Генерирование псевдослучайных распределений

19 октября 2020 г.

Содержание

Введение

В данной работе рассматривается алгоритмы генерации псевдослучайных чисел, подчиненных различным законам распределения. Основным типом алгоритмов являются генераторы псевдослучайных равномерно распределенных чисел. Они используются как сами по себе, так и для получения псевдослучайных чисел других распределений.

В первой части работы дается обзор ряда генераторов псевдослучайных равномерно распределенных чисел, в том числе описывается псевдоустройство `/dev/random` ОС Unix. Для большинства из них приводится алгоритм в виде псевдокода. Все алгоритмы реализованы на языке `C` и протестированы с помощью пакета тестов `dieharder`. На основе результатов тестирования выбираются лучшие алгоритмы для использования.

Вторая части работы посвящена алгоритмы генерации неравномерных распределений. Приводятся краткие сведения из теории случайных процессов, аксиоматические определения пуассоновского и винеровского процессов, а также алгоритмы, позволяющие генерировать эти процессы на компьютере.

1. Алгоритмы генерации равномерно распределенных псевдослучайных чисел

В данном разделе мы познакомимся с несколькими наиболее распространенными генераторами равномерно распределенных псевдослучайных чисел. Такие генераторы служат основой для получения последовательностей псевдослучайных чисел других распределений. Будем предполагать, что компьютер, на котором будут генерироваться псевдослучайные числа имеет 64 разрядный процессор, а языке программирования поддерживает беззнаковый целый тип (например, `unsigned long long int` в Си).

1.1. Линейный конгруэнтный метод

Линейный конгруэнтный метод был впервые предложен в 1949 году Д. Г. Лехмером (D. H. Lehmer) [1, 2]. Алгоритм 1 задается одной формулой:

$$x_{n+1} = (ax_n + c) \mod m, \;\; n \geqslant 0,$$

где m — натуральное число, называемое *модулем* (mask), a — *множитель* (multiplier) ($0 \leqslant a < m$), c — *приращение* ($0 \leqslant c < m$), x_0 — начальное значение, *зерно* (seed). Результатом многократного применения данной рекуррентный формулы является *линейная конгруэнтная последовательность* $x_1, \ldots, x_n$. Особый случай $c = 0$ называется *мультипликативным* конгруэнтным методом. Для краткого обозначения данного метода будем использовать аббревиатуру `LCG` (от английского названия **l**inear **c**ongruential **g**enerator).

Algorithm 1 `LCG` линейный конгруэнтный генератор

Require: n, $seed$
  const $m \leftarrow 2^{64}$
  const $a \leftarrow 6364136223846793005$
  const $c \leftarrow 1442695040888963407$
  $x_0 \leftarrow seed$
  for $i = 0$ to n **do**
    $x_i = (a \cdot x_{i-1} + c) \mod m$
  end for
  return $\{x_0, x_1, \ldots, x_n\}$

Числа m, a, c называют «волшебными» или «магическими», так как их значения задаются в коде программы и выбираются исходя из опыта применения генератора. Качество генерируемой последовательности существенно зависит от правильного выбора данных параметров. Последовательность $\{x_1, x_2, ...\}$ периодична и ее период зависит от числа m, которое поэтому должно быть большим. На практике выбирают m равным машинному слову (для 32-х битной архитектуры — 2^{32} и для 64-х битной — 2^{64}). В книге [1, 2] рекомендуется выбрать

$$a = 6364136223846793005, \; c = 1442695040888963407, \; m = 2^{64} = 18446744073709551616.$$

Другие значения можно найти в статье [3], где приведены объемные таблицы с оптимальными значениями a, b и m.

Стоит отметить, что в попытках усовершенствовать `LCG` метод были предложены во-первых *квадратичный конгруэнтный метод*:

$$x_n = (ax_{n-1}^2 + bx_{n-1} + c) \mod m,$$

во-вторых *кубический конгруэнтный метод*:

$$x_n = (ax_{n-1}^3 + bx_{n-1}^2 + cx_{n-1} + d) \mod m.$$

Однако данные методы широкого применения не нашли ввиду имеющихся лучших альтернатив, коотрые мы рассмотрим далее.

В настоящее время линейный конгруэнтный метод и его разновидности представляют по большей части лишь исторический интерес, так как они генерирует сравнительно некачественную псевдослучайную последовательность по сравнению с другими, не менее простыми генераторами. Пользоваться данным генератором имеет смысл только в учебных целях.

1.2. Метод Фибоначчи с запаздываниями

Развитием `LCG` генератора можно считать идею использовать для генерации i-го элемента псевдослучайной последовательности не один, а несколько предыдущих элементов. Согласно [1, 2] первый такой генератор был предложен в начале 50-х годов и основывался на формуле:

$$x_{n+1} = (x_n + x_{n-1}) \mod m.$$

Однако на практике он показал себя не лучшим образом. В 1958 году Дж. Ж. Митчелом (G. J. Mitchell) и Д. Ф. Муром (D. Ph. Moore) был был придуман генератор 2

$$x_n = (x_{n-n_a} + x_{n-n_b}) \mod m,\ n \geqslant \max(n_a, n_b).$$

который получил название *генератора Фибоначчи с запаздыванием* (сокращенно `LFG` от английского названия **l**agged **F**ibonacci **G**enerator).

Как и в случае `LCG` генератора, выбор «магических чисел» n_a и n_b сильно влияет на качество генерируемой последовательности. Авторы предложили использовать следующие значения n_a и n_b:

$$n_a = 24, n_b = 55.$$

Д. Кнут [1, 2] приводит ряд других значений, начиная от $(37, 100)$ и заканчивая $(9739, 23209)$. Длина периода данного генератора велика и в точности равна $2^{e-1}(2^{55} - 1)$ при выборе $m = 2^e$. Однако это преимущество компенсируется необходимостью использовать не одно начальное значение, а последовательность из $\max(n_a, n_b)$ случайных чисел.

В открытой библиотеке GNU Scientific Library (GSL) [4] используется *составной мультирекурсивный* генератор, предложенный в статье [5]. Данный генератор является разновидностью `LFG` и может быть задан следующими формулами:

$$\begin{aligned} x_n &= (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \mod m_1, \\ y_n &= (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \mod m_2, \\ z_n &= (x_n - y_n) \mod m_1. \end{aligned}$$

Algorithm 2 `LFG` генератор Фибоначчи с запаздываниями

$n_a \leftarrow 55$
$n_b \leftarrow 24$
Require: $s_0, s_1, \dots, s_{n_b}$, $n \geqslant 0$
$x_0, x_1, \dots, x_{n_b} \leftarrow r_0, r_1, \dots, r_{n_b}$
for $i = (n_a + 1)$ to n **do**
  if $x_{i-n_a} \geqslant x_{i-n_b}$ **then**
    $x_i = x_{i-n_a} - x_{i-n_b}$
  else if $x_{i-n_a} < x_{i-n_b}$ **then**
    $x_i = x_{i-n_a} - x_{i-n_b} + 1$
  end if
end for
return $\{x_0, x_1, \dots, x_n\}$

Составной характер данного алгоритма позволяет получить большой период, равный $10^{56} \approx 2^{185}$. В библиотеке `GSL` используются следующие значения параметров a_i, b_i, m_1, m_2:

$$\begin{array}{lll} a_1 = 0, & b_1 = 86098, & m_1 = 2^{32} - 1 = 2147483647, \\ a_2 = 63308, & b_2 = 0, & m_2 = 2145483479, \\ a_3 = -183326, & b_3 = -539608. & \end{array}$$

Еще одни метод, предложный в статье [6] также реализован в `GSL`, является разновидностью метода Фибоначчи и определяется формулой:

$$x_n = (a_1 x_{n-1} + a_5 x_{n-5}) \mod m,$$

В GSL использованы следующие параметры:

$$a_1 = 107374182, a_2 = 0, a_3 = 0, a_4 = 0, a_5 = 104480, m = 2^{31} - 1 = 2147483647.$$

Период этого генератора равен 10^{46}.

Генератор `LFG` 2 дает качественную псевдослучайную последовательность, однако это преимущество нивелируется необходимостью задать минимум 55 начальных значений, для генерации которых придется использовать какой-нибудь другой алгоритм.

1.3. Инверсный конгруэнтный генератор

Инверсный конгруэнтный метод (ICG — **I**nversive **C**ongruential **G**enerator) основан на использовании обратного по модулю целого числа.

$$x_{i+1} = (a x_i^{-1} + b) \mod m$$

где a — *множитель* ($0 \leqslant a < n$), b — *приращение* ($0 \leqslant b < n$), x_0 — начальное значение (seed). Кроме того НОД$(x_0, m) = 1$ и НОД$(a, m) = 1$.

Этот генератор превосходит обычный линейный метод, однако сложнее алгоритмически, так как необходимо искать обратные по модулю целые числа, что приводит к медленной скорости генерации чисел. Для вычисления обратного числа обычно применяется расширенный алгоритм Евклида [1, 2, §4.3.2].

1.4. Генераторы с использованием побитовых операций

Большинство генераторов, дающих наиболее качественные псевдослучайные последовательности используют в своих алгоритмах побитовые операции конъюнкции, дизъюнкции, отрицания, исключающей дизъюнкции (`xor`) и побитовые вправо/влево.

1.5. Вихрь Мерсенна

Вихрь Мерсенна (MT — **M**ersenne **T**wister) был разработан в 1997 году (Мацумото и Нишимура [7]). Существуют 32-,64-,128-разрядные версии вихря Мерсенна. Свое название алгоритм получил из-за использования простого числа Мерсенна $2^{19937} - 1$. В зависимости от реализации обеспечивается период вплоть до $2^{216091} - 1$.

Основным недостатком алгоритма является относительная громоздкость и, как следствие, сравнительно медленная работа. В остальном же данный генератор обеспечивает качественную псевдослучайную последовательность и проходит все тесты `DieHarder`. Важным преимуществом является требование лишь одного инициирующего числа (seed). Вихрь Мерсенна используется в качестве стандартного во многих современных языках программирования: Microsoft Visual C++, Maple, Mathematica, FreePascale, PHP6, Python, Ruby, Scilab, C++11 и т.д.

Стандартная имплементация алгоритма, созданная Мацумото и Нишимура свободно доступна по ссылке `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html`. Ниже приведен альтернативный код на языке Си.

```
#include "randomgen.h"

// Вихрь Мерсенна (Mersenne twister, MT)
// стандарт MT19937-64
#define w 64
#define n 312
#define m 156
#define r 31
#define a 0xb5026f5aa96619e9
#define u 29
#define d 0x5555555555555555
#define s 17
#define b 0x71d67fffeda60000
#define t 37
#define c 0xfff7eee000000000
#define l 43
#define f 6364136223846793005
#define lowest_w_bits 0xffffffffffffffff

unsigned long long int MT[n-1];
// idx вместо index так как по видимому существует функция index
unsigned long long int idx = n + 1;

```

```
const unsigned long long int lower_mask = (1llu << r) - 1llu;
// upper_mask = lowest w bits of (not lower_mask)
const unsigned long long int upper_mask =  lowest_w_bits & (~((1llu << r) - 1llu));

void seed_mt(unsigned long long int seed){
  idx = n;
  int i;
  MT[0] = seed;
  for(i=1; i<=n-1; i++){
    MT[i] = lowest_w_bits & (f * (MT[i-1] ^ (MT[i-1] >> (w-2))) + i);
  }
}

void twist(void){
  unsigned long long int  x;
  unsigned long long int  xA;
  int i;
  for(i=0; i<=(n-1); i++){
    x = (MT[i] & upper_mask) + (MT[(i+1) % n] & lower_mask);
    xA = x >> 1;
    if((x % 2) != 0){
      xA = xA ^ a;
    }
    MT[i] = MT[(i+m) % n] ^ xA;
  }
  idx = 0;
}
// extract_number
unsigned long long int mersenne_twister(unsigned long long int seed){
  unsigned long long int  y;

  if(idx >= n){
    if(idx > n){
      // printf("Генератор не инициализирован\n");
      // заменить константу на случайное число
      seed_mt(seed);
    }
    twist();
  }
  y = MT[idx];
  y = y ^ ((y >> u) & d);
  y = y ^ ((y << s) & b);
  y = y ^ ((y << t) & c);
  y = y ^ (y >> l);

```

```
69    ++idx;
70    return lowest_w_bits & y;
71  }
```

1.6. Генераторы `XorShift`

Несколько простых генераторов (алгоритмы 3 и 4), дающих качественную псевдослучайную последовательность были разработаны в 2003 году Дж. Марсальей (G. Marsaglia) [8, 9].

Algorithm 3 Генератор `xorshift*`

Require: n, $seed$
  $x \leftarrow seed$
  $y_0 \leftarrow x$
  for $i = 1$ to n **do**
    $x \leftarrow x \oplus x \gg 12$
    $x \leftarrow x \oplus x \ll 25$
    $x \leftarrow x \oplus x \gg 27$
    $y_i \leftarrow x \cdot 2685821657736338717$
  end for
  return $\{y_0, y_1, \dots, y_n\}$

Algorithm 4 Генератор `xorshift+`

Require: n, $seed_1$, $seed_2$
  for $i = 1$ to n **do**
    $x \leftarrow seed_1$
    $y \leftarrow seed_2$
    $seed_1 \leftarrow y$
    $x = x \oplus (x << 23)$
    $seed_2 = x \oplus y \oplus (x >> 17) \oplus (y >> 26)$
    $z_i \leftarrow seed_2 + y$
  end for
  return $\{z_1, \dots, z_n\}$

1.7. Генераторы KISS

Еще одно семейство генераторов, дающих не менее качественную последовательность псевдослучайных чисел [10], получило шутливую аббревиатуру KISS (**K**eep **I**t **S**imple **S**tupid) в качестве названия ввиду простоты алгоритма. Алгоритма `KISS` используется в процедуре `random_number()` языка `Frotran` (компилятор `gfortran` [11])

Algorithm 5 Генератор `KISS`

Require: n, $seed_0, seed_1, seed_2, seed_3$
  t
  for $i = 1$ to n **do**
    $seed_0 \leftarrow 69069 \cdot seed_0 + 123456$
    $seed_1 \leftarrow seed_1 \oplus (seed_1 << 13)$
    $seed_1 \leftarrow seed_1 \oplus (seed_1 >> 17)$
    $seed_1 \leftarrow seed_1 \oplus (seed_1 << 5)$
    $t \leftarrow 698769069 \cdot seed_2 + seed_3$
    $seed_3 \leftarrow (t >> 32)$
    $seed_1 \leftarrow t$
    $x_i \leftarrow seed_0 + seed_1 + seed_2$
  end for
  return $\{x_1, \dots, x_n\}$

Algorithm 6 Генератор `jKISS`

Require: n, $seed_0, seed_1, seed_2, seed_3$
  t
  for $i = 1$ to n **do**
    $seed_0 \leftarrow 314527869 \cdot seed_0 + 1234567$
    $seed_1 \leftarrow seed_1 \oplus (seed_1 << 5)$
    $seed_1 \leftarrow seed_1 \oplus (seed_1 >> 7)$
    $seed_1 \leftarrow seed_1 \oplus (seed_1 << 22)$
    $t \leftarrow 4294584393 \cdot seed_2 + seed_3$
    $seed_3 \leftarrow (t >> 32)$
    $seed_1 \leftarrow t$
    $x_i \leftarrow seed_0 + seed_1 + seed_2$
  end for
  return $\{x_1, \dots, x_n\}$

1.8. Устройства /dev/random и /dev/urandom

Ниже перевод комментариев к драйверу random.c.

Для создания истинно-случайной последовательности чисел с помощью компьютера, в некоторых Unix системах (в частности GNU/Linux) используется сбор «фонового шума» окружения операционной системы и аппаратного обеспечения. Источником такого случайного шума являются моменты времени между нажатия клавиш пользователем (inter-keyboard timings), различные системные прерывания и другие события, которые удовлетворяют двум требованиям: не детерминированности и сложной доступности для измерения внешним наблюдателем.

Неопределенность из таких источников собирается драйвером ядра и помещается в «энтропийный пул», который дополнительно перемешивается с помощью алгоритма, похожего на алгоритмы вычисления контрольных сум. Когда случайные байты запрашиваются системным вызовом, они извлекаются из энтропийного пула путем взятия SHA хеша от содержимого пула. Взятие хеша позволяет не показывать внутреннее состояние пула, так как восстановление содержимого по хешу считается вычислительно невыполнимой задачей. Дополнительно извлекающая процедура занижает размер содержимого пула для того, чтобы предотвратить выдачу хеша по всему содержимому и минимизировать теоретическую возможость определения его содержимого.

Во вне энтропийный пул доступен в виде символьного псевдоустройства `/dev/random`. а также в виде системного вызова:

```
1 void get_random_bytes(void *buf, int nbytes);
```

Устройство `/dev/random` можно использовать для получения очень качественной последовательности случайных чисел, однако оно возвращает число байт, равное размеру накопленного энтропийного пула, поэтому если требуется неограниченное количесто случайных чисел, то следует использовать символьное псевдоустройство `/dev/urandom` у которого нет данного ограничения, но оно уже генерирует псевдослучайные числа, качество которых, однако, достаточно для большинства не криптографических задач.

1.8.1. Результаты тестов и выводы

Все описнные в первой части генераторы были реализованы на языке Си и протестированы с помощью утилиты `dieharder`. Результаты тестов сведены в следующую таблицу:

Генератор	**Провалено**	**Слабо**	**Пройдено**
`LCG` (линейный конгруэнтный генератор)	52	6	55
`LCG2` (составной линейный конгруэнтный генератор)	51	8	54
`LFG` (линейный генератор Фибоначчи)	0	2	111
`ICG` (инверсный конгруэнтный генератор)	0	6	107
`KISS`	0	3	110
`jKISS`	0	4	109
`XorShift`	0	4	109
`XorShift+`	0	2	111
`XorShift*`	0	2	111
`MT` (вихрь Мерсенна)	0	2	111
`dev/urandom`	0	2	111

Из генераторов, использующих побитовые операции выделяются генераторы `xorshift*`, `xorshift+` и вихрь Мерсенна. Все они дают одинаково качественную последовательность. Алгоритм вихря Мерсенна, однако, намного более громоздок, чем `xorshift*` или `xorshift+`, поэтому для генерирования больших последовательностей предпочтительней использовать `xorshift*` или `xorshift+`.

Среди генераторов не использующих побитовые операции выделяется линейный генератор Фибоначчи, который при тестировании дает результаты на уровне `XorShift+` и вихря Мерсенна. Однако необходимость задать минимум 55 начальных значений для инициализации данного генератора сводит его полезность к минимуму. Инверсный конгруэнтный генератор показывает немногим худшие результаты, но требует всего одно число для инициации алгоритма.

2. Тестирование алгоритмов

2.1. Актуальные наборы статистических тестов

Исторически первым набором статистических тестов для тестирования генераторов псевдослучайных чисел был пакет тестов DieHard [12], созданный в 1995 году Джорджем Марсальей (George Marsaglia). Он распространялся на CD-диске и в настоящее время официальная страница доступна только в виде архива. Данный пакет тестов на данный момент не актуален. Тесты, входившие в его состав, сейчас присутствуют и в других наборах.

Из актуальных в настоящее время наборов тестов можно выделить следующие четыре.

- `TestU01` [13, 14] за авторством Пьера Л'Экуйе и Ричарда Симарда (Pierre L'Ecuyer, Richard Simard). Написана на `ANSI C`. На сегодняшний день является самым известным наборов тестов. Тестирует генераторы, генерирующие числа из интервала $[0, 1)$. Последняя версия `1.2.3` от 18 августа 2009 года.

- `PractRand` [15] за авторством Криса Доти-Хамфри (Chris Doty-Humphrey). Написана на `C++11` с элементами C99. Принимает на вход поток байт, может тестировать 32 и 64 битные генераторы. Способен справляться с очень большими объемами данных. Последняя версия `0.94` от 04 августа 2018 года.

- `gjrand` [16]. Контактов автора на официальном сайте найти не удалось. Написан на `C99`. Принимает на вход поток байт. Поставляется с набором различных генераторов, способных генерировать не только равномерно-распределенные последовательности псевдослучайных чисел, но и последовательности, подчиняющиеся нормальному, пуассоновскому и некоторым другим распределениям. Последняя версия `4.2.1` от 28 ноября 2014 года.

- `DieHarder` [17] за авторством Роберта Брауна. Позиционируется как наследник тестов DieHard. Написан на языке `C`. Требует для своей работы библиотеки `GSL` [4] и может тестировать любой генератор, с интерфейсом в стиле интерфейсов генераторов из GSL. Последняя версия `3.31.1` от 19 июня 2017 года.

В таблице 1 дана сводка основных характеристик обозреваемых пакетов. В колонке **Unix** указана возможность установки под *nix системами. В колонке **Windows** поставлен плюс только если программу возможно собрать без установки `CygWin` или `MinGW`. При должном старании любую библиотеку можно скомпилировать и под Windows тоже.

Пакет	Язык	CMD	Unix	Windows	Версия	Год	Сайт
`TestU01`	ANSI C	-	++	±	1.2.3	18.08.2009	[13]
`PractRand`	C99, C++11	+	+	+	0.94	04.08.2018	[15]
`gjrand`	C99	+	+	±	4.2.1	28.11.2014	[16]
`DieHarder`	C99	+	++	±	3.31.1	19.06.2017	[17]

Таблица 1: Сводная характеристика пакетов для тестирования.

Все перечисленные наборы тестов имеют открытый исходный код. TestU01 и DieHarder доступны для установки через официальные репозитории многих дистрибутивов, в частности Ubuntu 18.10. Два других набора тестов необходимо устанавливать вручную. Каждый

из перечисленных пакетов позволяет проводить тесты как путем подключения библиотек к C/C++ программе пользователя, так и предоставляет утилиту командной строки. Наиболее функциональная утилита командной строки у пакета DieHarder.

Далее опишем установку каждого из перечисленных пакетов на компьютер с GNU/Linux, дистрибутив Ubuntu 18.10.

2.2. Установка пакетов тестов под ОС типа Unix

Опишем процесс установки в домашний каталог пользователя без прав администратора. Все действия выполнялись в операционной системе GNU/Linux Ubuntu 18.10. Использовался компилятор `gcc` 8 версии. Из документации к пакетам следует, что подойдет любой компилятор, поддерживающий стандарты языков C до C99 включительно и C++ до C++11 включительно.

В домашнем каталоге пользователя создадим следующую иерархию каталогов:

```
mkdir -p ~/usr/bin ~/usr/lib ~/usr/share ~/usr/include
```

- Каталог `~/usr/bin` для исполняемых файлов.
- Каталог `~/usr/lib` для файлов разделяемых и статических библиотек (`.so` и `.a`).
- Каталог `~/usr/include` для заголовочных файлов.
- Каталог `~/usr/share` для примеров и документации.

Также в файл `~/.bashrc` добавляем следующие переменные окружения:

```
export PATH="$HOME/usr/bin/:$PATH"
export LD_LIBRARY_PATH="$HOME/usr/lib:$LD_LIBRARY_PATH"
export LIBRARY_PATH="$HOME/usr/lib:$LIBRARY_PATH"
export C_INCLUDE_PATH="$HOME/usr/include:$C_INCLUDE_PATH"
```

это даст возможность командному интерпретатору искать исполняемые файлы в том числе и в каталоге `~/usr/bin`, а компилятору автоматически подключать библиотеки и заголовочные файлы.

2.2.1. Установка TestU01

Для установки TestU01 скачаем zip архив с официального сайта [13], распакуем его и перейдем в корневую директорию.

```
wget http://simul.iro.umontreal.ca/testu01/TestU01.zip
uz TestU01.zip
cd TestU01-1.2.3/
```

TestU01 поставляется с набором скриптов `Autoconf` и `Automake`, поэтому процесс компиляции и установки сводится к выполнению следующих трех команд:

```
./configure --prefix=$HOME/usr
make
make install
```

Указание опции `--prefix=$HOME/usr` позволяет установить программу локально в `~/usr`.

После установки в директории `~/usr/include` будет создано большое количество заголовочных файлов. Для того, чтобы организовать их аккуратней, создадим директорию `~/usr/include/testu01` и перенесем туда все `h` файлы TestU01. То же самое можно сделать и директории `~/usr/lib` для файлов библиотек.

2.2.2. Установка gjrand

Скачиваем архив с исходными кодами с официального сайта [16]

```
wget https://datapacket.dl.sourceforge.net/project/gjrand/gjrand/gjrand.4.2.
↪ 1/gjrand.4.2.1.tar.bz2
tar -xvjf gjrand.4.2.1.tar.bz2
cd gjrand.4.2.1
```

Для компиляции `gjrand` используются bash-скрипты `compile`. Исходные файлы для получения библиотеки расположены в директории `src`.

```
cd src && ./compile
```

на выходе получаем скомпилированный файлы динамической `gjrand.so` и статической `gjrand.a` библиотек, которые вручную переместим в директорию `~/usr/lib`

```
cp -t ~/usr/lib/gjrand gjrand.a gjrand.so
cp -t ~/usr/include/gjrand gjrand.h
```

Вернемся в корневую директорию командой `cd␣..` и пройдемся таким же образом по всем поддиректориям. В каждой из них присутствует скрипт `compile`, который необходимо выполнить.

```
cd testother/src && ./compile
```

На выходе получаем исполняемые файлы в `testother/bin`. Вновь вернемся на уровень выше `cd ..` и перейдем в следующую директорию:

```
cd testmisc/ && ./compile
```

получаем исполняемый файл `kat`, который можно использовать для проверки корректности работы библиотеки. При запуске он выполнит ряд тестов, чтобы проверить работает ли программа так, как рассчитывал автор.

Перейдя в следующий каталог

```
cd testunif/src && ./compile
```

на выходе получим исполняемые файлы для каждого статистического теста. Они будут располагаться в директории `testunif/bin`. Непосредственно в `testunif` появятся два файла `mcp` и `pmcp` — командные утилиты для тестирования генерируемых битовых последовательностей. Эти утилиты используют исполняемые файлы из директории `testunif/bin` поэтому переместить их в другую директорию нельзя.

Следующий каталог

```
cd testfunif/src && ./compile
```

полностью аналогичен предыдущему, только тесты предназначены для чисел типа `double` из интервала $[0, 1)$.

В каталоге `testother`

```
cd testother/src && ./compile
```

находятся тесты для неравномерных распределений. Исполняемые файлы также будут помещены в `testother/bin`.

В результате установки `gjrand` мы получили два файла библиотеки в каталоге `~/usr/lib/gjrand` и заголовочный файл в каталоге `~/usr/include/gjrand`. Остальные утилиты останутся в соответствующих каталогах.

Следует отметить, что процесс установки прошел без ошибок. Автор пакета указал опцию `-Wall` для компилятора и в процессе сборки были видны лишь незначительные предупреждения об использовании условия `if` без ограничивающих скобок.

2.2.3. Установка DieHarder

Для сборки программы необходимо наличие библиотеки GSL (GNU Scientific Library) [4]. В Ubuntu ее можно установить выполнив команду

```
apt install libgsl-dev
```

Скачиваем архив с исходным кодом с официального сайта [17] и распаковываем:

```
wget https://webhome.phy.duke.edu/~rgb/General/dieharder/dieharder-3.31.1.tgz
uz dieharder-3.31.1.tgz
cd dieharder-3.31.1
```

Для установки DieHarder, как и в случае

```
./configure --prefix=$HOME/usr
make
make install
```

Дополнительно при выполнении `./configure` можно указать опцию `--disable-shared`, что приведет к статической компиляции утилиты командной строки и динамическая библиотека не будет создана. Это удобно, если вам нужна только утилита командной строки.

В процессе компиляции в нашем случае произошла ошибка: компилятор не смог найти определение типа `intptr_t`. Это можно исправить, включив в файл `./include/dieharder/`

libdieharder.h заголовочный файл stdint.h. Дальнейшая установка прошла без ошибок. В процессе выполнения make install в директорию ~/usr/lib были перенесены библиотечные файлы, в ~/usr/include была автоматически создана поддиректория dieharder и в нее помещены все заголовочные файлы. В ~/usr/bin оказалась утилита командной строки dieharder.

2.2.4. Установка PractRand

Скачиваем архив с исходным кодом с официального сайта [15] и распаковываем:

```
wget https://netcologne.dl.sourceforge.net/project/pracrand/PractRand_0.94.zip
uz PractRand_0.94.zip
```

В архиве поставляются уже собранные файлы динамической и статической библиотек для ОС Windows. Также присутствует файл проекта для Visual Studio. Для установки же под Unix перейдем в директорию unix, а сборку запустим командой make

```
cd PractRand_0.94/unix
make
```

Для сборки необходим компилятор для языка C++, поддерживающий стандарт C++11.

При сборке возникали ошибки, связанные с регистром букв в названии заголовочных файлов. Так, например файл Coup16.h был указан в test.cpp в нижнем регистре, хотя само имя файла начинается с заглавной буквы. Еще несколько таких ошибок были вызваны той же ошибкой в именах файлов NearSeq.h, birthday.h и каталога Tests. Такая путаница вызвана скорее всего тем, что автор разрабатывал программу под ОС Windows, где регистр значения не имеет.

После устранения ошибок получаем 4 исполняемых файла и статическую библиотеку libpracrand.a.

2.3. Утилиты командной строки

Пакеты PractRand, DieHarder и gjrand имеют в своем составе утилиты командной строки. Данные утилиты позволяют проводить статистические тесты над последовательностью случайных чисел считывая ее из стандартного потока ввода или из файла. Наиболее функциональная утилита входит в состав DieHarder.

2.3.1. Утилиты PractRand

После компиляции и сборки PractRand мы получим в свое распоряжение четыре исполняемых файла.

- RNG_output — позволяет запустить один из генераторов, входящих в состав пакета.
- RNG_test — утилита, предназначенная для проведения статистических тестов над встроенными генераторами или над потоком данных, подаваемом на стандартный ввод.

- `RNG_benchmark` — измерения скорости работы встроенных или добавленных пользователем генераторов.
- `Test_calibration` — для тестирования и настройки наборов тестов.

Для целей тестирования используется `RNG_test`. Для примера, запуск теста для встроенного генератора `jsf32` достаточно выполнить команду

```
RNG_test jsf32
```

Для теста внешней программы нужно подать генерируемый бинарный поток беззнаковых целых чисел на стандартный вход `RNG_test`, например так:

```
./random -G 4 -N 1000000000000000 | RNG_test stdin32
```

Опция `stdin32` заставляет `RNG_test` интерпретировать поток бинарных данных как набор 32-битных чисел. Если указана опция `stdin64`, то числа будут восприниматься как 64 битные, а при указании `stdin` программа сама решит какую разрядность использовать. Для тестирования данных из файла можно поступит, например так

```
cat file.data | RNG_test stdin64
```

Данные в файле также должны быть в бинарном формате. Данные в текстовом формате не поддерживаются.

Тесты проводятся довольно быстро и на выход распечатывается отчет, где перечисляются проваленные и подозрительные тесты. Указав опцию `-p 1` можно заставить программу печатать все результаты, а не только проваленные.

2.3.2. Утилиты gjrand

После сборки появляется множество разных исполняемых файлов. Для тестирования внешних генераторов предназначены два из них. Один расположен в директории `testfunif` и называется `mcp` (master computer program), а второй в директории `testother` и называется `fmcp`. Утилита `mcp` предназначена для тестирования последовательностей целых беззнаковых псевдослучайных чисел, а `fmcp` для чисел из интервала $[0, 1)$. В остальном между ними отличий нет.

Программа `mcp` принимает на стандартный вход только бинарный поток. Делается это с помощью конвейера

```
./random -G 4 -N 1000000000000000 | mcp --big
```

К переданному потоку применяется весть набор тестов. Результаты распечатываются по мере появления. Других опций, кроме настройки объема проверяемых данных (см. таблицу 2) у программы нет.

Опция	Описание
`--tiny`	(10 MB)
`--small`	(100 MB)
`--standard`	(1 GB) (default)
`--big`	(10 GB)
`--huge`	(100 GB)
`--tera`	(1 TB)
`--ten-tera`	(10 TB)
`number`	количество байтов
`--no-rewind`	не повторяться

Таблица 2: Опции `mcp` пакета gjrand

2.3.3. `dieharder`

После установки пакета `DieHarder` в консоли станет доступна команда `dieharder`. Данная команда позволяет запускать и тестировать встроенные в `DieHarder` и библиотеку `GSL` генераторы псевдослучайных чисел. Кроме того, она может тестировать поток случайных чисел из файлов или из стандартного ввода. Подробную справку по всем возможным опциям можно получить указав опцию `-h`. Перечислим кратко наиболее важные из них.

- `-l` — показать список доступных тестов.
- `-d n` — выбрать для применения тест `n`.
- `-a` — применить все тесты.
- `-g -1` — показать список доступных генераторов псевдослучайных чисел. Выбранный генератор можно протестировать указав опцию `-g n`, где `n` — номер генератора из списка.

В списке генераторов присутствуют, в частности, следующие опции.

- Опция `200` позволяет считывать поток бинарных данных, подаваемый на стандартный ввод `stdin`.
- Опции `201` и `202` включают считывания данных из файла в бинарном формате и текстовом формате соответственно.
- Опции `500` и `501` позволяют считывать поток байтов с псевдоустройств `/dev/random` и `/dev/urandom`.

Для тестирования внешних генераторов наиболее предпочтительным способом будет передача непрерывного потока двоичных данных через конвейер (pipe `|`). Сделать это можно, например, следующим способом:

```
./random -G 4 -N 1000000000000000 | dieharder -g 200 -a
```

Программа `random` будет использовать генератор под номером 4 для генерации 10^{15} чисел. Заметим, что это не приведет к зависанию процесса, так как `dieharder` сам закроет канал и завершит процесс, когда проведет все тесты.

В случае считывания псевдослучайных числе из текстового файла, данный файл должен иметь определенную структуру. Каждое псевдослучайное число должно располагаться на новой строке, а в первых строках файла необходимо указать следующие данные: тип чисел (`d` — целые числа двойной точности), количество чисел в файле и разрядность чисел (32 или 64 бита). Приведем пример начала такого файла:

```
type: d
count: 5
numbit: 64
1343742658553450546
16329942027498366702
3111285719358198731
2966160837142136004
17179712607770735227
```

Когда такой файл создан можно передать его `dieharder` для проведения тестирования

```
dieharder -a -g 202 -f file.in > file.out
```

флаг `-f` задает входной файл с числами для анализа. Результаты тестирования будут сохранены в `file.out`. Для полноценного теста более 10^9 чисел, поэтому размер файла может превысить десятки гигабайт. В случае меньшего количества чисел `dieharder` может начать считывать файл сначала, что приведет к ухудшению результатов теста.

3. Алгоритмы генерации псевдослучайных чисел, отличных от равномерно распределенных

Теперь перейдем к вопросу генерации псевдослучайных числе, подчиненных распределениям, отличным от равномерного. Особенно важны нормальное распределение и распределение Пуассона. Выбор именно этих двух распределений мотивирован их ключевой ролью в теории стохастических дифференциальных уравнений. Наиболее общий вид таких уравнений использует два случайных процесса: винеровский и пуассоновский [18]. Винеровский процесс позволяет учесть имплицитную стохастичность моделируемой системы, а пуассоновский процесс — внешнее воздействие. Мы также рассмотрим экспоненциальное распределение и распределение Вейбулла.

3.1. Генерирование равномерно распределенных псевдослучайных чисел из единичного промежутка

Все алгоритмы генерации **не**равномерно распределенных псевдослучайных чисел основаны на том или ином алгоритме генерации равномерно распределенных псевдослучайных чисел. В качестве такого алгоритма будем использовать один алгоритмов `xorshift` как наиболее простой и показавший лучшие результаты в тестировании.

Большинство алгоритмов генерации **не**равномерно распределенных псевдослучайных числе требуют задания случайного числа из интервала $[0, 1]$, в то время как подавляющее большинство генераторов равномерно распределенных псевдослучайных чисел дают последовательность из интервала $[0, m]$, где число m зависит от алгоритма, разрядности операционной системы и процессора.

Для получения чисел из интервала $[0, 1]$ можно поступить двумя способами. **Первый способ** заключается в нормировании полученной последовательности, путем деления каждого ее элемента на максимальный элемент последовательности. Такой подход гарантированно даст 1 в качестве одного из случайных чисел. Однако такой способ плох, когда последовательность псевдослучайных чисел слишком велика и не умещается в оперативную память. В этом случае лучше использовать **второй способ**, который заключается в делении каждого сгенерированного числа на m.

3.2. Генерирование нормального распределения

Метод генерации нормально распределенных псевдослучайных величин предложен в 1958 году Дж. Э. Р. Боксоми П. Мюллером [19] и назван в их честь *преобразованием Бокса-Мюллера*. Метод основан на двух сравнительно простых формулах, которые обычно записываются в одной из следующих форм:

- стандартная форма (она как раз и предложена авторами статьи [19]),
- полярная форма (предложена Дж. Беллом [20] и Р. Кнопом [21]).

Стандартная форма. Пусть x и y — два независимых, равномерно распределенных псевдослучайных числа из интервала $(0, 1)$, тогда числа z_1 и z_2 вычисляются по формуле

$$z_1 = \cos(2\pi y)\sqrt{-2\ln x},\ z_2 = \sin(2\pi y)\sqrt{-2\ln x}$$

и являются независимыми псевдослучайными числами, распределенными по стандартному нормальному закону $\mathcal{N}(0,1)$ с математическим ожиданием $\mu = 0$ и стандартным среднеквадратичным отклонением $\sigma = 1$.

Полярная форма. Пусть x и y — два независимых, равномерно распределенных псевдослучайных числа из интервала $[-1, 1]$. Вычислим вспомогательную величину $s = x^2 + y^2$, если $s > 1$ и $s = 0$, то значения x и y следует отбросить и проверить следующую пару. Если же $0 < s \leqslant 1$ тогда числа z_1 и z_2 вычисляются по формуле

$$z_1 = x\sqrt{\frac{-2\ln s}{s}},\ z_2 = y\sqrt{\frac{-2\ln s}{s}}$$

и являются независимыми псевдослучайными числами, распределенными по стандартному нормальному закону $\mathcal{N}(0,1)$.

При компьютерной реализации данного алгоритма предпочтительней использовать полярную форму, так как в этом случае приходится вычислять только одну трансцендентную функцию ln, а не три (ln, sin cos), как в стандартном варианте. Это компенсирует даже тот факт, что часть исходных равномерно распределенных числе отбрасывается — полярная версия метода все равно работает быстрее. Пример работы алгоритма изображен на рисунке 1

Для получения нормального распределения общего вида из стандартного нормального распределения используют формулу $Z = \sigma \cdot z + \mu$, где $z \sim \mathcal{N}(0,1)$, а $Z \sim \mathcal{N}(\mu, \sigma)$.

Algorithm 7 Генератор нормально распределенных псевдослучайных чисел

Require: $seed_1$, $seed_2$
Require: u_1, u_2, s, x, y
  while True **do** ▷ Бесконечный цикл
    $u_1 \leftarrow rand(seed_1)$
    $u_2 \leftarrow rand(seed_2)$
    $s \leftarrow u_1^2 + u_2^2$
    if $(s > 0)$ & $(s \leqslant 1)$ **then**
      Break
    end if
  end while
  $x \leftarrow u_1\sqrt{-\frac{2\ln s}{s}}$
  $y \leftarrow u_2\sqrt{-\frac{2\ln s}{s}}$
  return x, y

3.3. Генерирование экспоненциального распределения

Пусть x является равномерно распределенным псевдослучайным числом, тогда y, вычисляемый по формуле

$$y = -\frac{1}{\lambda}\ln(1 - x),$$

является экспоненциально распределенным псевдослучайным числом.

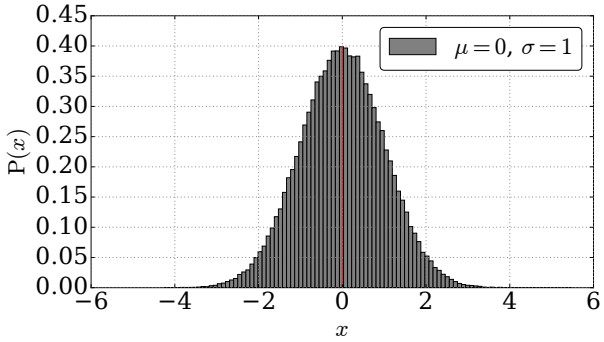

Рис. 1: Нормальное распределение

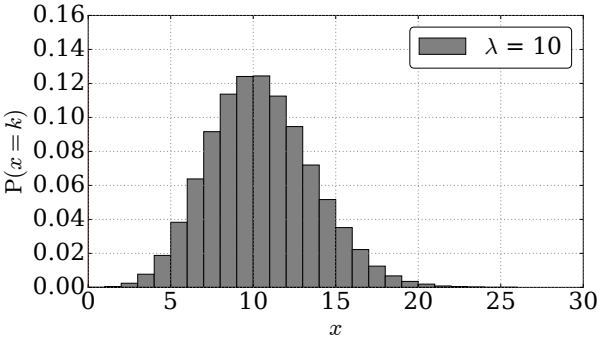

Рис. 2: Распределение Пуассона

3.4. Генерирование распределения Пуассона

Для генерирования распределения Пуассона существует большое число различных алгоритмов [22, 23, 24]. Наиболее простой был предложен Кнутом [1, 2]. Для работы алгоритма 8 необходимо уметь генерировать равномерные псевдослучайные числа из промежутка $[0, 1]$. Пример работы алгоритма изображе на рисунке 2

Algorithm 8 Генератор распределения Пуассона

Require: $seed$, λ
  $\Lambda \leftarrow \exp(-\lambda)$, $k \leftarrow 0$, $p \leftarrow 1$, $u \leftarrow seed$
  repeat
    $k \leftarrow k + 1$
    $u \leftarrow rand(u)$ ▷ генерируем равномерно распределенное случайное число
    $p = p \cdot u$
  until $p > \Lambda$
  return $k - 1$

3.5. Генерирование пуассоновского и винеровского процессов

Вышеописанные генераторы нормального и пуассоновского распределений мы используем для генерации винеровского и пуассоновского стохастических процессов. Дадим вначале определения данных процессов, а затем перейдем к описанию алгоритмов.

3.6. Генерирование распределения Вейбулла

Случайная величина X распределена по *Вейбуллу* [25, 26], если функция плотности вероятности задается следующей формулой:

$$f_W(x; \alpha, s) = \begin{cases} \frac{\alpha}{s} \left(\frac{x}{s}\right)^{\alpha-1} \exp\left[-\left(\frac{x}{s}\right)^{\alpha}\right], \ x \geqslant 0. \\ 0, \ x < 0. \end{cases}$$

Более подробно смотрите в [27, с. 607]. Для генерации псевдослучайных чисел, распределенных по Вейбуллу, используют следующую формулу [27, с. 656]:

$$X = s\left(-\ln(1-u)\right)^{\frac{1}{\alpha}},$$

где u — равномерно распределенное псевдослучайное число из интервала $[0, 1)$.

3.6.1. Определение винеровского и пуассоновского случайных процессов

Пусть $(\Omega, \mathcal{A}, P)$ — вероятностное пространство, где Ω — пространство элементарных событий, $\mathcal{A}$ — σ–алгебра подмножеств Ω (случайные события), P — вероятность или, иначе, вероятностная мера такая что $P(\Omega) = 1$.

Определение (см. [28, 29])**.** Семейство случайных величин $X = \{X_t, 0 \leqslant t \leqslant T\}$, где $X_t \in \mathbb{R}^d$ будет называться d–мерным стохастическим процессом, где совокупность их конечномерных функций распределения

$$F_{X_{t_1}, X_{t_2}, \ldots, X_{t_k}}(x_{i_1}, x_{i_2}, \ldots, x_{i_k}) = P(X_{t_1} \leqslant x_{i_1}, X_{t_2} \leqslant x_{i_2}, \ldots, X_{t_k} \leqslant x_{i_k})$$

для всех $i_k = 1, 2, 3, \ldots$, $k = 1, 2, 3, \ldots$, $x_i \in \mathbb{R}^d$ и $t_k \in \mathrm{T}$

Пространство состояний X — d–мерное евклидово пространство $\mathbb{R}^d$, $d = 1, 2, 3, \ldots$. Временной интервал $[0, T]$, где $T > 0$. В численных методах рассматривают последовательность моментов времени $\{t_0, t_1, t_2, \ldots\}$.

Определение (см. [28, 29])**.** Случайный кусочно-постоянный процесс $N = \{N_t, 0 \leqslant t \leqslant T\}$ с интенсивностью $\lambda > 0$ называется *процессом Пуассона* (пуассоновским) если выполняются следующие свойства.

1. $P\{N_0 = 0\} = 1$, иначе говоря $N_0 = 0$ почти наверное.
2. N_t — процесс с независимыми приращениями, то есть $\{\Delta N_0, \Delta N_1, \ldots\}$ независимые случайные величины; $\Delta N_{t_i} = N_{t_{i+1}} - N_{t_i}$ и $0 \leqslant t_0 < t_1 < t_2 < \ldots < t_n \leqslant T$; $\Delta N_{t_i} = N_{t_{i+1}} - N_{t_i}$ и $0 \leqslant t_0 < t_1 < t_2 < \ldots < t_n \leqslant T$.
3. Существует число $\lambda > 0$ такое, что для любого приращения ΔN_i, $i = 0, \ldots, n-1$, $E[\Delta N_i] = \lambda \Delta t_i$.
4. Если $P(s) = P\{N_{t+s} - N_t > 2\}$, то $\lim\limits_{s \to 0} \dfrac{P(s)}{s} = 0$.

Определение (см. [28, 29])**.** Случайный процесс $W = \{W_t, 0 \leqslant t \leqslant T\}$ называется скалярным *процессом Винера* (винеровским) если выполняются следующие условия.

1. $P\{W_0 = 0\} = 1$, иначе говоря $W_0 = 0$ почти наверное.
2. W_t — процесс с независимыми приращениями, то есть $\{\Delta W_0, \Delta W_1, \ldots\}$ независимые случайные величины; $\Delta W_{t_i} = W_{t_{i+1}} - W_{t_i}$ и $0 \leqslant t_0 < t_1 < t_2 < \ldots < t_n \leqslant T$.
3. $\Delta W_i = W_{t_{i+1}} - W_{t_i} \sim \mathcal{N}(0, t_{i+1} - t_i)$ где $0 \leqslant t_{i+1} < t_i < T$, $i = 0, 1, \ldots, n-1$.

Из определения следует, что ΔW_i — нормально распределенная случайная величина с математическим ожиданием $\mathbb{E}[\Delta W_i] = \mu = 0$ и дисперсией $\mathbb{D}[\Delta W_i] = \sigma^2 = \Delta t_i$.

Винеровский процесс является моделью *броуновского движения* (хаотического блуждания). Если рассмотреть процесс W_t в те моменты времени $0 = t_0 < t_1 < t_2 < ... < t_{N-1} < t_N$ когда он испытывает случайные аддитивные изменения, то непосредственно из определения следует: $W_{t_1} = W_{t_0} + \Delta W_0,\ W_{t_2} = W_{t_1} + \Delta W_1,\ ...,\ W(t_N) = W(t_{N-1}) + \Delta W_{n-1}$, где $\Delta W_i \sim \mathcal{N}(0, \Delta t_i),\ \forall i = 0, ..., n-1$.

Запишем W_{t_n} в виде накопленной суммы: $W_{t_n} = W_{t_0} + \sum\limits_{i=0}^{n-1} \Delta W_i$ и учтем, что $\mathbb{E}[\Delta W_i] = 0$ и $\mathbb{D}[\Delta W_i] = \Delta t_i$ можно показать, что сумма нормально распределенных случайных чисел ΔW_i также является нормально распределенным случайным числом:

$$\mathbb{E}\sum_{i=0}^{n-1} \Delta W_i = 0,\ \ \mathbb{D}\sum_{i=0}^{n-1} \Delta W_i = \sum_{i=0}^{n-1} \Delta t_i = T,\ \ \sum_{i=0}^{n-1} \Delta W_i \sim \mathcal{N}(0, T).$$

Многомерный винеровский процесс $\mathbf{W}_t \colon \Omega \times [t_0, T] \to \mathbb{R}^m$ определяют как случайный процесс составленный из совместно независимых одномерных винеровских процессов $W_t^1, ..., W_t^m$. Приращения $\Delta W_i^\alpha,\ \forall \alpha = 1, ..., m$ являются совместно независимыми нормально распределенными случайными величинами. С другой стороны вектор ΔW_i^α можно представить как многомерную нормально распределенную случайную величину с вектором математических ожиданий $\mu = 1$ и диагональной матрицей ковариаций.

3.6.2. Генерирование винеровского процесса

Для генерирования одномерного винеровского процесса необходимо сгенерировать N нормально распределенных случайных чисел $\varepsilon_1, ..., \varepsilon_N$ и построить их накопленные суммы ε_1, $\varepsilon_1 + \varepsilon_2$, $\varepsilon_1 + \varepsilon_2 + \varepsilon_3$ ... В результате мы получим *выборочную траекторию* винеровского процесса $W(t)$ см. рис. 3.

В случае многомерного случайного процесса следует сгенерировать уже m последовательностей из N нормально распределенных случайных величин.

3.6.3. Генерирование пуассоновского процесса

Генерирование пуассоновского процесса во многом аналогично винеровскому, но теперь необходимо сгенерировать последовательность из чисел распределенных по пуассоновскому закону, а затем вычислить их накопленные суммы. График процесса изображен на рис. 4. Из графика видно, что пуассоновский процесс представляет собой скачкообразное изменение числа произошедших с течением времени событий. От интенсивности λ зависит среднее количество событий за отрезок времени.

Из за такого характерного поведения пуассоновский процесс также называется скачкообразным, а стохастические дифференциальные уравнения где он участвует в качестве второго ведущего процесса получили названия уравнений со скачками [18]

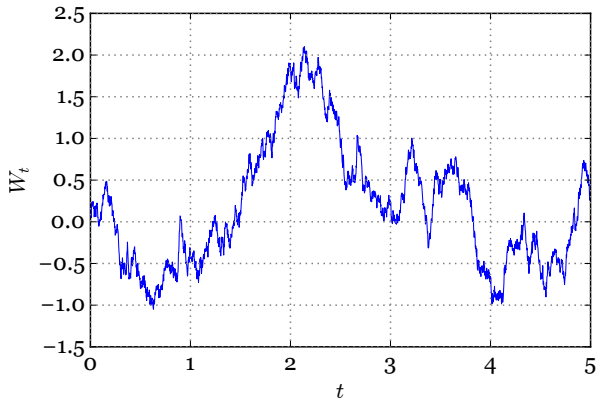

Рис. 3: Винеровский процесс

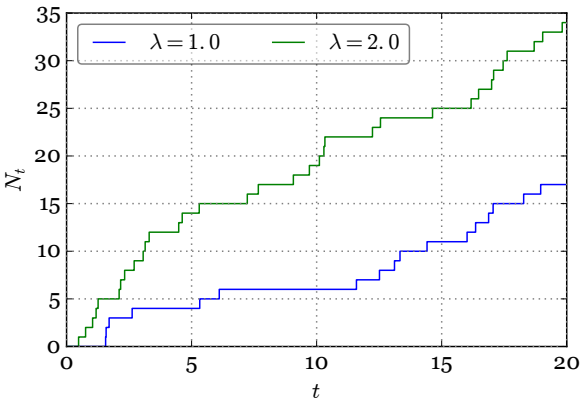

Рис. 4: Пуассоновский процесс

Заключение

Был описан процесс установки четырех наборов тестов для проверки генераторов последовательностей равномерно распределенных псевдослучайных чисел. Были описаны возможности утилит командной строки, поставляемых вместе с ними.

Список литературы

1. *Кнут Д. Э.* Искусство программирования. Т. 2. — 3-е изд. — Москва : Вильямс, 2004. — ISBN 5-8459-0081-6.
2. *Knuth D. E.* The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. Т. 2. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. — ISBN 0-201-89684-2.
3. *L'Ecuyer P.* Tables of linear congruential generators of different sizes and good lattice structure // Mathematics of Computation. — 1999. — Т. 68, № 225. — С. 249—260.
4. *Galassi M.*, *Gough B.*, *Jungman G.*, *Theiler J.*, *Davies J.*, *Booth M.*, *Rossi F.* GSL — GNU Scientific Library. — 2019. — URL: `https://www.gnu.org/software/gsl/`.
5. *L'Ecuyer P.* Combined multiple recursive random number generators // Operations Research. — 1996. — Т. 44, № 5. — С. 816—822.
6. *L'Ecuyer P.*, *Blouin F.*, *Couture R.* A search for good multiple recursive random number generators // ACM Transactions on Modeling and Computer Simulation (TOMACS). — 1993. — Т. 3, № 2. — С. 87—98.
7. *Matsumoto M.*, *Nishimura T.* Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator // ACM Trans. Model. Comput. Simul. — New York, NY, USA, 1998. — Янв. — Т. 8, № 1. — С. 3—30. — ISSN 1049-3301. — DOI: `10.1145/272991.272995`. — URL: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf`.
8. *Marsaglia G.* Xorshift RNGs // Journal of Statistical Software. — 2003. — Т. 8, № 1. — С. 1—6. — ISSN 1548-7660. — DOI: `10.18637/jss.v008.i14`. — URL: `https://www.jstatsoft.org/index.php/jss/article/view/v008i14`.
9. *Panneton F.*, *L'Ecuyer P.* On the Xorshift Random Number Generators // ACM Trans. Model. Comput. Simul. — New York, NY, USA, 2005. — Окт. — Т. 15, № 4. — С. 346—361. — ISSN 1049-3301. — URL: `http://doi.acm.org/10.1145/1113316.1113319`.
10. *Rose G.* KISS: A Bit Too Simple. — 2011. — URL: `https://eprint.iacr.org/2011/007.pdf`.
11. *team T. gfortran.* Using GNU Fortran. — 2015. — С. 304. — URL: `https://gcc.gnu.org/onlinedocs/`.
12. *Marsaglia G.* The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. — 1995. — URL: `https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/`.
13. *L'Ecuyer P.*, *Simard R.* TestU01 — Empirical Testing of Random Number Generators. — 2009. — URL: `http://simul.iro.umontreal.ca/testu01/tu01.html`.
14. *L'Ecuyer P.*, *Simard R.* TestU01: A C library for empirical testing of random number generators // ACM Transactions on Mathematical Software (TOMS). — 2007. — Т. 33, № 4. — С. 22. — URL: `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf`.

15. *Doty-Humphrey C.* PractRand official site. — 2018. — URL: http://pracrand.sourceforge.net/.

16. Gjrand random numbers official site. — 2014. — URL: http://gjrand.sourceforge.net/.

17. *Brown R. G.*, *Eddelbuettel D.*, *Bauer D.* Dieharder: A Random Number Test Suite. — 2017. — URL: %7Bhttp://www.phy.duke.edu/~rgb/General/rand_rate.php%7D.

18. *Platen E.*, *Bruti-Liberati N.* Numerical Solution of Stochastic Differential Equations with Jumps in Finance. — Heidelberg Dordrecht London New York : Springer, 2010.

19. *Box G. E. P.*, *Muller M. E.* A Note on the Generation of Random Normal Deviates // The Annals of Mathematical Statistics. — 1958. — Июнь. — Т. 29, № 2. — С. 610—611. — URL: http://dx.doi.org/10.1214/aoms/1177706645.

20. *Bell J. R.* Algorithm 334: Normal Random Deviates // Commun. ACM. — New York, NY, USA, 1968. — Июль. — Т. 11, № 7. — С. 498—. — ISSN 0001-0782. — DOI: 10.1145/363397.363547. — URL: http://doi.acm.org/10.1145/363397.363547.

21. *Knop R.* Remark on Algorithm 334 [G5]: Normal Random Deviates // Commun. ACM. — New York, NY, USA, 1969. — Май. — Т. 12, № 5. — С. 281—. — ISSN 0001-0782. — DOI: 10.1145/362946.362996. — URL: http://doi.acm.org/10.1145/362946.362996.

22. *Devroye L.* Non-Uniform Random Variate Generation. — New York : Springer-Verlag, 1986.

23. *Ahrens J. H.*, *Dieter U.* Computer methods for sampling from gamma, beta, poisson and bionomial distributions // Computing. — 1974. — Т. 12, № 3. — С. 223—246. — ISSN 1436-5057. — DOI: 10.1007/BF02293108. — URL: http://dx.doi.org/10.1007/BF02293108.

24. *Ahrens J. H.*, *Dieter U.* Computer Generation of Poisson Deviates from Modified Normal Distributions // ACM Trans. Math. Softw. — New York, NY, USA, 1982. — Июнь. — Т. 8, № 2. — С. 163—179. — ISSN 0098-3500. — DOI: 10.1145/355993.355997. — URL: http://doi.acm.org/10.1145/355993.355997.

25. *Fréchet M. R.* Sur la loi de probabilité de l'écart maximum // Annales de la Société Polonaise de Mathematique. — Cracovie, 1927. — Вып. 6. — С. 93—116.

26. *Weibull W.* A statistical distribution function of wide applicability // Journal of Applied Mechanics. — 1951. — Вып. 18. — С. 293—297.

27. *Джонсон Н.Л. Коц С. Б. Н.* Одномерные непрерывные распределения. В 2 частях. Ч. 1. — Бином. Лаборатория знаний, 2012. — (Теория вероятностных распределений). — ISBN 978-5-9963-1351-8.

28. *Kloeden P. E.*, *Platen E.* Numerical Solution of Stochastic Differential Equations. — 2-е изд. — Berlin Heidelberg New York : Springer, 1995. — ISBN 3-540-54062-8.

29. *Øksendal B.* Stochastic differential equations. An introduction with applications. — 6-е изд. — Berlin Heidelberg New York : Springer, 2003. — ISBN 3-540-04758-1.