УДК 681.3.06 DOI: 10.18698/0236-3933-2017-5-132-149

ПОЛНОЕ ФАКТОРИАЛЬНОЕ МОДЕЛИРОВАНИЕ РАВНОМЕРНЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ ЦЕЛЫХ СЛУЧАЙНЫХ ВЕЛИЧИН

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, Литл Рок, Соединенные Штаты Америки

Аннотация

Случайные последовательности широко используют в теоретических и практических областях человеческой и технической деятельности. Важная часть этих исследований относится к процедурам создания случайных величин. Одно направление относится к последовательной генерации псевдослучайных величин, а другое — использует полное множество всех стохастических последовательностей. Первое направление хорошо изучено и традиционно применяется, начиная от криптографии и технических систем и заканчивая биологическими и медицинскими исследованиями. Второе направление в основном используется во всеобъемлющих тестированиях. В настоящей работе исследовано второе направление, где требуются все последовательности заданного диапазона. В некоторых современных генераторах наблюдаются пропуски и повторения случайных величин. В связи с этим предложены ограничения, следуя которым указанные недостатки можно исключить, а также новые алгоритмы на основе факториального анализа, которые допускают быструю генерацию всех последовательностей без пропусков и повторений случайных величин. Рассмотрены достоинства и недостатки полученных результатов

Ключевые слова

Компьютерное моделирование, генераторы случайных величин, алгоритмы стохастических последовательностей

Поступила в редакцию 29.06.2017

Введение. Генераторы равномерных случайных величин (*Uniform Random Number Generator — URNG*) широко применяют в математических исследованиях [1–5], в криптографии [6, 7], в тестировании технических систем [8, 9], а также в других прикладных областях [10], включая модели исследований биологии [11, 12] и медицины [13, 14]. Особое место занимают современные вихревые равномерные генераторы [15–19]. Кроме того, сами *URNG* часто являются первичными элементами для создания других генераторов, как это сделано в нормальном генераторе Бокса — Мюллера [20]. В нем использованы одновременно два генератора. Пример такого фрагмента выглядит следующим образом:

```
        if (!generate)
           return z1 * sigma + mu;
        double u1, u2;
        do
         {
           u1 = rand() * (1.0 / RAND_MAX);
           u2 = rand() * (1.0 / RAND_MAX);
         }
        while ( u1 <= epsilon );
        z0 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
        z1 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2);
```

Здесь значения `z0` и `z1` являются нормальными случайными величинами, которые получены с помощью равномерных случайных величин `u1` и `u2` из полуоткрытого интервала (0, 1]. В качестве генератора равномерных случайных величин может выступать стандартная функция `rand()` среды *Microsoft Visual Studio* различных выпусков. Непосредственная проверка равномерности функции `rand()` показывает, что получаемые случайные величины `u1` и `u2` не обладают высоким качеством, допуская пропуск и повторяемость генерируемых величин. Программный код *H0101* на историческом языке программирования C, в котором моделируются 32 767 целых случайных величин, представлен ниже. Генератор `rand()` создает целые случайные величины из интервала $\left[\overline{0:32767}\right]$. Выполним 32 767 генераций и посмотрим, сколько раз будет создана любая случайная величина:

```
// H0101   C
// Тестирование rand()
#include <conio.h>                                          // _getch
#include <iostream>
using namespace std;                                          // cout
#include <iomanip>                                            // setw
#include <stdlib.h>                                     // srand, rand
#include <time.h>                                             // time
void main()
{    int w = 15;              // битовая длина случайной величины
      int N1 = (int)(0xFFFFFFFF >> (32 - w));          // максимум
   cout << "w = " << w << "   N1 = " << N1 << endl;
   int* c = new int[N1 + 1];                // массив счетчиков
   int n1 = 0;               // нижняя грань случайных величин
   int n2 = N1;             // верхняя грань случайных величин
   cout << "n1 = " << n1 << "   n2 = " << n2 << endl;
   for (int j = 0; j <= N1; j++) c[j] = 0;

   srand((unsigned int)time(NULL));          // начальное число
   for (int k = 0; k <= N1; k++)
   {
      int v = rand();
      if (n1 <= v && v < n2) c[v - n1]++;
   }
   int q0 = 0, q1 = 0, q2 = 0, q3 = 0;
   for (int j = 0; j <= N1; j++)
```

```
    {
        if (c[j] == 0) q0++;
        else if (c[j] == 1) q1++;
        else if (c[j] == 2) q2++;
        else q3++;
    }
    cout << "q0 = " << q0 << endl;
    cout << "q1 = " << q1 << endl;
    cout << "q2 = " << q2 << endl;
    cout << "q3 = " << q3 << endl;
    _getch();                                  // просмотр результата
}
```

После запуска этой программы на мониторе появляется следующий листинг:

```
w = 15  N1 = 32767
n1 = 0  n2 = 32767
q0 = 11996
q1 = 12162
q2 = 5977
q3 = 2633
```

Полученный результат показывает, что было пропущено `q0 = 11996` случайных величин. Только `q1 = 12162` случайные величины были созданы один раз и действительно обладают свойством равномерности на интервале $\left[\overline{0:32767}\right]$. Дважды были сгенерированы `q2 = 5977` случайные величины, нарушающие условие равномерности. Остальные `q3 = 2633` случайные величины могли быть созданы три и более раз. Это и есть результат низкого качества: что-то есть, но мало.

Для создания случайной сетки на плоскости, должны выполнять два декартовых условия равномерности:

1) сетка не должна иметь пропущенные узлы-вершины;

2) сетка не должна иметь повторяющиеся вершины.

Далее будем называть плоскость равномерно случайной, если на ней существует декартовая сетка случайных узлов-вершин.

Предыдущее тестирование *H0101* функции `rand()` показывает, что созданная на ее основе случайная плоскость обладает низким качеством, поскольку будет иметь пропущенные и повторяющиеся случайные величины. Из этого также будет следовать, что представленный выше программный код из генератора нормальных случайных величин методом Бокса — Мюллера не сможет обеспечить высокое качество последовательностей нормальных случайных величин. Причина — низкое качество функции `rand()`.

Возникает вопрос, какими должны быть равномерные случайные последовательности, чтобы обеспечить существование абсолютно всех равномерных последовательностей случайных величин длиной w бит? Доказательство существования полного множества таких последовательностей будет убедительным, если можно представить абсолютно все последовательности равномерных случайных величин длиной w бит.

Цель настоящей работы — моделирование полного множества абсолютно всех равномерных случайных последовательностей без повторений и пропусков исходных элементов наблюдений.

Теория. Компьютерные случайные величины характеризуются конечным числом w информативных битов, в которых можно наблюдать не более чем $N = 2^w$ случайных величин. Все они образуют множество с полным набором случайных величин $B = \{b_1, b_2, \ldots, b_{N=2^w}\}$. Множество B позволяет индуцировать множество конечных последовательностей D, составленных из случайных величин $b \in B$. Ограничим множество D только такими последовательностями, в которых находятся ровно $N = 2^w$ случайных величин. Если в некоторых последовательностях будут наблюдаться повторения случайных величин, то в таких последовательностях в то же время будут отсутствовать некоторые элементы $b \in B$, поскольку введенное условие ограничивает в ней число случайных величин как $N = 2^w$ элементов. Введем второе ограничение множества D, которое предполагает, что рассматриваются только такие последовательности $d \in D$, в которых отсутствуют пропуски или повторения случайных величин $b \in B$. Эти два ограничения позволяют утверждать, что такое множество D с указанными свойствами будет минимальным по числу различимых случайных последовательностей, т. е. D — множество всех последовательностей с неповторяющимися случайными величинами. Тогда каждая последовательность $d \in D$ содержит все элементы $b \in B$. Математически D — это множество всех перестановок объектов из B. Оно является минимальным над полем B.

Итак, каждый элемент $d \in D$ содержит ровно $N = 2^w$ неповторяющихся элементов $b \in B$. В этих обозначениях имеем полное множество всех последовательностей длиною $N = 2^w$, но при условии, что в каждой последовательности все случайные величины указаны по одному разу, т. е. простейшая полнота по объектам наблюдений $b \in B$ и простейшая полнота по последовательностям $d \in D$.

Здесь вопрос неопределенности выборки в наблюдениях можно рассматривать в двух аспектах:

1) неопределенность наблюдения объекта $b \in B$ в определенном месте последовательности, но объект обязательно проявляет себя явно;

2) неопределенность наблюдения последовательности, хотя каждая последовательность $d \in D$ обязательно проявляет себя, поскольку других последовательностей с неповторяющимися элементами $b \in B$ не может быть.

Мощность функционала F наблюдений связана с количеством выборок последовательностей $d \in D$ следующим образом:

$$card(F) = card(D) = 2^w !.$$

Это непосредственно следует из комбинаторного математического анализа [21], если рассматривать оценку количества перестановок индексов для N элементов.

В теории вероятностей [22] такой же результат получается для выборок без повторений: на первом месте в последовательности может находиться любой из

элементов $b_1 \in B$ — это $N = 2^w$ вариантов; на втором месте может находиться любой из меньшего множества ровно на один элемент $b_2 \in \left(B \setminus \{b_1\}\right)$ — $2^w - 1$ вариантов и т. д. Перемножая количество всех вариантов, получаем $2^w!$.

В результате приходим к заключению, что одной из характеристик неопределенности функционала выборки конкретной последовательности является факториальная полнота числа имеющихся последовательностей.

С одной стороны, при введении в рассмотрение функционала выборки, не было использовано понятие математического определения функции *f*, которая каждому элементу или набору элементов *r* ставит в соответствие единственный зависимый элемент $f(r)$. С другой стороны, каждая выборка при реализации или разрешении функционала ставит в соответствие точно одну последовательность d_r из полного множества всех допустимых последовательностей:

$$f_r = d_r \in D.$$

В таком функциональном обозначении *r* характеризует последовательность некоторых действий, выполнение которых позволяет получить точно одну последовательность из *D*. В свою очередь, мощность $|D|$ характеризует неопределенность до начала выборки, а удовлетворение *r* завершает выборку указанием конкретной последовательности d_r. Набор всех однозначных функций $f(r)$, где r указывает способ получения соответствующей последовательности, составляет функционал *F* на полном множестве *D*:

$$F = \{f_r \to d_r \in D\}.$$

Поскольку число последовательностей $card(D)$ совпадает с мощностью $|D|$, то мощность функционала $|F|$ совпадает с мощностью $|D|$:

$$|F| = |D| = N! = 2^w!.$$

Это означает, что диапазон индекса *r* определяет интервал $r \in \left[\overline{1, n!}\right]$. Необходимо ответить на вопрос, какому номеру $r \in \left[\overline{1, n!}\right]$ соответствует некоторая последовательность $d_r \in D$? Вернемся позднее к этому вопросу. Соберем вместе введенные обозначения в единое понятие модели, с помощью которой далее будут представлены соответствующие алгоритмы факториального моделирования.

Назовем моделью *M* полного множества *D* конечных последовательностей с неповторяющимися элементами $b \in B$ тройку множеств, в которой возможна реализация разрешения неопределенности для функционала *F*: $M = (B, D, F)$.

Множество случайных величин $b \in B$ задается априори согласно выбранному плану изучения явлений. Множество последовательностей *D* является минимальным множеством по числу простейших конечных последовательностей без пропусков и повторений всех случайных величин. Каждая последовательность из

D содержит все множество B с уникальным перечислением элементов. Множество функционала F включает в себя все функции, способные разрешить неопределенность в выборке каждый раз одной единственной последовательности.

Вернемся к вопросу, как реализовать или разрешить функционал F выборки d_r. Обратимся к истокам определения математического множества. Множество можно задать только одним из двух способов:

1) явно перечислить все элементы множества;

2) указать порядок действий, позволяющий предъявить любой элемент множества. Если порядок действий не полный, т. е. нельзя предъявить все элементы множества, то такой функционал действий является неполным, что равносильно свойству неполноты.

Представленные особенности формирования любого множества позволяют указать первый способ реализации функционала в модели M, а именно, перечислять последовательности $d \in D$ до тех пор, пока не будет обнаружена последовательность, удовлетворяющая критерию r: $d = d_r$. Однозначность такого способа явна, поскольку всегда, по построению D, найдется одна и только одна последовательность, обладающая заданной перестановкой r объектов из B. Перестановка r единственна, что гарантируется математической комбинаторикой.

Второй способ разрешения функционала F основан на анализе неопределенности для модели $M = (B, D, F)$. Суть в том, что число элементов-последовательностей в множестве D составляет $card(D) = |D| = n!$. Пусть r является некоторым числом на арифметическом интервале целых чисел $\left[\overline{1, n!}\right] = [1, 2, 3, \ldots, n!]$. Отметим, что элементы $r \in \left[\overline{1, n!}\right]$ строго упорядочены — это и есть ключ к разрешению поставленной задачи. Следовательно, необходимо добиться упорядоченности номеров последовательностей из D. Для этого используем арифметико-алгебраическое понятие позиционного представления целого числа X, содержащего n цифр x_i в некоторой системе счисления с основанием s:

$$X = x_{n-1}s^{n-1} + x_{n-2}s^{n-2} + \ldots + x_1 s^1 + x_0 s^0.$$

Теперь используем комбинаторное определение последовательности в множестве D — перестановка неповторяющихся индексов-чисел. Без потери общности пусть случайные величины $b \in B$ пронумерованы или обозначены целыми числами из арифметического интервала $\left[\overline{1, N}\right]$. Тогда первой или младшей последовательностью будет последовательность, соответствующая минимальному позиционному числу, в котором цифры взяты из позиционного представления индексов.

Пример. Пусть возможны три случайные величины $(n = 3)$, которые обозначим в множестве B целыми числами 1, 2, 3. Первой младшей последовательностью в D будет последовательность 1, 2, 3, которой соответствует целое позиционное число $X_1 = 123$. Второй последовательностью будет последователь-

ность с перестановкой 1, 3, 2. Ей соответствует число $X_2 = 132$. Сразу видна явная упорядоченность $123 < 132$, или $X_1 < X_2$. Остальные последовательности располагаются в упорядоченном множестве *D* следующим образом:

Последовательность	*X*	*r*
1, 2, 3	123	1
1, 3, 2	132	2
2, 1, 3	213	3
2, 3, 1	231	4
3, 1, 2	312	5
3, 2, 1	321	6

Всего $n! = 3! = 6$ последовательностей, которым однозначно поставлены номера *r* функций $f_1, \ldots, f_6$ из функционала *F*. Если неопределенность природы наблюдений предпочла функцию f_4, то в реальности наблюдалась последовательность 2, 3, 1.

Связь между переменной *r* и соответствующей последовательностью вычисляется следующим образом. Запишем в явном виде выражение факториала

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1.$$

В этой формуле столько сомножителей, сколько объектов в последовательности длиною *n*. Следует отметить, что коммутативное свойство факториала позволяет расположить на первом месте слева любое из *n* чисел. Чтобы определить старшую цифру слева, необходимо исключить правую часть факториала, используя его свойство $n! = n(n-1)!$. Поскольку $r_{\max} = n!$, циклически гарантирован поиск всех остальных чисел, если понижать факториал и исключать в силу коммутативности из рассмотрения те индексы, которые были определены на предыдущих итерациях. Коммутативность факториала обеспечивает однозначность алгоритма. Это является следствием утверждения, которое формально задает предлагаемая теорема.

Теорема. *Для того чтобы существовало биективное отношение между полным множеством конечных последовательностей D и функционалом выбора F в модели* $M = (B, D, F)$ *необходимо и достаточно, чтобы множество случайных величин* $b \in B$, *входящее полностью в каждую последовательность, было строго упорядочено.*

◄ Пусть существует отношение строгого порядка между элементами объектов, входящих во все последовательности полного множества. Сопоставим такому множеству арифметическое множество B^*, состоящее из чисел-объектов. Минимальным таким множеством может быть любой арифметический интервал, начиная с некоторого начального числа α. Не нарушая общности, примем $\alpha = 1$. Тогда минимальным арифметическим интервалом будет интервал $\left[\overline{1, n!}\right]$. Ясно, что в силу строгого порядка каждому элементу $b^* \in B^*$ строго соответствует один и только один элемент $b \in B$, т. е. B^* и *B* изоморфны.

Необходимость. Доказательство необходимости заключается в том, что следует указать последовательность действий, позволяющих вычислить номер r для любой случайной последовательности $d_r \in D$. Воспользуемся верхними индексами для обозначения произвольной последовательности $d = < d^1, d^2, \ldots, d^z, \ldots, d^{n-1}, d^n >$, явно подчеркивая, что индекс $z \in \left[\overline{1,n}\right]$. Всего в множестве D находятся $n!$ последовательностей. Разобьем D на n непересекающихся подмножеств $D_1 + D_2 + \ldots + D_n$ по условию упорядоченности. Отметим свойство, что в подмножество D_1, в силу упорядоченности D, войдут все последовательности, начиная с числа 1, в подмножество D_2 — все последовательности, начиная с 2 и т. д. Размер каждого подмножества, по свойству факториала $n! = n(n-1)!$, одинаковый

$$card(D_1) = card(D_2) = \ldots = card(D_n) = (n-1)!.$$

Представим результирующий номер r последовательности d как частичную сумму

$$r = r^1 + r^2 + \ldots + r^n.$$

В зависимости от значения $d^1 \in D_z$ вычислим

$$r^1 = \sum_{i=1}^{z-1} card(D_i) = (z-1)(n-1)!.$$

Перебирая итерационно числа $d^2, \ldots, d^{n-1}$ в заданной последовательности d, получаем соответствующие частичные значения $r^2, r^3, \ldots, r^{n-1}$. Последнее значение $r^n = 1$. Итоговый номер r определяется вычисленной суммой

$$r = \sum_{i=1}^{n} r^i.$$

Необходимость доказана.

Достаточность. Пусть задан номер $r \in \left[\overline{1,n!}\right]$, используя который функция $f_r \in F$ позволяет получить соответствующую последовательность $d_r \in D$. По прежнему будем полагать, что получаемая последовательность будет состоять из элементов $d_r = < d_r^1, d_r^2, \ldots, d_r^{n-1}, d_r^n >$. Разобьем D на n непересекающихся подмножеств $D_1 + D_2 + \ldots + D_n$ по условию упорядоченности. Опять используем свойство, что в подмножество D_1, в силу упорядоченности D, войдут все последовательности, начиная с числа 1, в подмножество D_2 — последовательности, начиная с 2 и т. д. Размер каждой группы, по свойству факториала $n! = n(n-1)!$, одинаковый

$$ng = card(D_1) = card(D_2) = \ldots = card(D_n) = (n-1)!.$$

Это позволяет определить номер m предыдущей группы, относительно элемента d_r^1 как результат целочисленного деления на число элементов ng в подмножестве факториального представления:

$$m=\begin{cases}\dfrac{r}{ng}, & \text{если остаток больше нуля;}\\ \dfrac{r}{ng}+1, & \text{если остаток равен нулю.}\end{cases}$$

Позиция m позволяет выбрать из множества объектов b соответствующий элемент $d_r^1 = b_m \in B$. Для следующей итерации относительно d_r^2 номер r следует уменьшить на величину $(m-1)ng$, а из множества B — удалить уже выбранный элемент b_m. Следующие итерации позволяют получить элементы $d_r^2, \ldots, d_r^{n-1}$. После итераций в множестве B останется один элемент, который будет последним элементом d_r^n в вычисляемой последовательности d_r. Достаточность доказана по построению. ►

Любая последовательность из модели M может быть задана или предъявлена случайным образом, если случай предоставит индекс r для функционала F. Авторы настоящей работы точно уверены, что предъявленная последовательность обязательно содержит все случайные величины длиной w без всяких пропусков и повторений.

Конструкция и результаты. Не нарушая общности получаемых результатов, определяем множество первичных объектов как совокупность целых чисел $B = \overline{[1, n]}$. Согласно теореме об упорядоченной биекции, в модели $M = (B, D, F)$ множество упорядоченных последовательностей D начинается с последовательности $1, 2, \ldots, n$. В качестве примера ниже приведена программа *P010301* на языке программирования C#, в которой моделируются все последовательности $d \in D$ полной перестановки элементов множества $B = \overline{[1, 7]}$. Последовательность $d_1 = 1, 2, 3, 4, 5, 6, 7$ является начальным элементом, за которым следует последовательность $d_2 = 1, 2, 3, 4, 5, 7, 6$ и т. д. Полное множество D всего содержит $N_s(D_{n=7}) = n! = 7! = 5040$ последовательностей. Последовательность $d_{5040} = 7, 6, 5, 4, 3, 2, 1$ — последняя.

Пример программы *P010301* на языке программирования C#

```
namespace P010301
{  class P010301
   {  static void Main(string[] args)
      {  int[] q = new int[] { 1, 2, 3, 4 }; //, 5, 6, 7 };
         int n = q.Length;          // длина последовательности
         int r = 0;                 // номер последовательности
         while (true)
         {  Console.Write("r = {0,4}", ++r);
            foreach (int w in q) Console.Write("{0,4}", w);
            Console.WriteLine();                // новая строка
            if (ProcessUp(q, n) == 0) break;         // процесс
         }
         Console.ReadKey();              // просмотр результата
      }
```

```
//-------------------------------------------------------------
      static public int FreeValue(int[] q, int n, int z)
      {  for (int k = 1; k <= n; k++)
         {  int j = 0;
            for (; j <= z; j++)
               if (q[j] == k) break;
            if (j > z) return k;
         }
            return 0;
      }
//-------------------------------------------------------------
      static public int NextFreeValue(int[] q, int n, int z)
      {  int v = q[z] + 1;                // возможное значение
         if (v > n) return 0;          // увеличение невозможно
         for (; v <= n; v++)      // область возможных значений
         {  int j = 0;                          // начало проверки
            for (; j < z; j++)         // элементы до z-позиции
               if (q[j] == v) break;     // значение не принято
            if (j >= z) break;      // ранее значения v не было
         }
         if (v > n) v = 0;
         return v;              // следующее свободное значение
      }
//-------------------------------------------------------------
      static public int ProcessUp(int[] q, int n)
      {  int z = n - 2;              // предпоследняя z-позиция
         while (true)
         {  int nv = NextFreeValue(q, n, z);
            if (nv != 0)       // смещение z влево не требуется
            {  q[z] = nv;          // следующее свободное число
               while (++z < n)             // z-позиции до конца
                  q[z] = FreeValue(q, n, z - 1);
               return z;
            }
            if (z == 0) break;        // последовательности нет
            z--;                               // z-позиция слева
         }
         return 0;          // следующей последовательности нет
      }
   }
}
```

Функция `NextFreeValue()` определяет следующее свободное число для позиции `z`. Это число должно быть минимальным и не совпадать ни с одним числом, находящимся в позициях до позиции `z`. Если такое число найти нельзя, то формируется отказ. Функция `FreeValue()` определяет минимальное число, которое должно находиться справа от позиции `z`. Функция `ProcessUP()` последовательно манипулирует функциями `NextFreeValue()` и `FreeValue()`, получая в итоге новую последовательность, которая непосредственно должна следовать за исходной последовательностью.

После выполнения программы *P010301* на мониторе появляется следующий листинг, который здесь приведен с сокращениями (на месте опущенных строк стоит прочерк):

```
r =    1      q =   1   2   3   4   5   6   7
r =    2      q =   1   2   3   4   5   7   6
```

```
r =    3      q =   1   2   3   4   6   5   7
   -   - - - - -
r =  100      q =   1   2   7   3   5   6   4
r =  101      q =   1   2   7   3   6   4   5
r =  102      q =   1   2   7   3   6   5   4
   -   - - - - -
r = 1000      q =   2   4   3   6   5   7   1
r = 1001      q =   2   4   3   6   7   1   5
r = 1002      q =   2   4   3   6   7   5   1
   -   - - - - -
r = 2200      q =   4   1   3   6   5   7   2
r = 2201      q =   4   1   3   6   7   2   5
r = 2202      q =   4   1   3   6   7   5   2
   -   - - - - -
r = 3300      q =   5   4   3   2   7   6   1
r = 3301      q =   5   4   3   6   1   2   7
r = 3302      q =   5   4   3   6   1   7   2
   -   - - - - -
r = 5038      q =   7   6   5   4   2   3   1
r = 5039      q =   7   6   5   4   3   1   2
r = 5040      q =   7   6   5   4   3   2   1
```

Смоделированные последовательности строго упорядочены по номерам *r*, что подтверждает абсолютную полноту факториального моделирования равномерных последовательностей случайных величин.

Факториальные алгоритмы. Доказательство теоремы об упорядоченной биекции предлагает алгоритмы необходимости и достаточности взаимного соответствия между равномерными последовательностями и их номерами *r* в предыдущей моделирующей программе *P010301*. Ниже в программе *P010302* представлена функция `DeonSequence()`, которая по заданному произвольному номеру *r* функции $f_r \in F$ в модели $M=(B,D,F)$ определяет биективную равномерную последовательность $d_r \in D$ целых чисел из множества $B=\overline{[1,7]}$. В качестве примера использован произвольный номер $r=3302$.

Пример программы *P010302*

```
namespace P010302
{  class P010302
   {  static void Main(string[] args)
      {  int n = 7;                // длина последовательности
         Console.WriteLine("n = {0}", n);          // монитор
         int[] q = new int[n]; // элементы последовательности
         int r = 3302;          // номер функции в функционале
         Console.WriteLine("r = {0}", r);          // монитор
         DeonSequence(q, r); // вычисление последовательности
         Console.Write("q = ");
         for (int i = 0; i < n; i++)
            Console.Write("{0,4}", q[i]);
         Console.ReadKey();             // просмотр результата
      }
//-------------------------------------------------------------
      static void DeonSequence(int[] q, int r)
      {  int n = q.Length;
```

```
            int nb = n;                    //      число объектов
            int[] b = new int[nb];            // массив объектов
            for (int i = 0; i < n; i++) b[i] = i + 1;
            int nF = 1;    // nF-факториал подпоследовательности
            for (int i = 2; i <= nb; i++) nF *= i;
            int iq = 0;  // индекс для формируемого элемента в q
            for (int z = n - 1; z > 0; z--)      // цикл позиций
            {  int ng = nF / nb;     // число элементов в группе
               int w = r / ng;         // число предыдущих групп
               if ((w * ng) < r) w++;      // номер группы для r
               int zb = w - 1;      // позиция выбора из b для q
               q[iq++] = b[zb];  // элемент в последовательности
               for (int i = zb; i < z; i++) b[i] = b[i + 1];
               r -= (w - 1) * ng;     // r для следующей позиции
               nb--;                    // на один объект меньше
               nF /= z + 1;            // следующий nF-факториал
            }
            q[n - 1] = b[0];            // последний элемент в q
         }
      }
}
```

После выполнения программы на мониторе появляется листинг:

```
n = 7
r = 3302
q =  5  4  3  6  1  7  2
```

Соответствие последовательности q заданному номеру r можно проверить по распечатке моделирования в программе *P010301*, приведенной выше. Функция `DeonSequence()` реализует условие достаточности в доказательстве теоремы об упорядоченной биекции.

В следующей программе *P010303* функция `DeonNumber()` вычисляет факториальный номер r функции $f_r \in F$ для заданной равномерной последовательности $d = d_r \in D$, состоящей из целых чисел множества $B = \overline{[1,n]}$ в модели $M = (B, D, F)$. В качестве примера использована предыдущая последовательность $5,4,3,6,1,7,2$.

Пример программы *P010303*

```
namespace P010303
{  class P010303
   {  static void Main(string[] args)
      {  int[] q = new int[] { 5, 4, 3, 6, 1, 7, 2 };
         int n = q.Length;         // длина последовательности
         Console.WriteLine("n = {0}", n);          // монитор
         Console.Write("q = ");
         for (int i = 0; i < n; i++)
             Console.Write("{0,4}", q[i]);
         int r = DeonNumber(q);   // номер последовательности
         Console.WriteLine("\nr = {0}", r);        // монитор
         Console.ReadKey();             // просмотр результата
```

```
      }
//-------------------------------------------------------------
      static int DeonNumber(int[] q)
      {  int r = 0;          // пока неизвестный номер функции
         int n = q.Length;        // длина последовательности
         int nb = n;          //      начальное число объектов
         int[] b = new int[nb]; // порядковые номера объектов
         for (int i = 0; i < nb; i++) b[i] = i + 1;
         int zF = 1;                     // zF-факториал для b
         for (int i = 2; i < n; i++) zF *= i;
         for (int z = 0; z < n - 1; z++)  // цикл позиций в q
         {  int k = 0;
            for (; k < nb; k++)
               if (q[z] == b[k]) break;
            int rg = k * zF;           // номер перед группой
            r += rg;                 // неполный номер функции
               // сдвиг: удаление элемента b[k] из b
            for (int i = k; i < nb - 1; i++) b[i] = b[i + 1];
            nb--;                     // на один объект меньше
            zF /= n - z - 1;             // zF-факториал для z
         }
         r++;  // учет последнего элемента последовательности
         return r;                            // номер функции
      }
   }
}
```

После выполнения программы на мониторе появляется следующий результат:

```
n = 7
q = 5 4 3 6 1 7 2
r = 3302
```

Соответствие заданного номера *r* последовательности *q* также можно проверить по распечатке моделирования в программе *P010301*. Функция DeonNumber() реализует условие необходимости в доказательстве теоремы об упорядоченной биекции.

Обсуждение. Два современных генератора равномерных случайных величин *MT19937* [16] и *nsDeonYuliTwist32D* [19] превосходят по своим возможностям остальные аналогичные генераторы. Кроме того, отсутствие пропусков и повторений, а также возможности произвольной битовой длины *w* случайных величин вихревого генератор *nsDeonYuliTwist32D* позволяют получать случайные последовательности огромной длины. Эти свойства дают возможность вихревому генератору *nsDeonYuliTwist32D* превзойти генератор *MT19937*.

Согласно исследованию [19], вихревой генератор *nsDeonYuliTwist32D* может создать число $N_T(w)$ случайных последовательностей:

$$N_T\left(w\right)=w\cdot 2^w.$$

Выше факториальное моделирование равномерных последовательностей случайных величин показывает, что при битовой длине *w* случайных величин, число $N_F(w)$ случайных последовательностей составляет

$$N_T(w) = 2^w ! = 1 \cdot 2 \cdot 3 \cdot (2^w - 1) \cdot 2^w.$$

Непосредственный сравнительный анализ этих двух формул свидетельствует о том, что множество факториальных равномерных последовательностей превосходит по их числу множество вихревых последовательностей:

$$N_F(w) > N_T(w).$$

Однако вихревой генератор *nsDeonYuliTwist32D* уже успешно существует [19], а факториальный генератор находится еще только на стадии начального анализа. Очевидно, что только факториальный вариант генерации может обеспечить абсолютно все последовательности случайных величин длиной w бит. Вихревой генератор может создать только часть их них.

Заключение. Генераторы равномерных случайных последовательностей могут допускать повторяемость и пропуски генерации случайных величин. Однако современные вихревые кольцевые генераторы позволяют исключить повторяемость и пропуски генерации. Такой вид технологии использует битовый сдвиг внутри массива, создавая различные вихревые последовательности. Всего можно создать только $w \cdot 2^w$ вихревых последовательностей. Как было доказано в теореме об упорядоченной биекции, полное множество содержит $2^w !$ равномерных последовательностей неповторяющихся случайных величин длиной w бит. Проведенное моделирование подтверждает теоретические результаты. Полученные результаты можно использовать в огромном количестве прикладных задач тестовых испытаний и планировании экспериментов.

ЛИТЕРАТУРА

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

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

3. *White D.R., Clark J., Jacob J., Poulding S.M.* Searching for resource-efficient programs: Low-power pseudorandom number generators // Proc. 10th Annual Conf. on Genetic and Evolutionary Computation. New York, ACM, 2008. P. 1775–1782. DOI: 10.1145/1389095.1389437
URL: http://dl.acm.org/citation.cfm?doid=1389095.1389437

4. *Langdon W.B.* A fast high quality pseudo random number generator for nVidia CUDA // Proc. 11th Annual Conf. on Genetic and Evolutionary Computation. New York, ACM, 2009. P. 2511–2514. DOI: 10.1145/1570256.1570353
URL: http://dl.acm.org/citation.cfm?doid=1570256.1570353

5. *Deon A.F., Menyaev Y.A.* The complete set simulation of stochastic sequences without repeated and skipped elements // Journal of Universal Computer Science. 2016. Vol. 22. Iss. 8. P. 1023–1047. DOI: 10.3217/jucs-022-08-1023

6. *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. New York, ACM, 2009. P. 112–120. DOI: 10.1145/1653662.1653677

7. *Claessen K., Palka M.H.* Splittable pseudorandom number generators using cryptographic hashing // Proc. 2013 ACM SIGPLAN Symp. on Haskell. New York, ACM, 2013. P. 47–58. DOI: 10.1145/2503778.2503784

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

9. *Mandal K., Fan X., Gong G.* Design and implementation of warbler family of lightweight pseudo-random number generators for smart devices // TECS. 2016. Vol. 15. Iss. 1. Article No. 1. DOI: 10.1145/2808230

10. *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

11. *Juratly M.A., Menyaev Y.A., Sarimollaoglu M., et al.* Real-time label-free embolus detection using *in vivo* photoacoustic flow cytometry // PLoS One. 2016. Vol. 11. No. 5. P. e0156269. DOI: 10.1371/journal.pone.0156269
URL: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0156269

12. *Cai C., Carey K.A., Nedosekin D.A., et al. In vivo* photoacoustic flow cytometry for early malaria diagnosis // Cytometry A. 2016. Vol. 89. Iss. 6. P. 531–542. DOI: 10.1002/cyto.a.22854

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

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

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

16. *Matsumoto M., Saito M., Haramoto H., Nishimura T.* Pseudorandom number generation: Impossibility and compromise // Journal of Universal Computer Science. 2016. Vol. 12. Iss. 6. P. 672–690. DOI: 10.3217/jucs-012-06-0672

17. *Deon A., Menyaev Y.* Parametrical tuning of twisting generators // Journal of Computer Science. 2016. Vol. 12. Iss. 8. P. 363–378. DOI: 10.3844/jcssp.2016.363.378

18. *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

19. *Deon A.F., Menyaev Y.A.* Twister generator of arbitrary uniform sequences // Journal of Universal Computer Science. 2017. Vol. 23. Iss. 4. P. 353–384.

20. *Box G.E.P., Muller M.E.* A note on the generation of random normal deviates // The Annals of Mathematical Statistics. 1958. Vol. 29. No. 2. P. 610–611. DOI: 10.1214/aoms/1177706645

21. *Набебин А.А.* Дискретная математика. М.: Научный мир, 2010. 512 с.

22. *Гнеденко Б.В.* Курс теории вероятностей. М.: Едиториал УРСС, 2005. 448 с.

Деон Алексей Федорович — канд. техн. наук, доцент кафедры «Программное обеспечение ЭВМ и информационные технологии» (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 (4018 W Capitol Ave, Little Rock, AR 72205, США).

Просьба ссылаться на эту статью следующим образом:

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

COMPLETE FACTORIAL SIMULATION OF INTEGER RANDOM NUMBER UNIFORM SEQUENCES

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

Random sequences are widely used in theoretical and practical areas of interests in human and technical activities. An important part of these fields refers to the procedures of producing stochastic values. One direction adapts the sequenced generation of pseudorandom numbers and the other direction uses a complete set of all stochastic sequences. The first direction is well studied and traditionally applied in varies areas ranging from cryptography and technical systems to medical and biological trials. The second direction generally uses systems for preliminary universal testing. In this paper, we follow the second direction, where the underlying approaches in modern generators of random numbers are considered. In some of the modern random numbers generators the skipping and repeating random values may be found. We have formed the requirements that if followed help to solve the problems of skipping and repeating. Moreover, we propose novel algorithms based on factorial expansion which provide fast generation of such sequences. Finally, we describe advantages and disadvantages of findings of our research

Keywords

Computer simulation, random number generator, stochastic sequences algorithm

Received 29.06.2017

REFERENCES

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

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

[3] White D.R., Clark J., Jacob J., Poulding S.M. Searching for resource-efficient programs: Low-power pseudorandom number generators. *Proc. 10th Annual Conf. on Genetic and Evolutionary Computation*, New York, ACM, 2008, pp. 1775–1782. DOI: 10.1145/1389095.1389437 Available at: http://dl.acm.org/citation.cfm?doid=1389095.1389437

[4] Langdon W.B. A fast high quality pseudo random number generator for nVidia CUDA. *Proc. 11th Annual Conf. on Genetic and Evolutionary Computation*, New York, ACM, 2009, pp. 2511–2514. DOI: 10.1145/1570256.1570353
Available at: http://dl.acm.org/citation.cfm?doid=1570256.1570353

[5] Deon A.F., Menyaev Y.A. The complete set simulation of stochastic sequences without repeated and skipped elements. *Journal of Universal Computer Science*, 2016, vol. 22, iss. 8, pp. 1023–1047. DOI: 10.3217/jucs-022-08-1023

[6] 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*, New York, ACM, 2009, pp. 112–120. DOI: 10.1145/1653662.1653677

[7] Claessen K., Palka M.H. Splittable pseudorandom number generators using cryptographic hashing. *Proc. 2013 ACM SIGPLAN Symp. on Haskell*, New York, ACM, 2013, pp. 47–58. DOI: 10.1145/2503778.2503784

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

[9] Mandal K., Fan X., Gong G. Design and implementation of warbler family of lightweight pseudorandom number generators for smart devices. *TECS*, 2016, vol. 15, iss. 1, article no. 1. DOI: 10.1145/2808230

[10] 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

[11] 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., Cai Ch. Real-time label-free embolus detection using *in vivo* photoacoustic flow cytometry. *PLoS One*, 2016, vol. 11, no. 5, pp. e0156269. DOI: 10.1371/journal.pone.0156269
Available at: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0156269

[12] Cai C., Carey K.A., Nedosekin D.A., Menyaev Y.A., Sarimollaoglu M., Galanzha E.I., Stumhofer J.S., Zharov V.P. *In vivo* photoacoustic flow cytometry for early malaria diagnosis. *Cytometry A*, 2016, vol. 89, iss. 6, pp. 531–542. DOI: 10.1002/cyto.a.22854

[13] Menyaev Yulian A., Carey Kai A., Nedosekin Dmitry A., Sarimollaoglu Mustafa, Galanzha Ekaterina I., Stumhofer Jason S., Zharov Vladimir 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

[14] Menyaev Y.A., Nedosekin D.A., Sarimollaoglu M., Juratli M.A., Galanzha E.I., Tuchin V.V., Zharov V.P. Optical clearing in photoacoustic flowcytometry. *Biomed. Opt. Express*, 2013, vol. 4, iss. 12, pp. 3030–3041. DOI: 10.1364/BOE.4.003030

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

[16] Matsumoto M., Saito M., Haramoto H., Nishimura T. Pseudorandom number generation: Impossibility and compromise. *Journal of Universal Computer Science,* 2016, vol. 12, iss. 6, pp. 672–690. DOI: 10.3217/jucs-012-06-0672

[17] Deon A., Menyaev Y. Parametrical tuning of twisting generators. *Journal of Computer Science*, 2016, vol. 12, iss. 8, pp. 363–378. DOI: 10.3844/jcssp.2016.363.378

[18] 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.], 2017, no. 2, pp. 86–110 (in Russ.). DOI: 10.18698/0236-3933-2017-2-86-110

[19] Deon A.F., Menyaev Y.A. Twister generator of arbitrary uniform sequences. *Journal of Universal Computer Science*, 2017, vol. 23, iss. 4, pp. 353–384.

[20] Box G.E.P., Muller M.E. A Note on the generation of random normal deviates. *The Annals of Mathematical Statistics*, 1958, vol. 29, no. 2, pp. 610–611.
DOI: 10.1214/aoms/1177706645

[21] Nabebin A.A. Diskretnaya matematika [Discrete mathematics]. Moscow, Nauchnyy mir Publ., 2010. 512 p.

[22] Gnedenko B.V. Kurs teorii veroyatnostey [Theory of probability course]. Moscow, Editorial URSS Publ., 2005. 448 p.

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, str. 1, Moscow, 105005 Russian Federation).

Menyaev Yu.A. — Cand. Sc. (Eng.), Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Science (4018 W Capitol Ave, Little Rock, AR 72205, USA).

Please cite this article in English as:

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.
DOI: 10.18698/0236-3933-2017-5-132-149