равномерность генерации достигается только на полных последовательностях случайных величин. В этом случае каждая полная последовательность содержит неповторяющиеся случайные величины длиной $w$ бит. Интервал генерации определяется длиной $w$ каждого числа $x \in \left[\overline{0:2^w - 1}\right]$. В таком интервале глобальный кольцевой вихрь обеспечивает абсолютную равномерность генерации на исходном конгруэнтном массиве:

```
static public int w = 24;                          // number bit length
static public int N = 1 << w;                        // sequence length
static public int[] x = new int[N];                              // sequence
static public int maskW = (int)(0xFFFFFFFF >> (32 - w));
x[0] = x0;                                  // the beginning of sequence
for ( int i = 1; i < N; i++ )
        x[i] = ( a * x[i - 1] + c ) & maskW;
```

В приведенном листинге длина случайной величины составляет 24 бита. Следовательно, конгруэнтная генерация обеспечивает интервал всех чисел $x \in [0:2^w - 1] = [0:2^{24} - 1] = [0:16777215]$. Все они присутствуют в массиве $x$ ровно 1 раз. Но что делать, если необходимо генерировать числа произвольной битовой длины $w$? С одной стороны, если применяется технология *MT19937*, то нет гарантии, что будут созданы все числа с непредсказуемым пропуском чисел в равномерной генерации. С другой стороны, если использовать технологию полных последовательностей, то массив может не поместиться в *RAM* компьютера или микроконтроллера, т. е. если длина числа составляет $w = 32$ бита, то 32-адресные шины не оставят места для программы, поскольку все байты заняты массивом конгруэнтно-вихревой генерации.

*Цель настоящей работы* — найти решение для выполнения генерации полных последовательностей равномерно распределенных случайных величин в интервале $[0:2^w - 1]$ с произвольной битовой длиной $w \in [3:32]$. Другими словами, необходимо отказаться от технологии применения массива для хранения случайных величин в вихревой генерации.

**Теория.** Рассмотрим последовательность чисел одинаковой длиной $w$ бит. Начальное число обозначим как $x_0$. Остальные числа можно вычислить с помощью конгруэнтных формул (1) и (2). Если число уникальных неповторяющихся чисел равно $2^w$, то такую последовательность называют полной, поскольку она содержит все числа из интервала $[0:2^w - 1]$. Экспериментально было подтверждено [26, 27], что конгруэнтные последовательности длиной $2^w$ могут обладать свойством полноты, если конгруэнтная константа $a$ подчиняется условию $(a-1) \bmod 4 = 0$, и константой $c$, которая должна быть нечетной: $c \bmod 2 \neq 0$. Обе константы $a$ и $c$ не выходят за интервал полноты $[0:2^w - 1]$.

В полных конгруэнтных последовательностях глобальные кольцевые вихри также создают полные последовательности. Кроме того, следует учесть, что полные последовательности имеют равномерное единичное распределение своих элементов. Это непосредственно следует из определения полноты. Перечисленных свойств полных конгруэнтных и вихревых последовательностей достаточно для дальнейших построений.

Задача состоит в том, чтобы отказаться от исходного конгруэнтного массива, но обеспечить полную конгруэнтно-вихревую генерацию равномерно распределенных случайных последовательностей.

Пусть имеется начальное число $x0 \in [1:2^w - 1]$ с битовой маской $mask\ W = 0xFFFFFFFF \gg (32 - w)$. Представим пару $< xL, xR >$ смежных (левое и правое) конгруэнтных чисел. Примем левое значение как $xL = x0$. Правое значение определяется конгруэнцией $xR = (axL + c) \,\&\, \text{mask}\, W$. Конгруэнтная последовательность будет создана, если последовательно выполнить две операции:

```
xL = xR;
xR = ( a * x + c ) & maskW.
```

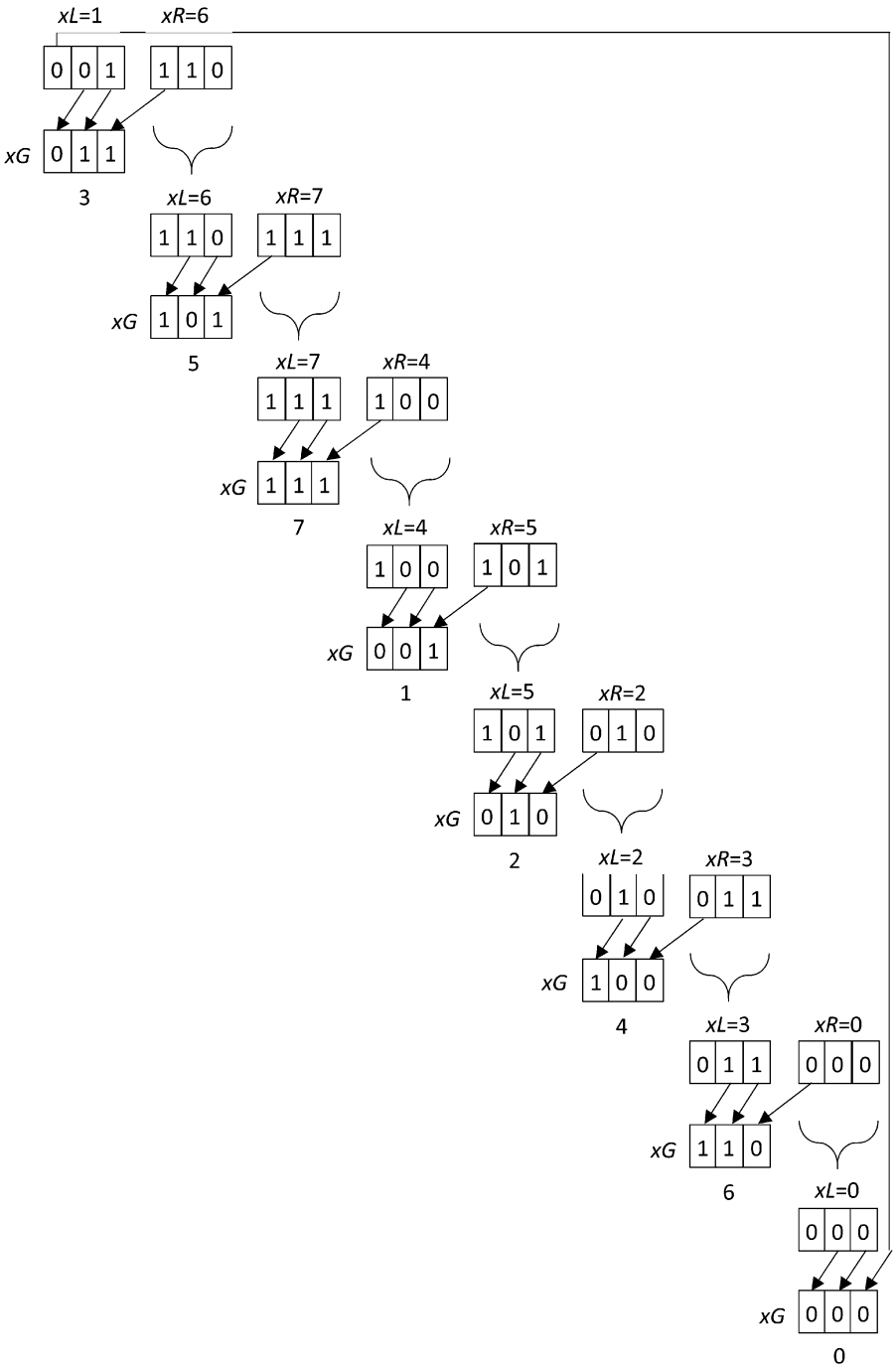


**Рис. 1.** Схема генерации глобального вихря 1 без конгруэнтного массива

Число итераций на единицу меньше числа чисел в последовательности, поскольку начальное число совпадает с началом последовательности $xL = x0$. Таким образом, для генерации конгруэнтного вихря 0 можно не применять рабочий массив. Очередная случайная величина вычисляется на очередной итерации.

Перейдем к генерации вихря 1, который по определению является глобальным левым кольцевым сдвигом на 1 бит конгруэнтной последовательности (вихрь 0). Это отображено на схеме генерации (рис. 1) с последовательностью чисел длиной $w = 3$ бита. Длина полной последовательности составляет $N = 2^w = 2^3 = 8$. Примем $x_0 = 1$, $a = 5$, $c = 1$. В таком варианте конгруэнтная генерация создает последовательность 1 6 7 4 5 2 3 0. Основываясь на этом, вихрь 1 создает следующую последовательность 3 5 7 1 2 4 6 0, элементы которой представлены как $xG$ (см. рис. 1). В начальной итерации $xL = 1$, $xR = 6$. Совместный левый сдвиг на 1 бит создает число $xG = 3$. Вторая итерация начинается с того, что значение из переменной $xR$ копируется в $xL = xR = 6$. В то же время новое значение $xR$ вычисляется по (2) конгруэнтным образом $xR = (axL + c) \operatorname{mask} W = (5 \cdot 6 + 1) \& 111_2 = 7$. По аналогии с первой итерацией второе число вихря $xG = 5$ получается путем совместного сдвига на 1 бит новых значений пары $xL = 6$, $xR = 7$. Повторяя все итерации в общей сложности 8 раз, получаем все восемь чисел $xG$ полной вихревой последовательности.

Очевиден тот факт, что получаемая полная последовательность случайных величин не ограничена техническими возможностями применяемых средств вычислительной техники. Однако следует учитывать, что хотя длина полной последовательности может быть произвольной, битовая длина $w$ самих случайных величин ограничена возможностями логических операций используемых на компьютерах с шинами 32 или 64 бита.

**Конструкция и испытания.** При создании вихревых последовательностей большое значение имеет выбор констант $a$ и $c$ в конгруэнтной формуле (2) для генерации случайных величин длиной $w$ бит. Обе константы должны принадлежать интервалу $a, c \in [1 : 2^w - 1]$ и обладать следующими свойствами:

– $(a - 1) \bmod w = 0$;

– $c \bmod 2 \neq 0$ — только нечетные значения.

Автоматическую настройку константы $a$ по задаваемому подынтервалу $[ab : ae] \subset [1 : 2^w - 1]$ можно выполнить различными способами. В настоящей работе применен алгоритм двуинтервального моделирования константы $a$. Схема реализации двух подынтервалов $a1 + a2 = [a1b : a1e] + [a2b : a2e] \subset [1 : 2^w - 1]$ приведена на рис. 2.

Значение $a1e$ расположено слева от $N / 2$, значение $a2b = a1e + 4$ — справа от $N / 2$. Движение $a$ в интервале $a1$ выполняется справа налево — от $a1e$ к $a1b$ с шагом −4, движение $a$ в интервале $a2$ — слева направо — от $a2b$ к $a2e$ с шагом +4. Такой выбор конгруэнтной константы $a$ предпринят искусственно, чтобы обеспечить лучшую перемешиваемость производимой генерации.


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

Конгруэнтная константа *c* проходит все нечетные значения из интервала $[1:2^w-1]$. Кодовые состояния процесса генерации случайных величин с автоматической настройкой конгруэнтных констант и масок вихрей показаны на рис. 3. Назначения состояний следующее:

– состояние 1 начинает общую генерацию;

– состояние 2 устанавливает параметры блока 1 для продолжения генерации после изменения конгруэнтных констант;

– состояние 101 обеспечивает конгруэнтную генерацию случайных величин в полной последовательности;

– состояние 102 настраивает параметры для вихревой генерации;

– состояние 103 занимается вихревой генерацией случайных величин в полной последовательности;

– состояние 104 устанавливает конгруэнтное начало следующей конгруэнтно-вихревой полной последовательности;

– состояние 105 обеспечивает переход к блоку 2 для изменения конгруэнтных констант;

– состояние 201 изменяет конгруэнтную константу *c*;

– состояние 202 определяет подынтервал *a*1 или *a*2 конгруэнтной константы *a*;

– состояние 203 вычисляет новое значение *a* в подынтервале *a*1;

– состояние 204 вычисляет новое значение *a* в подынтервале *a*2;

– состояние 205 завершает общую генерацию и формирует переход в состояние 1.

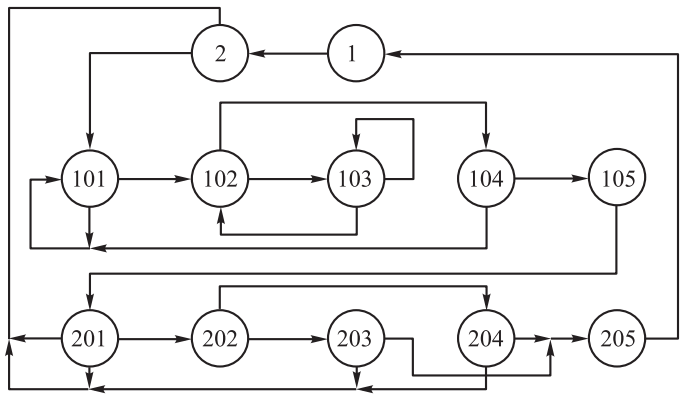


**Рис. 3.** Кодовые состояния при настройке генерации случайных величин

Код пространства имен *nsDeonYuliTwist32D*, в котором динамический класс *cDeonYuliTwist32D* обеспечивает вихревую генерацию случайных величин с произвольной длиной до 32 бит включительно, а также тестирование этого класса в программе *P030401* представлены ниже:

```
namespace nsDeonYuliTwist32D
{  class cDeonYuliTwist32D
   {  public uint w = 16U;               // битовая длина числа
      public uint N1 = 0U;                // максимальное число
      public uint x0 = 1U;          // начало последовательности
      uint xB = 1U;                   // начало очередного вихря
      uint xG = 0U;              // созданная случайная величина
      uint xL = 0U, xR = 1U;                  // парные величины
      double abf = 0.39;               // относительное начало a
      double aef = 0.39;            // относительное окончание a
      public uint a1b = 1U, a1e = 0U;             // интервал a1
      uint a1s = 0U;                   // состояние интервала a1
      public uint a2b = 1U, a2e = 0U;             // интервал a2
      uint a2s = 0U;                   // состояние интервала a2
      uint a1 = 5U;                // константа для интервала a1
      uint a2 = 5U;                // константа для интервала a2
      uint nA = 1U;                 // номер константы a1 или a2
      public uint a = 5U;        // текущее значение константы a
      double cbf = 0.1;                // относительное начало c
      double cef = 0.3;             // относительное окончание c
      public uint cb = 1U, ce = 0U;                // интервал c
      public uint c = 1U;           // конгруэнтная константа c
      uint stG = 0U;                   // номер группы состояний
      uint st0 = 1U;               // группа начальных состояний
      uint st1 = 101U;                    // группа генерации xG
      uint st2 = 201U;                // группа смены параметров
      public uint nW = 0U;            // номер парного вихря в w
      public uint nT = 0U;                          // номер вихря
      public uint nV = 0U;                 // номер элемента в x
      public uint maskW = 0U;                       // маска числа
      public uint maskU = 0U;             // маска старшего бита
      public uint maskT = 0U;                    // вихревые биты
//-------------------------------------------------------------
      public cDeonYuliTwist32D()
      {  N1 = 0xFFFFFFFF >> (32 - (int)w);// максимальное число
         x0 = N1 / 7;              // начало последовательности
      }
//-------------------------------------------------------------
      public uint Next()
      {  bool FlagNext = true;
         while (FlagNext)
         {  switch (stG)                       // группы состояний
            {  case 0U:           // группа начальных состояний
                  FlagNext = DeonYuli_Next0();
                  break;
               case 1U:                    // группа генерации xG
                  FlagNext = DeonYuli_Next1();
                  break;
               case 2U:               // группа смены параметров
                  FlagNext = DeonYuli_Next2();
                  break;
            }  // switch
         }     // while
         return xG;             // созданная случайная величина
      }
//-------------------------------------------------------------
```

```
      bool DeonYuli_Next0()
      {  bool FlagWhile0 = true;        // установка параметров
         while (FlagWhile0)            // переход по состояниям
         {  switch (st0)          // выбор установки параметров
            {  case 1U:                    // начальные действия
                  nA = 1U;         // генерация начинается на a1
                  a1s = 1U;             // создать вихрь 0 на a1
                  a2s = 0U;           // a2 пока не используется
                  a1 = a1e;            // окончание интервала a1
                  a = a1;                 // текущая константа a
                  a2 = a2b - 4U;    // левее начала интервала a2
                  c = cb;                  // начало интервала c
                  st0 = 2U;               // параметры генерации
                  break;
               case 2U:             // при измененных параметрах
                  xB = x0;          // начало последовательности
                  xR = xB;              // окончание пары xL, xR
                  nT = 0U;                       // номер вихря 0
                  nW = 0U;                     // номер вихря в w
                  nV = 0U;          // номер начального значения
                  stG = 1U;               // группа генерации xG
                  st1 = 101U;                  // генерация xG
                  FlagWhile0 = false;        // завершить работу
                  break;
            }   // switch
         }       // while
         return true;                // необходима генерация xG
      }
//-------------------------------------------------------------
      bool DeonYuli_Next1()
      {  bool FlagNext1 = false;             // xG будет создано
         bool FlagWhile1 = true;                  // поиск вихря
         while (FlagWhile1)         // переходы по состояниям st
         {  switch (st1)           // переключение по состояниям
            {  case 101U:              // конгруэнтная генерация
                  xL = xR;                          // начало пары
                  xR = DeonYuli_Cong(xL);         // пара xL, xR
                  xG = xL;           // сгенерированная величина
                  if (nV < N1) nV++;          // следующий номер
                  else st1 = 102U;
                  FlagWhile1 = false;           // число создано
                  break;
               case 102U:                // для парного вихря nW
                  nW++;               // номер парного вихря в w
                  if (nW < w)
                  {  maskT = maskU;     // старшая 1 маски вихря
                     for (int m = 1; m < nW; m++)
                        maskT |= maskU >> m;      // маска вихря
                     xL = xB;          // заготовка начала вихря
                     xR = DeonYuli Cong(xL);      // пара xL, xR
                     nV = 0U;          // номер числа в вихре nT
                     st1 = 103U;        // генерировать вихрь nT
                  }
                  else st1 = 104U;
                  break;
               case 103U:                        // генерация вихря
                  xG = DeonYuli_TwistPair();       // вихрь пары
                  xL = xR;                // начало следующей пары
                  xR = DeonYuli_Cong(xL);         // пара xL, xR
                  if (nV == N1) st1 = 102U;
                  else nV++;         // номер следующей величины
                  FlagWhile1 = false;        // величина создана
                  break;
```

```
               case 104U:     // закончились вихри nW внутри nT
                  if (nT < N1)
                  {  nT++;     // номер nT для группы вихрей nW
                     xB = DeonYuli_Cong(xB);
                     xR = xB;
                     nW = 0U;                 // номер вихря в w
                     nV = 0U;          // номер величины в вихре
                     st1 = 101U;           // генерация вихря nT
                  }
                  else st1 = 105U;
                  break;
               case 105U:                  // изменение констант
                  stG = 2U;      // группа изменения параметров
                  st2 = 201U;           // изменение параметров
                  FlagWhile1 = false;         // выход из группы
                  FlagNext1 = true;       // переход в группу 2
                  break;
            }   // switch
         }       // while
         return FlagNext1;               // результат генерации
      }
//-------------------------------------------------------------
      bool DeonYuli_Next2()
      {  bool FlagNext2 = true;     // параметры будут изменены
         bool FlagWhile2 = true;      // переходы по состояниям
         while (FlagWhile2)
         {  switch (st2)
            {  case 201U:                // изменить параметр c
                  c += 2U;             // следующая константа c
                  if (c <= ce)
                  {  stG = 0U;     // группа начальных действий
                     st0 = 2U;    // текущие начальные действия
                     FlagWhile2 = false;    // работа завершена
                     FlagNext2 = true;    // перейти в группу 0
                  }
                  else st2 = 202U;
                  break;
               case 202U:             // заменить интервал по a
                  c = cb;               // начальное значение c
                  if (nA == 1U) nA = 2U; else nA = 1U;
                  if (nA == 1U) st2 = 203U;      // интервал a1
                  else st2 = 204U;               // интервал a2
                  break;
               case 203U:               // новое значение из a1
                  a1 -= 4U;
                  if (a1 < a1b)                  // a1 исчерпан
                  {  a1s = 2U;          // интервал a1 исчерпан
                     st2 = 205U;        // состояния интервалов
                     break;
                  }
                  a = a1;                // текущая константа a
                  c = cb;                 // начало константы c
                  a1s = 1U;              // интервал a1 активен
                  stG = 0U;         // группа начальных действий
                  st0 = 2U;        // текущие начальные действия
                  FlagWhile2 = false;        // работа завершена
                  FlagNext2 = true;       // перейти в группу 0
                  break;
               case 204U:               // новое значение из a2
                  a2 += 4U;
                  if (a2 > a2e)                   // a2 пройден
                  {  a2s = 2U;           // интервал a2 пройден
                     st2 = 205U;        // состояния интервалов
```

```
                    break;
                 }
                 a = a2;                // текущая константа a
                 c = cb;                 // начало константы c
                 a2s = 1U;              // интервал a2 активен
                 stG = 0U;         // группа начальных действий
                 st0 = 2U;         // общая начальная генерация
                 FlagWhile2 = false;       // работа завершена
                 FlagNext2 = true;       // перейти в группу 0
                 break;
              case 205U:          // один из a1 или a2 пройден
                 if (a2s != 2U) st2 = 204U;
                 else if (a1s != 2U) st2 = 203U;
                      else
                      {  stG = 0U;                 // группа 0
                         st0 = 1U;             // общее начало
                         FlagWhile2 = false;
                         FlagNext2 = true;      // перейти в 0
                      }
                 break;
           }      // switch
        }         // while
        return FlagNext2;                  // работа завершена
     }
//---------------------------------------------------------------
     uint DeonYuli_Cong(uint z)
     {  return (a * z + c) & maskW;      // следующая величина
     }
//---------------------------------------------------------------
     uint DeonYuli_TwistPair()
       {  uint g = (xR & maskT) >> (int)(w - nW);   // старшие
          return ((xL << (int)nW) & maskW) | g;     // младшие
       }
//---------------------------------------------------------------
     public void Start()
     {  N1 = 0xFFFFFFFF >> (32 - (int)w);// максимальное число
        maskW = 0xFFFFFFFF >> (32 - (int)w);    // маска числа
        maskU = 1U << ((int)w - 1);     // маска старшего бита
        maskT = maskU;                  // первый вихревой бит
        DeonYuli_SetA();     // установить границы для a1 и a2
        DeonYuli_SetC();           // установить границы для c
        x0 &= maskW;
        stG = 0U;                       // группа генерации xG
        st0 = 1U;                  // инициализация генератора
     }
//---------------------------------------------------------------
     public void TimeStart()
     {  x0 = (uint)DateTime.Now.Millisecond;   // миллисекунды
        Start();                             // старт генератора
     }
//---------------------------------------------------------------
     public void SetW(int sw)
     {  w = (uint)Math.Abs(sw);         // битовая длина числа
        if (w < 3U) w = 3U;               // минимальная длина
        else if (w > 32U) w = 32U;        // максимальная длина
        N1 = 0xFFFFFFFF >> (32 - (int)w);// максимальное число
        x0 = N1 / 7U;             // начало последовательности
     }
//---------------------------------------------------------------
     public void SetA(double sab, double sae)
     {  abf = Math.Abs(sab);
        aef = Math.Abs(sae);
        if (abf > 1.0) abf = 1.0;
```

```
         if (aef > 1.0) aef = 1.0;
         if (abf > aef) aef = abf;
      }
//---------------------------------------------------------------
      void DeonYuli_SetA()
      {  a1b = (uint)(N1 * abf);          // нижняя грань для a1
         a1b = DeonYuli_PlusA(a1b);       // начало интервала a1
         a2e = (uint)(N1 * aef);         // верхняя грань для a2
         a2e = DeonYuli_MinusA(a2e);  // окончание интервала a2
         uint 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 == 4U)                     // одноточечные a1 и a2
         {  a1e = a1b;              // a1 состоит из одной точки
            a2b = a2e;              // a2 состоит из одной точки
            return;
         }
         if (r == 8U)      // a1 имеет 2 точки, a2 - одну точку
         {  a1e = a1b + 4U;                      // окончание a1
            a2b = a2e;                              // начало a2
            return;
         }
         a1e = (a1b + a2e) / 2U;              // середина для a
         a1e = DeonYuli_MinusA(a1e);       // слева от середины
         a2b = a1e + 4U;                  // справа от середины
      }
//---------------------------------------------------------------
      uint DeonYuli_PlusA(uint a)
      {  if (a < 1U) { a = 1U; return a; }
         uint z = a;                          // нижняя граница для a
         for (uint i = 0U; i < 3U; i++)
            if (a % 4U != 0U) a--;     // условие равномерности
            else break;
         a++;                 // правильное значение константы a
         if (a < z) a += 4U;         // справа от нижней границы
         if (a >= N1 - 1) a -= 4U;  // слева от верхней границы
         return a;
      }
//---------------------------------------------------------------
      uint DeonYuli_MinusA(uint a)
      {  if (a < 1U) { a = 1U; return a; }
         uint z = a;                          // нижняя граница для a
         for (uint i = 0U; i < 3U; i++)
            if (a % 4U != 0U) a--;     // условие равномерности
            else break;
         a++;                 // правильное значение константы a
         if (a > z) a -= 4U;         // 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 = (uint)(N1 * cbf);           // нижняя грань для c
         if (cb % 2U == 0U) cb += 1;       // только нечетное c
         if (cb > N1) cb = N1;          // максимальное значение
         ce = (uint)(N1 * cef);          // верхняя грань для c
         if (ce % 2U == 0U) ce -= 1U;      // только нечетное c
         if (ce > N1 - 1) ce = N1;     // максимальное значение
         if (cb > ce) ce = cb;
         c = cb;             // начало конгруэнтной константы c
      }
//-------------------------------------------------------------
      public void SetX0(double xs)
      {  x0 = (uint)(N1 * Math.Abs(xs));
      }
//=============================================================
   }
}
```

В классе *cDeonYuliTwist32D* зарезервировано несколько переменных, которые можно настраивать с помощью инкапсулированных функций. В качестве первого примера используем значения параметров по умолчанию для генерации нескольких случайных величин длиной $w=16$ бит из диапазона $[0:2^w-1]=[0:2^{16}-1]=[0:65535]$.

В следующей программе *P030402* генерируются несколько начальных случайных величин длиной $w=32$ бита, используя технологию представленного выше класса *nsDeonYuliTwist32D* без вихревого массива:

```
using nsDeonYuliTwist32D;             // класс вихревого генератора
namespace P030402
{ class P030402
  {  static void Main(string[] args)
     {  cDeonYuliTwist32D CT = new cDeonYuliTwist32D();
        CT.SetW(32);             // длина случайных величин 32 бита
        CT.Start();                              // запуск генератора
        Console.WriteLine(
                  "CT.x0 = {0}  CT.a = {1}  CT.c = {2}",
                   CT.x0, CT.a, CT.c);
        for (int j = 0; j < 8; j++)
        {  uint z = CT.Next();               // случайная величина
           Console.WriteLine("{0,10} ", z);              // монитор
        }
        Console.ReadKey();                   // просмотр результата
     }
  }
}
```

После выполнения программы на мониторе появляется следующий результат:

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

Число $N$ целых неповторяющихся случайных величин в одной последовательности составляет

$$N = 2^w. \tag{3}$$

Полное число $N_T$ вихревых последовательностей оценивается как

$$N_T = wN = w \cdot 2^w. \tag{4}$$

Из формул (3) и (4) следует, что генератор создает $N_s$ случайных величин во всех вихревых последовательностях для каждой пары констант $a$ и $c$ (2):

$$N_s = NN_T = 2^w w \cdot 2^w = w \cdot 2^{2w}. \tag{5}$$

При $w = 32$ бита предыдущая программа *P030402* способна создать в одной последовательности $N = 2^{32} = 4\,294\,967\,296$ чисел. Для одной пары констант $a$ и $c$ (2) максимальное число создаваемых вихревых последовательностей составляет $N_T = w \cdot 2^w = 2^5 \cdot 2^{32} = 2^{37} = 137\,438\,953\,474$. Тогда общее число случайных величин составит $N_s = 32 \cdot 2^{2 \cdot 32} = 2^5 \cdot 2^{64} = 2^{69}$ чисел.

**Обсуждение.** Представленный генератор *nsDeonYuliTwist32D* позволяет настраивать генерацию при различных сочетаниях пар $a$ и $c$. Для полных последовательностей константа $a$ должна удовлетворять условию $(a-1) \bmod 4 = 0$. Поскольку $a \in [1 : N-1] = [1 : 2^w - 1]$, число $N_a$ различных значений константы определяется как

$$N_a = \frac{N}{4} = \frac{2^w}{2^2} = 2^{w-2}. \tag{6}$$

При генерации полных последовательностей значение константы $c$ должно быть нечетным. Следовательно, полное число значений $c$ составляет

$$N_c = \frac{N}{2} = \frac{2^w}{2^1} = 2^{w-1}. \tag{7}$$

Произведение формул (6) и (7) позволяет определить полное число $N_{ac}$ константных пар $a$ и $c$:

$$N_{ac} = N_a N_c = 2^{w-2} \cdot 2^{w-1} = 2^{2w-3}. \tag{8}$$

Совместное произведение чисел $N_s$ (5) во всех вихрях одной пары констант $a$ и $c$ и константных пар $N_{ac}$ (8) определяет общее число случайных величин $N_{sac}$ во всех вихревых последовательностях при заданной битовой длине $w$ каждого числа:

$$N_{sac} = w \cdot 2^{2w} \cdot 2^{2w-3} = w \cdot 2^{4w-3}. \tag{9}$$

Битовая длина $N_{bsac}$ всех чисел полной генерации вихревых последовательностей составляет (9):

$$N_{bsac} = wN_{sac} = w^2 \cdot 2^{4w-3}. \quad (10)$$

Учитывая (10), получаем, что длина информационной неповторяемости $N_I$ генерации полного равномерного вихревого цикла имеет следующую оценку:

$$N_I = 2^{w^2 \cdot 2^{4w-3}}.$$

При $w = 32$ бита неповторяемость представленного вихревого генератора $N_I = 2^{32^2 \cdot 2^{4 \cdot 32-3}}$ значительно превосходит неповторяемость генератора *MT19937*, которая составляет $2^{19937} - 1$.

Показано, что объектный вихревой генератор *nsDeonYuliTwist32D* может создавать достаточно длинные последовательности случайных величин с абсолютным уровнем равномерного распределения. Причем генератор не использует дополнительный массив для хранения вихревых промежуточных последовательностей, занимая минимально возможный объем оперативной памяти компьютера.

**Заключение.** Анализ исходного материала показывает, что алгоритмы современных вихревых генераторов обычно основаны на исходном конгруэнтном массиве ограниченного размера. Такой вид технологии использует битовый сдвиг внутри массива, создавая различные вихревые последовательности. Недостаток в том, что ограниченный размер массива может быть неприемлемым для некоторых практических применений. Чтобы увеличить длину последовательностей, необходимо увеличить число элементов в массиве с учетом доказанного в работе факта, что абсолютной равномерностью обладают только полные последовательности. Однако их длина зависит от битовой длины генерируемых случайных величин. Если битовая длина очень большая, то она ограничит имеющуюся память для размещения программы. Чтобы убрать это ограничение, предложен алгоритм, в котором вообще не применяется исходный конгруэнтно-вихревой массив. Это позволяет генерировать последовательности очень больших размеров. Ограничением остается только допустимая битовая длина чисел, обрабатываемых непосредственно командами процессора компьютера. Проведенные эксперименты подтверждают абсолютное равномерное распределение получаемых случайных величин. Автоматическая настройка конгруэнтных параметров позволяет получать управляемый уровень повторения в генерируемом равномерном распределении. Полученные результаты можно использовать в прикладных задачах, где есть равномерное распределение случайных величин.

## ЛИТЕРАТУРА

1. *Shamir A.* On the generation of cryptographically strong pseudorandom sequences // ACM TOCS. 1983. Vol. 1. Iss. 1. P. 38–44. DOI: 10.1145/357353.357357

2. *Lewko A.B., Waters B.* Efficient pseudorandom functions from the decisional linear assumption and weaker variants // Proc. 16th ACM Conf. on Computer and Communications Security. 2009. P. 112–120. DOI: 10.1145/1653662.1653677

3. *Claessen K., Palka M.H.* Splittable pseudorandom number generators using cryptographic hashing // Proc. 2013 ACM SIGPLAN Symposium on Haskell. 2013. Vol. 48. Iss. 12. P. 47–58. DOI: 10.1145/2503778.2503784

4. *Eichenauer-Herrmann J., Niederreiter H.* Digital inversive pseudorandom numbers // ACM TOMACS. 1994. Vol. 4. Iss. 4. P. 339–349. DOI: 10.1145/200883.200896

5. *Hellekalek P.* Inversive pseudorandom number generators: concepts, results and links // Proc. 27th Conf. on Winter Simulation. 1995. P. 255–262. DOI: 10.1145/224401.224612

6. *Sussman M., Crutchfield W., Papakipos M.* Pseudorandom number generation on the GPU // Proc. 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. 2006. P. 87–94. DOI: 10.1145/1283900.1283914

7. *Mandal K., Fan X., Gong G.* Design and implementation of warbler family of lightweight pseudorandom number generators for smart devices // ACM TECS. 2016. Vol. 15. Iss. 1. Art. 1. DOI: 10.1145/2808230

8. *Li M.* Generation of teletraffic of generalized Cauchy type // Phys. Scr. 2010. Vol. 81. Iss. 2. Art. 025007. DOI: 10.1088/0031-8949/81/02/025007

9. *Li M.* Record length requirement of long-range dependent teletraffic // Physica A: Statistical Mechanics and its Applications. 2017. Vol. 472. P. 164–187.
DOI: 10.1016/j.physa.2016.12.069

10. *Niederreiter H.* New methods for pseudorandom numbers and pseudorandom vector generation // Proc. 24th Conf. on Winter Simulation. 1992. P. 264–269.
DOI: 10.1145/167293.167348

11. *Meka R., Zuckerman D.* Pseudorandom generators for polynomial threshold functions // STOC'10 Proc. 42nd ACM Symposium on Theory of Computing. 2010. P. 427–436.
DOI: 10.1145/1806689.1806749

12. *Gopalan P., Meka R., Reingold O., Zuckerman D.* Pseudorandom generators for combinatorial shapes // STOC'11 Proc. 43rd Annual ACM Symposium on Theory of Computing. 2011. P. 253–262. DOI: 10.1145/1993636.1993671

13. *Leva J.L.* A fast normal random number generator // ACM TOMS. 1992. Vol. 18. Iss. 4. P. 449–453. DOI: 10.1145/138351.138364

14. *Applebaum B.* Pseudorandom generators with long stretch and low locality from random local one-way functions // STOC'12 Proc. 44th Annual ACM Symposium on Theory of Computing. 2012. P. 805–816. DOI: 10.1145/2213977.2214050

15. *White D.R., Clark J., Jacob J., Poulding S.M.* Searching for resource-efficient programs: low-power pseudorandom number generators // GECCO'08 Proc. 10th Annual Conf. on Genetic and Evolutionary Computation. 2008. P. 1775–1782. DOI: 10.1145/1389095.1389437

16. *Langdon W.B.* A fast high quality pseudorandom number generator for nVidia CUDA // GECCO'09 Proc. 11th Annual Conf. on Genetic and Evolutionary Computation. 2009. P. 2511–2514. DOI: 10.1145/1570256.1570353

17. *Real-time* label-free embolus detection using *in vivo* photoacoustic flow cytometry / M.A. Juratly, Y.A. Menyaev, M. Sarimollaoglu, E.R. Siegel, et al. // PLoS One. 2016. Vol. 11. No. 5. Art. e0156269. DOI: 10.1371/journal.pone.0156269

18. *Bioinspired* hemozoin nanocrystals as high contrast photoacoustic agents for ultrasensitive malaria diagnosis / K.A. Carey, Y.A. Menyaev, C. Cai, J.S. Stumhofer, et al. // J. Nanomed. Nanotechnol. 2016. Vol. 7. No. 3. P. 49. DOI: 10.4172/2157–7439.C1.031

19. *Photoacoustic* flow cytometry for single sickle cell detection *in vitro* and *in vivo* / C. Cai, D.A. Nedosekin, Y.A. Menyaev, M. Sarimollaoglu, et al. // Anal. Cell. Pathol. 2016. Vol. 2016. Art. 2642361. DOI: 10.1155/2016/2642361
URL: https://www.hindawi.com/journals/acp/2016/2642361

20. *Optical* clearing in photoacoustic flowcytometry / Y.A. Menyaev, D.A. Nedosekin, M. Sarimollaoglu, M.A. Juratli, et al. // Biomed. Opt. Express. 2013. Vol. 4. Iss. 12. P. 3030–3041. DOI: 10.1364/BOE.4.003030

21. *Preclinical* photoacoustic models: application for ultrasensitive single cell malaria diagnosis in large vein and artery / Y.A. Menyaev, K.A. Carey, D.A. Nedosekin, M. Sarimollaoglu, et al. // Biomed. Opt. Express. 2016. Vol. 7. Iss. 9. P. 3643–3658. DOI: 10.1364/BOE.7.003643

22. *Matsumoto M., Nishimura T.* Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator // ACM TOMACS. 1998. Vol. 8. Iss. 1. P. 3–30.
DOI: 10.1145/272991.272995

23. *Matsumoto M., Saito M., Haramoto H., Nishimura T.* Pseudorandom number generation: impossibility and compromise // J. Univers. Comput. Sci. 2006. Vol. 12. Iss. 6. P. 672–690. DOI: 10.3217/jucs-012-06-0672

24. *Matsumoto M., Wada I., Kuramoto A., Ashihara H.* Common defects in initialization of pseudorandom number generators // ACM TOMACS. 2007. Vol. 17. Iss. 4. Art. 15.
DOI: 10.1145/1276927.1276928

25. *Bos J.W., Kleinjung T., Lenstra A.K., Montgomery P.L.* Efficient SIMD arithmetic modulo a Mersenne number // Proc. 2011 IEEE 20th Symposium on Computer Arithmetic. 2011. P. 213–221. DOI: 10.1109/ARITH.2011.37

26. *Deon A., Menyaev Y.* Parametrical tuning of twisting generators // J. Comput. Sci. 2016. Vol. 12. Iss. 8. P. 363–378. DOI: 10.3844/jcssp.2016.363.378

27. *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

28. *Saito M., Matsumoto M.* SIMD-oriented fast Mersenne twister: a 128-bit pseudoran-dom number generator / A. Keller, S. Heinrich, H. Niederreiter, eds. Monte Carlo and quasi-Monte Carlo methods 2006. Springer, 2008. P. 607–622.

29. *Deon A., Menyaev Y.* The complete set simulation of stochastic sequences without repeated and skipped elements // J. Univers. Comput. Sci. 2016. Vol. 22. Iss. 8. P. 1023–1047.

30. *Деон А.Ф., Меняев Ю.А.* Полное факториальное моделирование равномерных последовательностей целых случайных величин // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2017. № 5. С. 132–149. DOI: 10.18698/0236-3933-2017-5-132-149

**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, стр. 1).

**Меняев Юлиан Алексеевич** — канд. техн. наук, сотрудник Института исследования рака им. Уинтропа Рокфеллера (США, AR 72202, Little Rock, 4018 W Capitol Ave).

**Просьба ссылаться на эту статью следующим образом:**

Деон А.Ф., Меняев Ю.А. Генератор равномерных вихревых последовательностей целых случайных величин без запоминающего массива // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2018. № 3. С. 51–69. DOI: 10.18698/0236-3933-2018-3-51-69

# GENERATOR OF UNIFORM TWISTER SEQUENCES OF RANDOM INTEGER NUMBERS WITHOUT STORAGE ARRAYS

**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, Little Rock, USA**

**Abstract**

Mersenne Twister random integer number generators may use arrays to implement twister shift operations internally. This technology provides excellent results for random number arrays in short and medium sequences. However, today's generators tend towards fairly long sequences, arrays of which may overflow the computer's random-access memory. These options make the older array technology unusable, since then there is no place anymore for applications and the operating system. We propose a new technology of generating twister sequences without using congruent twister arrays. It makes it possible to create uniform random integer sequences of arbitrary length that do not use up random access memory. Simulation results confirm that the random numbers obtained feature a perfectly uniform distribution over a range of unique sequences. Moreover, combining this new approach with the twister generation adjustment algorithm allows for significant increases in the length of sequences created, not involving any extra random-access memory

**Keywords**

*Computer simulation, random number generators, stochastic sequence algorithms*

Received 05.02.2018


## REFERENCES

[1] Shamir A. On the generation of cryptographically strong pseudorandom sequences. *ACM TOCS*, 1983, vol. 1, iss. 1, pp. 38–44. DOI: 10.1145/357353.357357

[2] Lewko A.B., Waters B. Efficient pseudorandom functions from the decisional linear assumption and weaker variants. *Proc. 16th ACM Conf. on Computer and Communications Security*, 2009, pp. 112–120. DOI: 10.1145/1653662.1653677

[3] Claessen K., Palka M.H. Splittable pseudorandom number generators using cryptographic hashing. *Proc. 2013 ACM SIGPLAN Symposium on Haskell*, 2013, vol. 48, iss. 12, pp. 47–58. DOI: 10.1145/2503778.2503784

[4] Eichenauer-Herrmann J., Niederreiter H. Digital inversive pseudorandom numbers. *ACM TOMACS*, 1994, vol. 4, iss. 4, pp. 339–349. DOI: 10.1145/200883.200896

[5] Hellekalek P. Inversive pseudorandom number generators: concepts, results and links. *Proc. 27th Conf. on Winter Simulation*, 1995, pp. 255–262. DOI: 10.1145/224401.224612

[6] Sussman M., Crutchfield W., Papakipos M. Pseudorandom number generation on the GPU. *Proc. 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, 2006, pp. 87–94. DOI: 10.1145/1283900.1283914

[7] Mandal K., Fan X., Gong G. Design and implementation of warbler family of lightweight pseudorandom number generators for smart devices. *ACM TECS*, 2016, vol. 15, iss. 1, art. 1. DOI: 10.1145/2808230

[8] Li M. Generation of teletraffic of generalized Cauchy type. *Phys. Scr.*, 2010, vol. 81, iss. 2, art. 025007. DOI: 10.1088/0031-8949/81/02/025007

[9] Li M. Record length requirement of long-range dependent teletraffic. *Physica A: Statistical Mechanics and its Applications*, 2017, vol. 472, pp. 164–187. DOI: 10.1016/j.physa.2016.12.069

[10] Niederreiter H. New methods for pseudorandom numbers and pseudorandom vector generation. *Proc. 24th Conf. on Winter Simulation*, 1992, pp. 264–269.
DOI: 10.1145/167293.167348

[11] Meka R., Zuckerman D. Pseudorandom generators for polynomial threshold functions. *STOC'10 Proc. 42nd ACM Symposium on Theory of Computing*, 2010, pp. 427–436.
DOI: 10.1145/1806689.1806749

[12] Gopalan P., Meka R., Reingold O., Zuckerman D. Pseudorandom generators for combinatorial shapes. *STOC'11 Proc. 43rd Annual ACM Symposium on Theory of Computing*, 2011, pp. 253–262. DOI: 10.1145/1993636.1993671

[13] Leva J.L. A fast normal random number generator. *ACM TOMS*, 1992, vol. 18, iss. 4, pp. 449–453. DOI: 10.1145/138351.138364

[14] Applebaum B. Pseudorandom generators with long stretch and low locality from random local one-way functions. *SNOC'12 Proc. 44th Annual ACM Symposium on Theory of Computing*, 2012, pp. 805–816. DOI: 10.1145/2213977.2214050

[15] White D.R., Clark J., Jacob J., Poulding S.M. Searching for resource-efficient programs: low-power pseudorandom number generators. *GECCO'08 Proc. 10th Annual Conf. on Genetic and Evolutionary Computation*, 2008, pp. 1775–1782. DOI: 10.1145/1389095.1389437

[16] Langdon W.B. A fast high quality pseudorandom number generator for nVidia CUDA. *GECCO'09 Proc. 11th Annual Conf. on Genetic and Evolutionary Computation*, 2009, pp. 2511–2514. DOI: 10.1145/1570256.1570353

[17] Juratli M.A., Menyaev Y.A., Sarimollaoglu M., Siegel E.R., Nedosekin D.A., Suen J.Y., Melerzanov A.V., Juratli T.A., Galanzha E.I., Zharov V.P. Real-time label-free embolus detection using *in vivo* photoacoustic flow cytometry. *PLoS One*, vol. 11, no. 5, art. e0156269.
DOI: 10.1371/journal.pone.0156269

[18] Juratli M.A., Menyaev Y.A., Sarimollaoglu M., Siegel E.R., Nedosekin D.A., Suen J.Y., Melerzanov A.M., Juratli T.A., Galanzha E.I., Zharov V.P. Bioinspired hemozoin nanocrystals as high contrast photoacoustic agents for ultrasensitive malaria diagnosis. *J. Nanomed. Nanotechnol.*, 2016, vol. 7, no. 3, pp. 49. DOI: 10.4172/2157–7439.C1.031

[19] Cai C., Nedosekin D.A., Menyaev Y.A., Sarimollaoglu M., Proskurnin M.A., Zharov V.P. Photoacoustic flow cytometry for single sickle cell detection *in vitro* and *in vivo*. *Anal. Cell. Pathol.*, 2016, vol. 2016, art. 2642361. DOI: 10.1155/2016/2642361
Available at: https://www.hindawi.com/journals/acp/2016/2642361

[20] Menyaev Y.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, iss. 12, pp. 3030–3041. DOI: 10.1364/BOE.4.003030

[21] Menyaev Y.A., Carey K.A., Nedosekin D.A., Sarimollaoglu M., Galanzha E.I., Stumhofer J.S., Zharov V.P. Preclinical photoacoustic models: application for ultrasensitive single cell malaria diagnosis in large vein and artery. *Biomed. Opt. Express*, 2016, vol. 7, iss. 9, pp. 3643–3658. DOI: 10.1364/BOE.7.003643

[22] Matsumoto M., Nishimura T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM TOMACS*, 1998, vol. 8, iss. 1, pp. 3–30.
DOI: 10.1145/272991.272995

[23] Matsumoto M., Saito M., Haramoto H., Nishimura T. Pseudorandom number generation: impossibility and compromise. *J. Univers. Comput. Sci.*, 2006, vol. 12, iss. 6, pp. 672–690. DOI: 10.3217/jucs-012-06-0672

[24] Matsumoto M., Wada I., Kuramoto A., Ashihara H. Common defects in initialization of pseudorandom number generators. *ACM TOMACS*, 2007, vol. 17, iss. 4, art. 15. DOI: 10.1145/1276927.1276928

[25] Bos J.W., Kleinjung T., Lenstra A.K., Montgomery P.L. Efficient SIMD arithmetic modulo a Mersenne number. *Proc. 2011 IEEE 20th Symposium on Computer Arithmetic*, 2011, pp. 213–221. DOI: 10.1109/ARITH.2011.37

[26] Deon A., Menyaev Y. Parametrical tuning of twisting generators. *J. Comput. Sci.*, 2016, vol. 12, iss. 8, pp. 363–378. DOI: 10.3844/jcssp.2016.363.378

[27] Deon A.F., Menyaev Yu.A. Uniform random quantity generator using complete vortex array technology. *Vestn. Mosk. Gos. Tekh. Univ. im. N.E. Baumana, Priborostr.* [Herald of the Bauman Moscow State Tech. Univ., Instrum. Eng.], no. 2, pp. 86–110 (in Russ.). DOI: 10.18698/0236-3933-2017-2-86-110

[28] Saito M., Matsumoto M. SIMD-oriented fast Mersenne twister: a 128-bit pseudorandom number generator. A. Keller, S. Heinrich, H. Niederreiter, eds. In: Monte Carlo and quasi-Monte Carlo methods 2006. Springer, 2008, pp. 607–622.

[29] Deon A., Menyaev Y. The complete set simulation of stochastic sequences without repeated and skipped elements. *J. Univers. Comput. Sci.*, 2016, vol. 22, iss. 8, pp. 1023–1047.

[30] Deon A.F., Menyaev Yu.A. Complete factorial simulation of integer random number uniform sequences. *Vestn. Mosk. Gos. Tekh. Univ. im. N.E. Baumana, Priborostr.* [Herald of the Bauman Moscow State Tech. Univ., Instrum. Eng.], 2017, no. 5, pp. 132–149 (in Russ.). DOI: 10.18698/0236-3933-2017-5-132-149

**Deon A.F.** — Cand. Sc. (Eng.), Assoc. Professor, Department of Computer Software and Information Technology, Bauman Moscow State Technical University (2-ya Baumanskaya ul. 5, str. 1, Moscow, 105005 Russian Federation).

**Menyaev Yu.A.** — Cand. Sc. (Eng.), works in Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Science (4018 W Capitol Ave, Little Rock, AR 72202 USA).

**Please cite this article in English as:**

Deon A.F., Menyaev Yu.A. Generator of Uniform Twister Sequences of Random Integer Numbers without Storage Arrays. *Vestn. Mosk. Gos. Tekh. Univ. im. N.E. Baumana, Priborostr.* [Herald of the Bauman Moscow State Tech. Univ., Instrum. Eng.], 2018, no. 3, pp. 51–69 (in Russ.). DOI: 10.18698/0236-3933-2018-3-51-69