РОССИЙСКАЯ АКАДЕМИЯ НАУК
Ордена Ленина Сибирское отделение
ИНСТИТУТ ЯДЕРНОЙ ФИЗИКИ
им. Г.И. Будкера СО РАН

А.Д. Букин

РАЗРАБОТКА АЛЬТЕРНАТИВНОГО ГЕНЕРАТОРА СЛУЧАЙНЫХ ЧИСЕЛ ДЛЯ МОДЕЛИРОВАНИЯ ДЕТЕКТОРА СНД

ИЯФ 2007-21

НОВОСИБИРСК
2007

Разработка альтернативного генератора случайных чисел для моделирования детектора СНД

А.Д. Букин

Институт ядерной физики им. Г.И.Будкера
630090, Новосибирск, Россия

Аннотация

В работе приведено описание альтернативного генератора псевдослучайных чисел DRNDM, разработанного для моделирования детектора СНД. Программа моделирования детектора создается на основе программы GEANT-4, основным генератором случайных чисел выбран генератор Ranecu. Однако, для повышения надежности моделирования полезно иметь возможность использовать какой-либо альтернативный генератор. Проведенные статистические тесты позволяют заключить, что генератор DRNDM вполне пригоден для моделирования экспериментов на детекторе СНД.

Development of random number generator for simulation of the detector SND

A.D. Bukin

Budker Institute of Nuclear Physics
630090, Novosibirsk, Russia

Abstract

In the paper a pseudo-random number generator DRNDM is described, which is developed for simulation of the detector SND. The simulation program of the detector is carried out on the base of the package GEANT-4. As the main random number generator it was chosen the generator Ranecu. However for more reliability it is useful to have the possibility to use some alternative generator. The statistical tests used to check the quality of the generator allow to conclude that DRNDM generator is good enough to be used to simulate experiments with SND detector.

1 Введение

Моделирование экспериментов с детектором СНД [1] на коллайдере ВЭПП-2000 будет проводиться на основе программы GEANT [2]. Важным элементом любой программы моделирования является генератор случайных чисел с равномерным распределением вероятностей. В принципе, программа GEANT4 допускает использование любого генератора. Однако, в моделировании экспериментов немаловажно, как организована процедура проведения статистически независимых расчётов одного и того же процесса. В настоящий момент моделирование СНД настроено на использование генератора Ranecu [3]. Для того, чтобы обозначить для GEANT использование конкретной программы-генератора случайных чисел, надо включать строку вида

CLHEP::HepRandom::setTheEngine(new CLHEP::RanecuEngine);

Для такого ответственного дела, как моделирование эксперимента, желательно иметь возможность использовать хотя бы один альтернативный генератор. Данная работа посвящена реализации на языке C^{++} генератора DRNDM [4] с интерфейсом, совместимым с требованиями к генератору при моделировании СНД.

2 Алгоритм

Алгоритм, реализованный в данной программе, включает в себя (как частный случай) алгоритм программы DRNDM, реализованный для OS VM, VAX и LINUX на языке Фортран.

Для реализации алгоритма необходимо уметь получать младшие M двоичных разрядов в произведении двух целых чисел.

$$k_{i+1} = k_i \cdot K_R \bmod 2^M, \tag{1}$$

где k_i — i-е случайное число (целое), $0 < k_i < 2^M$, K_R — производящая константа ряда. Для правильной работы алгоритма числа k_i, K_R должны

быть нечётными, и более того,

$$K_R = 3 \text{ или } K_R = 5 \bmod 2^3. \tag{2}$$

В данной реализации генератора числа M и K_R могут быть модифицированы. При $M = 32$ по умолчанию производящая константа принимается равной $K_R = K_1 = 69069$ (такой же, как в генераторе RNDM из библиотеки CERN, при $M = 63$ $K_R = K_2 = 70369817985301$ (такая же, как в DRNDM на Фортране). В промежуточных случаях принимаются урезанные части этих констант (отбрасываются старшие биты в их двоичном представлении). В двоичном виде эти константы равны

$$K_1 = 00010000110111001101_2 = 10DCD_{16},$$

$$K_2 = 010000000000000001000000000000010000000100010101_2 = \\ = 400040010115_{16}$$

Если $M > 63$, то по умолчанию константа K_2 наращивается «сверху» двоичными разрядами по следующему простому правилу: 64-й бит полагается равным 1, следующие 3 бита — нулевые, затем снова одна единица, затем 3 нуля и т.д., пока не заполнятся $\frac{2}{3}M$ двоичных разрядов (старшие $\frac{1}{3}M$ разрядов остаются нулевыми). Имеется возможность задать целиком производящую константу (принимаются только константы, удовлетворяющие условию (2)).

Начальное случайное число по умолчанию полагается равным

$$k_1 = 2^{E[M/4]} + 1, \tag{3}$$

однако, можно ввести произвольное нечётное число. Здесь $E[x]$ — целая часть x.

В качестве выходного случайного числа выдаётся случайное число в интервале $(0, 1)$ с двойной точностью. Несмотря на ускорение работы алгоритма при $M < 63$ (скорость работы алгоритма повышается с уменьшением M), это навряд ли является достойной компенсацией за ухудшение статистических свойств ряда.

Период ряда при любой производящей константе, удовлетворяющей условию (2), равен

$$T = 2^{M-2} \tag{4}$$

3 Интерфейс программы

Основой генератора является класс RndDrndmEngine. Размер разрядной сетки M задаётся в конструкторе при создании объекта класса и не может быть изменён впоследствии. Конструктор по умолчанию (без аргументов) задаёт $M = 63$. Если в конструкторе есть аргумент, то он воспринимается, как значение M, которое должно быть в интервале $(8, 1000)$.

Скрытые данные (private):

size_t M; // разрядность целочисленной арифметики

size_t Mb; // длина целочисленных переменных в словах (Mb=E[(M+15)/16])

std::vector<size_t> Kr ; // Производящая константа

std::vector<size_t> ki ; // Очередное случайное целое число

mutable std::vector<size_t> kr1; // рабочее поле

mutable std::vector<size_t> kr2; // рабочее поле

mutable std::vector<size_t> kr3; // рабочее поле

Конструктор:

RndDrndmEngine::RndDrndmEngine(size_t M=63) — M задаётся в аргументе, остальные константы формируются по описанным в предыдущем разделе правилам.

Массивы Kr, ki, $kr1$, $kr2$, $kr3$ представляют длинные целые числа без знака, по 16 бит в каждом слове. Слово $Kr[0]$ содержит самые младшие 16 разрядов числа:

$$Kr[0] = K_R \bmod 2^{16},$$

следующие слова — в порядке возрастания. Последнее слово может иметь не все биты значащими, это зависит от того, делится ли M нацело на 16. Если не делится, то старшие $(16 - M \bmod 16)$ разрядов последнего слова всегда равны нулю.

Общие функции (public):

bool setRandomSeed(const std::string& seed) — установить начальное случайное число. Если текстовое поле содержит первым символом «z» или «Z», то остальные символы представляют начальное случайное число (целое) в 16-ричном формате (из символов 0123456789ABCDEFabcdef), если первый символ «b» или «B», то

число представлено в двоичном формате, иначе — десятичное представление. Пробелы игнорируются (можно вставлять без ограничений). Ошибкой считается, если встретился символ, недопустимый в данной кодировке, или присутствует ненулевой разряд свыше разрешённых M двоичных разрядов, или младший двоичный разряд равен нулю. В этом случае функция возвращает «false». Если всё хорошо — возвращается «true». *Запись числа всегда начинается со старших разрядов!*

std::string getCurrentSeed(SeedType = DEC) const — возвращается указатель на текстовую строку, содержащую текущее случайное целое число в кодировке, в точности соответствующее правилам, описанным для функции setRandomSeed, так что может быть без изменений использован как её входной параметр. Параметр SeedType управляет выбором кодировки (может быть равен DEC, BIN, или HEX).

bool setGenerConst(const std::string& genconst) — установить новую производящую константу ряда. Если текстовое поле содержит первым символом «z» или «Z», то остальные символы представляют производящую константу ряда — число (целое) в 16-ричном формате (из символов 0123456789ABCDEFabcdef), если первый символ «b» или «B», то число представлено в двоичном формате, иначе — десятичное представление. Пробелы игнорируются (можно вставлять без ограничений). Ошибкой считается, если встретился символ, недопустимый в данной кодировке, или присутствует ненулевой разряд свыше разрешённых M двоичных разрядов, или младшие три двоичных разряда не равны числу 3 или 5. В этом случае функция возвращает «false». Если всё хорошо — возвращается «true». *Запись числа всегда начинается со старших разрядов, но если разрядов меньше, чем определено в M, то автоматическое дополнение нулями производится в старших разрядах!*

void skip(size_t n1, size_t n2, size_t n3) — пропуск заказанного количества случайных чисел. Полное количество пропущенных случайных чисел равно $N_{skip} = n_1 \cdot n_2 \cdot n_3$. Так как генерация случайных чисел занимает непренебрежимое время процессора, то организовывать такой пропуск большого количества случайных чисел в виде простого цикла с вызовом генератора недопустимо. Естественно, здесь использован алгоритм, применимый только к мультипликативным генераторам. Время счета, необходимое на пропуск боль-

шого количества N_{skip} случайных чисел, в этом алгоритме растет примерно как $\ln(N_{skip})$.

double flat() — образуется следующее случайное целое число k_i и возвращается дробное число $r_i = k_i/2^M$

long* getCurrentSeeds() — получение текущего целого случайного числа в виде массива чисел, тождественного массиву ki.

void setSeeds(const long* seeds, HepInt) — установка нового целого случайного числа ki. seeds — указатель на массив 16-битовых целых чисел размерности Mb. Эти числа переносятся в ki без изменения. В этом алгоритме случайные числа должны быть нечётными. Если проверка показывает, что предоставлено чётное число, то выдаётся сообщение и исполнение аварийно завершается. Второй аргумент не используется — введён для совместимости с Ranecu.

Защищённые функции (protected):

void multiply(const std::vector<size_t>& n1, const std::vector<size_t>& n2, std::vector<size_t>& n3) — умножение целых чисел n1 и n2 и запись младших M двоичных разрядов в n3.

4 Проверка работы программы

Основой проверки будет сравнение с работающей программой DRNDM, но не только. Примитивная проверка — цикл генерации сотни чисел с распечаткой этих чисел проделаем в первую очередь.

4.1 Ввод и вывод начального случайного числа

Здесь, кроме того, что всё работает и не сбивается, проверим ввод и вывод друг на друге: введём начальное число и тут же его выведем — числа, естественно, должны быть одни и те же.

Ввод производящей константы проверим таким образом: при $M = 80$ установим начальное случайное число, равное 1. Установим производящую константу $z1cd2505$, сгенерируем случайное число и выведем текущее целое случайное число. Оно должно совпасть с производящей константой.

Всё совпало!

4.2 Сравнение 10 случайных чисел и ещё десяти после пропуска 100000 чисел

В программе DRNDM на фортране нет функции «пропустить заданное количество чисел», поэтому там это сделано просто циклом генерации 100000 чисел.

Начальное случайное число равно 1.

В программе RndDrndmEngine на C^{++} эта последовательность случайных чисел получена двумя способами: первый в точности повторил тестовую программу на Фортране (простой цикл), во втором использована функция skip(10, 100, 100). Результирующие случайные числа приведены в том виде, как напечаны командой $print\ *, ri$ на Фортране и $std::cout << ri$ на C^{++}.

Номер	DRNDM	RndDrndmEngine
1	0.000007629	0.00000762951
2	0.129242008	0.129242
3	0.143925196	0.143925
4	0.437236140	0.437236
5	0.461373618	0.461374
6	0.920593861	0.920594
7	0.277040276	0.27704
8	0.487567789	0.487568
9	0.456381667	0.456382
10	0.0624851025	0.0624851
	⋮	
100011	0.591521056	0.591521
100012	0.638002876	0.638003
100013	0.891796358	0.891796
100014	0.151137893	0.151138
100015	0.0255085967	0.0255086
100016	0.000103838165	0.000103838
100017	0.176406997	0.176407
100018	0.334234166	0.334234
100019	0.466333743	0.466334
100020	0.425019447	0.425019

Здесь кроме самого алгоритма генерации проверена функция пропуска заданного количества чисел ряда.

После исправления ошибки в функции *skip* всё заработало, как надо.

4.3 Непосредственная проверка одного шага генерации при большой разрядности

Для заданной производящей константы и исходного случайного числа при $M = 80$ проведем вычисления на одном шаге вручную и сравним результат:

```
  00000000011111111122222222233333333334444444445555555556666666667777777778
  12345678901234567890123456789012345678901234567890123456789012345678901234567890

  ----------------------------------------------------------------------------------

× 00000000000000000000000000100000000000010000000000100000000100000010000100101011
  10010000101011110100010010000100010010101110111011000111111010101110111000001011

  ----------------------------------------------------------------------------------

  10010000101011110100010010000100010010101110111011000111111010101110111000001011
  00100001010111010001001000010001001010111011101100011111101010111011100000 1011
  10000101011110100010010000100010010101110111011000111111010101110111000001011
  00010101111010001001000010001001010111011101100011111101010111011100000 1011
+ 10101111010001001000010001001010111011101100011111101010111011100000 1011
  11101000100100001000100101011101110110001111110101011101110000 1011
  01001000010001001010111011101100011111101010111011100000 1011
  10001001010111011101100011111101010111011100000 1011
  11101110110001111110101011101100000 1011
  11111101010111011100000 1011

  ----------------------------------------------------------------------------------

= 10100011000011011100010000011001010001010101000100011101110101010110011011011001
```

Полученная мантисса соответствует случайному числу 0.636928802651997886. Проверка Drndm дала точно такие результаты.

4.4 Время счёта и равномерность

Представляет интерес время счёта при разной разрядности. Заодно проверим и равномерность распределения, так как для обеих целей нужна большая статистика.

В табл. 1 приведены результаты замера времени счёта программы RndDrndmEngine 12 февраля 2007 года на машине SNDAS2, полученные командой time. При повторении одного и того же счёта результаты «болтаются» в пределах 5%. По-видимому, это можно считать точностью измерения. В этом же цикле работают команды построения гистограммы, но время, потребное на эту процедуру, не превышает 0.1 мкс, чем вполне можно пренебречь.

На рис. 1 представлен график зависимости времени счёта от количества бит в мантиссе. Время счёта программы очень слабо зависит от производящей константы и ещё меньше от начального числа. В то же время проверка согласия распределения по случайным числам по критерию χ^2

Таблица 1: Время счёта программы RndDrndmEngine, замеренное на машине SNDAS2 командой time при разных значениях параметра M. Nloop — количество повторений цикла, t_{tot} — полное время счёта, τ — время на одно случайное число, T — период ряда.

M	16	32	45	63	80	150
Nloop	$6.2 \cdot 10^6$	$3.1 \cdot 10^6$	$2.2 \cdot 10^6$	$1.5 \cdot 10^6$	$1.2 \cdot 10^6$	$0.6 \cdot 10^6$
t_{tot}, сек	4.30	4.94	5.87	6.78	8.22	13.62
τ, μсек	0.69	1.59	2.67	4.52	6.85	22.7
Период T	$1.6 \cdot 10^4$	$1.1 \cdot 10^9$	$8.8 \cdot 10^{12}$	$2.3 \cdot 10^{18}$	$3.0 \cdot 10^{23}$	$3.6 \cdot 10^{44}$
χ^2 [1]	31.06	98.12	89.51	78.59	91.73	112.23
$P_{99}(\chi^2)$	1.0000	0.5061	0.7420	0.9352	0.6852	0.1715
χ^2 [2]	31.09	106.62	74.37	90.59	80.41	96.10
$P_{99}(\chi^2)$	1.0000	0.2825	0.9694	0.7149	0.9140	0.5638

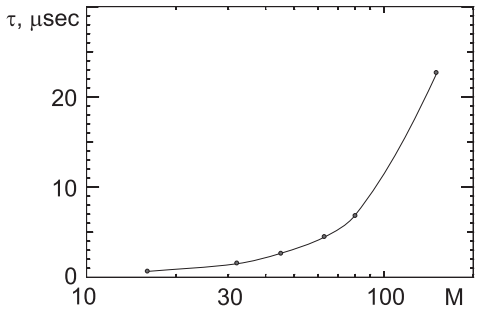

Рис. 1: Использование процессорного времени машины SNDAS2 на одно случайное число в программе RndDrndmEngine в зависимости от параметра M (число бит в целочисленной арифметике).

(приведено в той же табл. 1) сильно зависит от производящей константы и разрядности (для любой разрядности можно найти много «очень плохих констант») и при разных начальных случайных числах подвержена статистическим флуктуациям («псевдо*случайные*» числа, однако!). Можно было бы составить список «хороших» производящих констант для нескольких M, но это обширная работа. Поэтому все эти проверки проведены для производящих констант, устанавливаемых по умолчанию, и для двух вариантов начальных случайных чисел. Если устанавливается «нестандартная» производящая константа, для неё необходима отдельная проверка статистических свойств (в принципе!).

Из таблицы явно видно, что при $M = 16$ степень согласия недопустима высока. Это понятно, почему — статистика событий в гистограмме во много раз больше периода ряда. Во всех остальных случаях период много больше использованной статистики, поэтому поведение χ^2 укладывается в разумные статистические пределы.

Наша старая добрая программа DRNDM тратит 0.16 мкс на одно число. Программа Ranecu, которая в настоящий момент используется нами в GEANT4, на одно обращение тратит 0.12 мкс (период ряда в этой программе, наверно, равен $2 \cdot 10^{18}$). Получается, что RndDrndmEngine в несколько раз проигрывает по времени счёта этим двум программам. Всё же её параллельное использование с генератором Ranecu не лишено смысла. Например, половину статистики моделировать с одним генератором, другую половину — с другим. Сравнение результатов поможет оценить систематическую ошибку, связанную с неидеальными статистическими свойствами генераторов. Различие в скорости работы программ может сильно маскироваться другими вычислениями (например, работа геометрических программ), поэтому сравнение скоростей работы желательно провести ещё и для полной программы моделирования СНД.

Рис. 2 представляет гистограммы распределения по случайному числу в разных вариантах RndDrndmEngine.

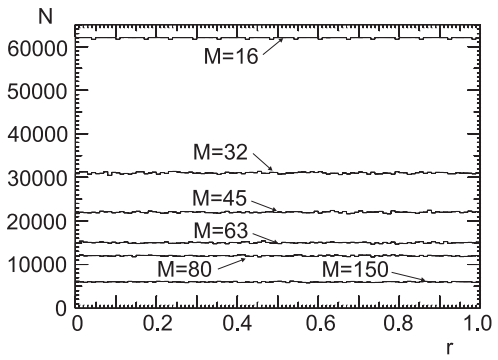

Рис. 2: Распределение вероятностей по случайным числам r генератора RndDrndmEngine при разных значениях параметра M

Распределение вероятностей по случайным числам от Ranecu приведено на рис. 3. По критерию χ^2 значение $\chi^2/n_D = 106.96/99$ соответствует $P = 0.2748$.

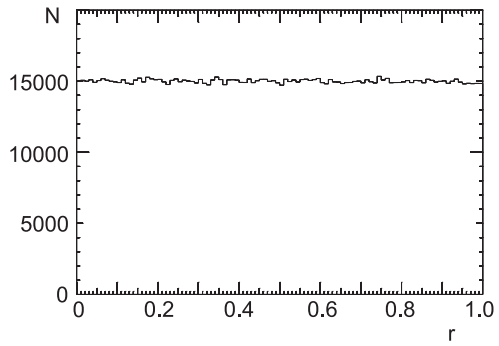

Рис. 3: Распределение вероятностей по случайным числам r генератора RndRanecuEngine

4.5 Парные корреляции

Истинные случайные числа должны иметь нулевые парные коэффициенты корреляции r_i и r_{i+k}:

$$q_k = 12 \cdot \left\langle \left(r_i - \frac{1}{2} \right) \left(r_{i+k} - \frac{1}{2} \right) \right\rangle = 0. \tag{5}$$

Для псевдослучайных чисел такое точное равенство маловероятно, но величина коэффициента для хороших генераторов должна быть маленькой. При проверке мы вычисляем среднее значение:

$$q_k \approx \frac{12}{N} \sum_{i=1}^{N} \left(r_i - \frac{1}{2} \right) \left(r_{i+k} - \frac{1}{2} \right), \tag{6}$$

и полученный результат имеет две составляющие ошибки: систематическую из-за ошибок округления и статистическую. Статистическая ошибка может быть оценена как

$$\Delta q_{stat} = \frac{1}{\sqrt{N}} \tag{7}$$

и уменьшается с ростом статистики, систематическая же не уменьшается с ростом N: при каждой операции добавления нового слагаемого в сумму ошибка составляет порядка 10^{-15}, и учитывая, что сама сумма близка к нулю, а ошибка знакопеременная и накопленная сумма в итоге делится на количество слагаемых, можно надеяться, что систематическая ошибка не

растёт и составляет как раз эти 10^{-15}. Но если коэффициент корреляции не равен нулю, то сумма растёт

$$\sum_{i=1}^{N}\left(r_i-\frac{1}{2}\right)\left(r_{i+k}-\frac{1}{2}\right)\sim\frac{1}{12}N\cdot q,$$

и ошибка прибавления i-го слагаемого уже составляет примерно $\frac{1}{12}i\cdot q\cdot 10^{-15}$, что для всей суммы соответствует примерно

$$\Delta q_{syst}\approx\frac{N}{2}\,|q|\cdot 10^{-15}<N\cdot 10^{-15}. \tag{8}$$

Естественно, хочется, чтобы систематическая ошибка была пренебрежимо мала по сравнению со статистической, т.е.

$$\Delta q_{stat}\gg\Delta q_{syst}\Longrightarrow N^{3/2}\ll 10^{15}\Longrightarrow N\ll 10^{10}. \tag{9}$$

Это означает, что мы не можем проверить такой коэффициент корреляции на равенство нулю точнее, чем 10^{-5}.

Далее будем использовать «безразмерный» коэффициент корреляции:

$$Q_k=\frac{q_k}{\Delta q_{stat}}=\frac{12}{\sqrt{N}}\sum_{i=1}^{N}\left(r_i-\frac{1}{2}\right)\left(r_{i+k}-\frac{1}{2}\right), \tag{10}$$

отличие которого от нуля сразу представлено в единицах стандартного отклонения.

Кроме генераторов Ranecu и Drndm будем также испытывать заведомо плохой генератор RANDU, который тоже является мультипликативным генератором, но с плохой производящей константой: $K_R=65539=z10003$ для $M=29$ (продукт фирмы IBM на заре развития науки моделирования).

В табл. 2 представлены результаты проверки парных корреляций для разных генераторов.

Видно, что все генераторы, в том числе и RANDU, выглядят неплохо в этом тесте. Статистика высокая, нормировочная $\Delta q_{stat}=0.001$, поэтому можно сказать, что коэффициенты корреляции отличаются от нуля не больше, чем на несколько тысячных в абсолютном измерении. Проверим, как изменится для случая $M=80$ максимальное отклонение $Q_6=3.19$, если статистику увеличить в 10 раз.В этом случае максимальный коэффициент корреляции стал $Q_6=2.08$. Изменим начальное случайное число на $zFFF$. После этого получаем $Q_6=-0.72$, а новый максимум стал $Q_5=-1.89$. Это согласуется с тем, что наблюдаемые максимальные значения могут быть статистическими выбросами.

Таблица 2: Нормированный на стат. ошибку коэффициент корреляции Q_k для разных генераторов и $k = 1 \div 10$. Статистика везде $N = 10^6$ событий.

Генератор	Значения Q_k									
	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
RANDU: $M = 29$ $K_R = z10003$	1.79	0.33	0.93	0.37	0.85	0.51	-2.42	0.63	1.68	0.78
Drndm: $M = 16$ $K_R = zDCD$	0.27	3.52	-0.28	-0.20	-0.67	-0.09	-0.30	0.51	0.11	0.74
Drndm: $M = 32$ $K_R = z10DCD$	-0.21	0.65	1.08	0.46	-2.00	0.13	-1.68	-0.37	-0.01	-1.18
Drndm: $M = 45$ $K_R = z40010115$	-0.39	-1.12	1.11	0.51	-1.81	0.65	-0.73	0.21	0.71	0.86
Drndm: $M = 63$ $K_R = z400040010115$	-0.99	0.24	-3.78	-1.44	1.46	1.37	0.64	-1.79	-0.10	-0.78
Drndm: $M = 80$ $K_R = z400040010115$	-0.20	0.13	1.26	0.28	-1.21	3.19	0.80	0.30	2.18	-1.06
Drndm: $M = 150$ $K_R = z888888888888$ $\hookrightarrow$ 000400040010115	-0.83	-0.94	-0.98	-0.35	-0.69	2.60	-0.74	-0.45	-0.23	0.42
Ranecu	1.39	-2.66	1.37	1.02	-0.61	-0.55	1.84	0.08	0.14	-0.55

4.6 Корреляции в многомерном пространстве

Один из мощных тестов — проверка, как выглядит функция распределения в n-мерном пространстве (спектральный анализ).

Пусть точки в n-мерном кубе со стороной, равной единице, имеют координаты, образованные последовательными числами псевдослучайного ряда. Если бы они были истинно случайными числами, то распределение вероятностей записывалось бы в следующем виде

$$\mathrm{d}W = F(r_1, r_2, \ldots, r_n)\,\mathrm{d}r_1\,\mathrm{d}r_2 \cdots \mathrm{d}r_n, \quad F(r_1, r_2, \ldots, r_n) = 1. \tag{11}$$

Разложение функции плотности вероятности в n-мерный ряд Фурье выглядит следующим образом:

$$F(r_1, r_2, \ldots, r_n) = \sum_{k_j=-\infty}^{+\infty} A(\overrightarrow{\mathrm{k}}) \exp\left[2\pi i \overrightarrow{\mathrm{k}}\,\overrightarrow{\mathrm{r}}\right]. \tag{12}$$

Амплитуда A в этом разложении равна

$$A(\overrightarrow{\mathrm{k}}) = \int F(r_1, r_2, \ldots, r_n)\,\mathrm{d}\overrightarrow{\mathrm{r}} \cdot \exp\left[-2\pi i \overrightarrow{\mathrm{k}}\,\overrightarrow{\mathrm{r}}\right]. \tag{13}$$

Эти соотношения очевидны, неочевидно только, что любая функция может быть однозначно представлена в виде такого ряда, но это выходит за рамки данной работы. Если функция распределения тождественно равна единице, то результат тривиальный:

$$A(\overrightarrow{0}) = 1, \quad A(\overrightarrow{\mathrm{k}} \neq 0) = 0. \tag{14}$$

Если какая-то амплитуда существенно отлична от нуля, то в n-мерном пространстве плотность вероятности имеет волнообразную структуру.

Посмотрим, как оценить амплитуды по значениям случайных чисел ряда.

$$A(\overrightarrow{\mathrm{k}}) = \int \mathrm{d}W \cdot \exp\left[-2\pi i \overrightarrow{\mathrm{k}}\,\overrightarrow{\mathrm{r}}\right] = \left\langle \exp\left[-2\pi i \overrightarrow{\mathrm{k}}\,\overrightarrow{\mathrm{r}}\right] \right\rangle \approx$$

$$\approx \tfrac{1}{N} \sum_{j=1}^{N} \exp\left[-2\pi i \overrightarrow{\mathrm{k}}\,\overrightarrow{\mathrm{r}_j}\right] = \tfrac{1}{N} \sum_{j=1}^{N} \left[\cos\left(2\pi \overrightarrow{\mathrm{k}}\,\overrightarrow{\mathrm{r}_j}\right) - i \sin\left(2\pi \overrightarrow{\mathrm{k}}\,\overrightarrow{\mathrm{r}_j}\right)\right]. \tag{15}$$

Если оценить статистические ошибки реальной и мнимой частей этого выражения в предположении истинно случайных чисел, то они оказываются равными

$$\Delta A_{Re} = \Delta A_{Im} = \Delta A = \frac{1}{\sqrt{2N}} \text{ для } \overrightarrow{\mathrm{k}} \neq 0. \tag{16}$$

Для анализа отличия этих амплитуд от нуля удобно вычислять сразу нормированные на стат. ошибку амплитуды:

$$A_N(\vec{\mathrm{k}}) = \sqrt{\frac{2}{N}} \sum_{j=1}^{N} \left[\cos\left(2\pi\, \vec{\mathrm{k}}\, \vec{\mathrm{r}_j}\right) - i \sin\left(2\pi\, \vec{\mathrm{k}}\, \vec{\mathrm{r}_j}\right) \right]. \tag{17}$$

Существенно осложняет применение этого теста то, что амплитуд бесконечно много. С другой стороны, неприятны ненулевые амплитуды как раз при малых значениях k_j. Исследуем более подробно свойства этого теста на генераторе RANDU, плохие свойства которого должны проявиться именно в многомерных корреляциях. Табл. 3 представляет результаты этих расчётов. Ни одна амплитуда по модулю не превысила трёх. Амплитуда $A(0,0)$ не в счёт — она и не должна быть нулевой. Если проанализировать амплитуды с $-10 \le k_j \le +10$, то среди них найдётся несколько штук, превышающих по модулю 3. Максимальное отклонение от нуля в $A_N(6,1) = -3.25 + 0.69i$. Это как-то трудно считать непройденным тестом.

Нетрудно заметить в таблице, что $A^*_N(k_1,k_2) = A_N(-k_1,-k_2)$, как и должно быть в разложении в ряд действительной функции, но это позволяет сократить число анализируемых амплитуд всего в два раза.

Посмотрим, какие результаты получатся в трёхмерном пространств (табл. 4). Сюрприз! Гармоника $\vec{k} = (9,-6,1)$ оказалась больше, чем на 1000 стандартных отклонений отличной от нуля. Посмотрим, как должен быть распределён параметр

$$\mu = k_1 r_1 + k_2 r_2 + k_3 r_3 = 9r_1 - 6r_2 + r_3 \tag{18}$$

для истинно случайных независимых чисел r_1, r_2, r_3, имеющих равномерное распределение в $(0,1)$.

Характеристическая функция этого распределения легко вычисляется:

$$\begin{aligned} \varphi(t) &= \frac{\left(e^{itk_1}-1\right)\left(e^{itk_2}-1\right)\left(e^{itk_3}-1\right)}{(it)^3 k_1 k_2 k_3} = \\ &= \frac{1}{(it)^3 k_1 k_2 k_3} \left(e^{it(k_1+k_2+k_3)} - e^{it(k_1+k_2)} - e^{it(k_1+k_3)} - e^{it(k_2+k_3)} + \right. \\ &\qquad \left. + e^{itk_1} + e^{itk_2} + e^{itk_3} - 1 \right). \end{aligned} \tag{19}$$

Теперь мы можем вычислить функцию распределения вероятностей $F(\mu)$:

$$F(\mu) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \varphi(t) e^{-it\mu}\, \mathrm{d}t. \tag{20}$$

Таблица 3: Амплитуды разложения функции распределения случайных чисел от генератора RANDU в 2-мерном пространстве в ряд Фурье (использовано 10^6 событий).

k_1	k_2	A_N	k_1	k_2	A_N	k_1	k_2	A_N
-5	-5	-0.58-0.72i	-2	-3	-0.10+0.98i	1	-1	0.83+0.36i
-4	-5	-2.29+0.27i	-1	-3	-1.56-1.43i	2	-1	0.59+0.57i
-3	-5	-1.11+0.54i	0	-3	-0.13-0.30i	3	-1	0.02-0.03i
-2	-5	-1.51-0.45i	1	-3	0.51-1.68i	4	-1	-1.53+0.32i
-1	-5	-1.44+0.825i	2	-3	-1.17-1.20i	5	-1	-0.67-0.95i
0	-5	1.06-0.39i	3	-3	-1.28+1.83i	-5	0	0.87+0.44i
1	-5	0.49+1.13i	4	-3	-0.33-0.03i	-4	0	-1.72-0.00i
2	-5	-0.77+0.28i	5	-3	0.04+2.20i	-3	0	-0.97-1.16i
3	-5	0.46-0.36i	-5	-2	-0.64+0.40i	-2	0	0.50+0.63i
4	-5	-0.28+1.68i	-4	-2	0.51-0.23i	-1	0	0.73+0.63i
5	-5	0.63-2.65i	-3	-2	-2.26+0.72i	0	0	1414+0.00i
-5	-4	0.43-0.37i	-2	-2	-0.15-1.55i	1	0	0.73-0.63i
-4	-4	-0.98-1.39i	-1	-2	0.29+0.65i	2	0	0.50-0.63i
-3	-4	-0.20-0.05i	0	-2	-1.35-0.15i	3	0	-0.97+1.16i
-2	-4	-1.83-0.93i	1	-2	0.56-1.22i	4	0	-1.72+0.00i
-1	-4	-0.82-0.05i	2	-2	-0.24+0.09i	5	0	0.87-0.44i
0	-4	0.57-0.03i	3	-2	0.36+0.39i	-5	1	-0.67+0.95i
1	-4	1.68+1.01i	4	-2	-0.33+1.78i	-4	1	-1.53-0.32i
2	-4	-0.84+0.92i	5	-2	-0.19+0.46i	-3	1	0.02+0.03i
3	-4	0.77+0.84i	-5	-1	-1.09-1.17i	-2	1	0.59-0.57i
4	-4	0.94-0.46i	-4	-1	-0.08-1.54i	-1	1	0.83-0.36i
5	-4	0.69-1.01i	-3	-1	-1.00-0.50i	0	1	1.24+0.33i
-5	-3	-2.07-0.84i	-2	-1	-0.55-0.48i	1	1	-0.09+0.17i
-4	-3	1.17-1.33i	-1	-1	-0.09-0.17i	2	1	-0.55+0.48i
-3	-3	-1.11-1.51i	0	-1	1.24-0.33i	3	1	-1.00+0.50i
4	1	-0.08+1.54i	-4	4	0.94+0.46i	5	1	-1.09+1.17i
-3	4	0.77-0.84i	-5	2	-0.19-0.46i	-2	4	-0.84-0.92i
-4	2	-0.33-1.78i	-1	4	1.68-1.01i	-2	2	-0.24-0.09i
1	4	-0.82+0.05i	-3	2	0.36-0.39i	0	4	0.57+0.03i
-1	2	0.56+1.22i	2	4	-1.83+0.93i	0	2	-1.35+0.15i
3	4	-0.20+0.05i	1	2	0.29-0.65i	4	4	-0.98+1.39i
2	2	-0.15+1.55i	5	4	0.43+0.37i	3	2	-2.26-0.72i
-5	5	0.63+2.65i	4	2	0.51+0.23i	-4	5	-0.28-1.68i
5	2	-0.64-0.40i	-3	5	0.46+0.36i	-5	3	0.04-2.20i
-2	5	-0.77-0.28i	-4	3	-0.33+0.03i	-1	5	0.49-1.13i
-3	3	-1.28-1.83i	0	5	1.06+0.39i	-2	3	-1.17+1.20i
1	5	-1.43-0.25i	-1	3	0.51+1.68i	2	5	-1.51+0.45i
0	3	-0.13+0.30i	3	5	-1.11-0.54i	1	3	-1.56+1.43i
4	5	-2.29-0.27i	2	3	-0.10-0.88i	5	5	-0.58+0.72i
3	3	-1.11+1.51i	4	3	1.17+1.33i	5	3	-2.07+0.84i
-5	4	0.69+1.01i						

Таблица 4: Амплитуды разложения функции распределения случайных чисел от генератора RANDU в 3-мерном пространстве в ряд Фурье (использовано 10^6 событий). Приведены только гармоники с $|A_N| > 3$ и $k_1 \geq 0$.

k_1	k_2	k_3	A_N	k_1	k_2	k_3	A_N	k_1	k_2	k_3	A_N
0	0	0	1414.2+0i	5	8	0	-2.92-0.80i	9	-6	1	1414.2+0i
3	-2	7	1.59-2.69i	-6	4	6	1.59-2.69i				

Комплексная функция $\varphi(t)$ аналитическая во всём пространстве, поэтому значение интеграла (20) не изменится, если мы модифицируем путь интегрирования C таким образом, чтобы он проходил ниже точки $t = 0$. Тогда значения восьми слагаемых в этом интеграле легко вычисляются методом вычетов:

$$\int\limits_C e^{it(a-\mu)} \frac{\mathrm{d}t}{t^3} = \begin{cases} 2\pi i \operatorname*{res}\limits_{t=0} \frac{e^{it(a-\mu)}}{t^3}, & \mu < a \\ 0, & \mu > a \end{cases} = -\pi i \left(a-\mu\right)_+^2, \qquad (21)$$

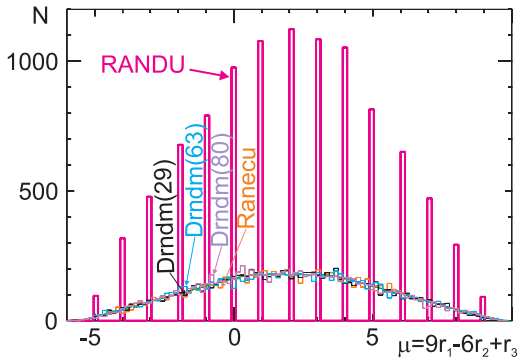

Рис. 4: Гистограммы распределений по параметру $\mu = 9r_1 - 6r_2 + r_3$, где r_1, r_2, r_3 — случайные числа, полученные соответствующим генератором (подпись к гистограмме). Гладкая кривая — теоретическое распределение, справедливое, когда все числа истинно случайные и независимые.

где введено обозначение для функции

$$(x)_+ = \begin{cases} x, & x > 0, \\ 0, & x < 0. \end{cases} \tag{22}$$

В итоге получаем:

$$\begin{aligned} F(\mu) = \tfrac{1}{2k_1k_2k_3} \cdot \Big[& (k_1+k_2+k_3-\mu)_+^2 - (k_1+k_2-\mu)_+^2 - \\ & - (k_1+k_3-\mu)_+^2 - (k_2+k_3-\mu)_+^2 + \\ & + (k_1-\mu)_+^2 + (k_2-\mu)_+^2 + (k_3-\mu)_+^2 - (0-\mu)_+^2 \Big] \end{aligned} \tag{23}$$

Нетрудно убедиться, что полученная функция нормирована на единицу.

На рис. 4 представлен график этой теоретической функции, а также гистограммы распределений по этому параметру, полученные с генераторами RANDU, Drndm(M=29), Drndm(M=63), Drndm(M=80), Ranecu. Что означает это любопытное распределение для генератора RANDU? Это — иллюстрация того, что в трёхмерном пространстве в единичном кубе все вектора, координаты которых образованы последовательными случайными числами генератора RANDU, падают на параллельные эквидистантные плоскости, расстояние между которыми примерно равно $\frac{1}{\sqrt{9^2+6^2+1^2}} = 0.09$. Всего в кубе помещается 15 таких плоскостей. Наиболее очевидное следствие этого дефекта — при интегрировании методом Монте-Карло с помощью генератора RANDU пикованной функции трёх аргументов с размером пика меньше 0.1 систематическая ошибка будет близка к 100%.

Интересно — такого сильного эффекта не было в двумерном пространстве. Может, эффект пропадёт при переходе к бóльшим размерностям? В табл. 5 приведены наибольшие амплитуды разложения функции распределения в 4-мерном пространстве.

Можно было ожидать, что дефектными окажутся амплитуды, где один из коэффициентов k_j нулевой (то есть некоторое 3-мерное подпространство), однако, среди дефектных оказались и истинно 4-мерные амплитуды. Можно ожидать, что и для бóльших размерностей найдутся плохие гармоники (табл. 6). К сожалению, потребление процессорного времени и, что не менее важно, оперативной памяти, экспоненциально растёт с увеличением размерности пространства. Никаких ухищрений по вычислению только нужных амплитуд оказывается недостаточно (заранее неизвестно, какая гармоника окажется «плохой»). Например, в 6-мерном пространстве при $|k_j| \le 10$ приходится анализировать более $4 \cdot 10^7$

Таблица 5: Амплитуды разложения функции распределения случайных чисел от генератора RANDU в 4-мерном пространстве в ряд Фурье (использовано 10^4 событий). Приведены только гармоники с $|A_N| > 4$ и $k_1 \geq 0$.

$\overrightarrow{k}$	A_N	$\overrightarrow{k}$	A_N	$\overrightarrow{k}$	A_N
1,-1,-4,-10	1.78+3.61i	10,-7,-3,-10	1.78+3.61i	6,-3,5,-10	1.30+4.23i
1,8,-10,-9	1.78+3.61i	10,2,-9,-9	1.78+3.61i	6,6,-1,-9	1.30+4.23i
7,2,-2,-5	-0.37+4.03i	8,-8,-9,-4	2.74-3.32i	3,-10,-7,-4	-3.50+1.99i
3,0,-7,-2	3.73-1.96i	0,1,-3,-2	-0.38+4.15i	9,-5,-2,-2	-0.38+4.15i
0,10,-9,-1	-0.38+4.15i	9,4,-8,-1	-0.38+4.15i	0,0,0,0	141.4+0i
0,9,-6,1	141.4+0i	9,3,-5,1	141.4+0i	9,-7,4,2	-0.38-4.15i
3,-2,-10,3	-3.56-1.84i	0,8,-3,3	-0.38-4.15i	9,2,-2,3	-0.38-4.15i
3,6,6,3	4.17-1.29i	6,4,8,4	-3.50-1.99i	1,2,10,4	2.74+3.32i
10,5,5,5	2.74+3.32i	2,1,-3,6	-0.37-4.03i	2,10,-9,7	-0.37-4.03i
8,-5,5,10	1.78-3.61i	1,8,-10,-9	1.78+3.61i	7,-7,4,-6	-0.37+4.03i
3,-9,-1,-3	3.73-1.96i	6,5,5,-2	-3.56+1.84i	9,-6,1,0	141.4+0i
6,-6,8,2	3.72+1.96i	6,3,2,3	3.73+1.96i	2,-8,3,5	-0.37-4.03i
3,-3,-4,10	1.30-4.23i				

Таблица 6: Максимальные амплитуды разложения функции распределения случайных чисел от генератора RANDU в n-мерном пространстве в ряд Фурье.

n	N_{ev}	$\max\|k_j\|$	$\overrightarrow{k}$	A_N
2	10^6	10	6,1	-3.25 + 0.69i
			0,0	1414 + 0i
3	10^6	10	9,-6,1	1414.2 + 0i
			0,0,0	1414.2 + 0i
4	10^4	10	9,3,-5,1	141.4 + 0i
			0,0,0,0	141.4 + 0i
5	10^3	10	9,-6,-8,6,-1	44.7+0i
			0,0,0,0,0	44.7 + 0i
6	10^3	10	9,-6,1,9,-6,1	44.7 +0i
			0,0,0,0,0,0	44.7 +0i
7	10^2	7	3,5,-5,7,7,6,-3	0 + 10i
			0,0,0,0,0,0,0	14.14 + 0i
8	10^2	6	3,2,-1,-3,-2,-5,-4,2	0 + 14.14i
			0,0,0,0,0,0,0,0	14.14 + 0i
9	10^2	5	3,-1,3,5,4,-5,-2,-5,2	0 + 10i
			0,0,0,0,0,0,0,0,0	14.14 + 0i
10	10^2	5[a]	4,3,3,1,-2,-2,4,-1,-5,-5	0 + 14.14i
			0,0,0,0,0,0,0,0,0,0	14.14 + 0i

[a] Не весь набор комбинаций индексов перебран — только $6.6 \cdot 10^9$.

Таблица 7: Максимальные амплитуды разложения функции распределения случайных чисел от генератора Ranecu в n-мерном пространстве в ряд Фурье.

n	N_{ev}	$\max\lvert k_j\rvert$	$\overrightarrow{k}$	A_N
2	10^6	10	9,-6	3.02-0.34i
			0,0	1412.2+i
3	10^6	10	7,10,4	2.14 - 3.47i
			0,0,0	1414.2 + 0i
4	10^4	10	9,-4,4,-2	-5.07 + 0.60i
			0,0,0,0	141.4 + 0i
5	10^3	10	4,-2,-3,3,-3	5.37 - 0.06i
			0,0,0,0,0	44.7 + 0i
6	10^2	10	10,-8,-2,-10,-3,2	-4.94 + 2.94i
			0,0,0,0,0,0	14.14+0i
7	10^2	8	4,-6,2,0,-2,1,3	-1.05 + 5.81i
			0,0,0,0,0,0,0	14.14 + 0i
			8,7,-6,3,-6,2,-7	-5.11 + 2.74i
8	10^2	6	1,1,-3,2,-6,5,-1,-1	-2.25 - 5.72i
			0,0,0,0,0,0,0,0	14.14 +0i
9	10^2	5	5,-1,0,-5,5,-5,-3,0,-5	1.63 + 5.81i
			0,0,0,0,0,0,0,0,0	14.14 + 0i
			3,-3,-5,-5,3,-3,-5,-5,-5	3.93 - 3.76i
10	10^2	4[a]	1,-2,-4,4,4,-1,3,-2,-4,-1	-5.1+4.0i
			0,0,0,0,0,0,0,0,0,0	14.14 + 0i

[a] Не весь набор комбинаций индексов перебран — только $9.3 \cdot 10^8$.

комплексных амплитуд. Поэтому пришлось сокращать интервал значений индексов k_j для больших размерностей пространства. Чем это чревато, видно на примере 5-мерного пространства: если бы мы исследовали амплитуды только с $|k_j| < 9$, то мы не обнаружили бы эту дефектную гармонику.

До сих пор мы изучали результат применения теста на корреляции в многомерном пространстве на примере заведомо плохого генератора. Теперь применим его к рабочим версиям генераторов Ranecu (табл. 7) и Drndm (табл. 8 и 9).

Для генераторов мультипликативного типа известен теоретический результат — в n-мерном пространстве вектора с координатами, образованными из последовательных случайных чисел, лежат на параллельных эквидистантных плоскостях [5]. Теоретический предел N_M для количества плоскостей N_h равен

$$N_h < N_M = \left(n!\, 2^M\right)^{1/n}.$$

Таблица 8: Максимальные амплитуды разложения функции распределения случайных чисел от генератора Drndm ($M = 63$, $K_R = z400040010115$) в n-мерном пространстве в ряд Фурье.

n	N_{ev}	$\max\|k_j\|$	$\overrightarrow{k}$	A_N
2	10^6	10	3,9 0,0	1.64 - 2.97i 1414.2 + 0i
3	10^6	10	8,-8,7 0,0,0	3.84 - 2.53i 1414.2 + 0i
4	10^4	10	7,2,8,-2 0,0,0,0	1.95 + 4.46i 141.4 + 0i
5	10^3	10	6,1,2,4,1 0,0,0,0,0	-5.66 + 0.49i 44.7 + 0i
6	10^2	10	0,5,0,7,-8,4 5,-8,-4,-3,-2,-5 0,0,0,0,0,0	5.24 + 3.03i -1.37 + 5.73i 14.14 + 0i
7	10^2	7	7,-3,-3,3,5,2,6 0,0,0,0,0,0,0	-5.71 + 1.81i 14.14 + 0i
8	10^2	5	5,-1,5,4,-1,-5,-2,3 0,0,0,0,0,0,0,0	4.49 + 3.88i 14.14 + 0i
9	10^2	3	1,2,-1,-1,2,-2,3,-2,1 0,0,0,0,0,0,0,0,0	2.80 + 5.28i 14.14 + 0i
10	10^2	4	3,4,-2,-3,-2,-3,3,-4,1,-1 0,0,0,0,0,0,0,0,0,0	4.43-4.95i 14.14 + 0i

Это достаточно далёкий предел. Например, для $M = 63$, $n = 10$ получаем $N_M = \left(10!\, 2^{63}\right)^{1/10} = 357$. Это количество плоскостей можно пересчитать в минимальное расстояние h_M между плоскостями

$$h_M = \frac{\sqrt{n}}{N_M} = \frac{\sqrt{n}}{\left(n!\, 2^M\right)^{1/n}} \underset{M=63,n=10}{\Longrightarrow} 0.009$$

В свою очередь, это может быть использовано для оценки максимального значения модуля $\left|\overrightarrow{k}\right|$ «плохой» гармоники:

$$\left|\overrightarrow{k}\right| < \frac{1}{h_M} \underset{M=63,n=10}{\Longrightarrow} 113. \tag{24}$$

Другими словами, если бы мы перебирали все амплитуды с $|k_j| < 113$ для генератора DRNDM, то плохая гармоника в 10-мерном пространстве обязательно бы нашлась. Очевидно, что это непосильная задача для полного перебора (число комбинаций порядка $113^{10} = 3.4 \cdot 10^{20}$). В то же

Таблица 9: Максимальные амплитуды разложения функции распределения случайных чисел от генератора Drndm ($M = 80$, $K_R = z400040010115$) в n-мерном пространстве в ряд Фурье.

n	N_{ev}	$\max\|k_j\|$	$\overrightarrow{k}$	A_N
2	10^6	10	3,2	0.10 - 2.97i
			0,0	1414.2+0i
3	10^6	10	5,-4,0	-1.29 + 4.06i
			6,-3,-9	-1.33 + 3.29i
			0,0,0	1414.2 + 0i
4	10^4	10	6,7,-10,8	-4.47 + 1.70i
			0,0,0,0	141.4 + 0i
5	10^3	10	2,-2,1,-10,-3	5.22 + 1.38i
			0,0,0,0,0	44.7 + 0i
6	10^2	10	2,8,,4,-7,5,-7	5.30 + 2.83i
			0,0,0,0,0,0	14.14 + 0i
7	10^2	7	7,-3,3,2,4,5,-6	2.89 - 4.95i
			0,0,0,0,0,0,0	14.14 + 0i
8	10^2	5	3,5,5,-2,-3,1,1,5	-3.44 + 5.22i
			0,0,0,0,0,0,0,0	14.14 + 0i
9	10^2	3	2,-3,0,1,1,3,2,0,1	-2.82 + 4.87i
			2,1,1,2,3,-2,-1,-3,-3	-4.69 - 0.90i
			0,0,0,0,0,0,0,0,0	14.14 + 0i
10	10^2	4	1,-2,-1,1,3,-2,-4,2,-3,-3	-3.31 - 5.32i
			0,0,0,0,0,0,0,0,0,0	14.14 + 0i

время, используя теоретические знания, можно просто пытаться провести параллельный набор плоскостей через набор точек в n-мерном пространстве, полученный с помощью мультипликативного генератора, что и было сделано в работе [4] для генератора DRNDM. Результаты можно свести в табл. 10. В отличие от поиска неизвестной «плохой» гармоники, её проверка в процедуре спектрального анализа проводится быстро. Все гармоники, приведённые в табл. 10, действительно оказались плохими. И как и должно быть, все соседние, отличающиеся в любом индексе k_j хотя бы на единицу, показали нормальное статистическое отклонение от нуля. Например, среди $(3^{10} - 1) = 59048$ амплитуд с индексами, отклоняющимися от указанных в таблице не более, чем на единицу, наибольшее отклонение от нуля получилось 4.4 (стандартных отклонений), в то время как проверяемая «плохая» амплитуда отклонилась от нуля на 45 стандартных отклонений.

Таблица 10: Параметры плохих амплитуд в разложении в ряд Фурье вероятности распределения случайных векторов, полученных с генератором DRNDM в единичном кубе.

n	$\overrightarrow{k}$	n	$\overrightarrow{k}$
3	1002845,-409088, 56635	7	220, 151, 76, 39, -192, 244, -122
4	-12794, 6116, 4837, -10611	8	6, 99, -43, 13, -34, -58, 105, 24
	14952, -3664, -6499, -32161		11, -88, -73, -26, -24, 18, 113, 105
5	1215, 1195, 2928, 90, 504	9	-9, 36, -39, 15, -25, -66, -35, -31, -46
	1898, 1880, 70, 2177, -1141		100, 13, -18, 9, -40, -20, 35, -18, 19
6	71, -222, 350, 232, -306, 15	10	-12, 21, -17, 57, 2, 11, 4, 4, -29, -9

4.7 Интеграл от функции с узким пиком

Посмотрим, как можно обнаружить дефект генератора случайных чисел, сравнивая интеграл от функции с узким пиком, взятый методом Монте-Карло, с известным ответом. Рассмотрим функцию $F\left(\overrightarrow{\mathrm{x}}\right)$ с узким пиком такого вида, что интеграл легко вычисляется

$$\begin{aligned} R = \int\limits_0^1 \mathrm{d}x_1 \cdots \int\limits_0^1 \mathrm{d}x_n \; F\left(\overrightarrow{\mathrm{x}}\right) = A \cdot \prod_{k=1}^{n} \int\limits_0^1 \frac{\mathrm{d}x_k}{\beta^2 + (x_k - p_k)^2} = \\ = \frac{A}{\beta^n} \prod_{k=1}^{n} \operatorname{arcctg}\left(\beta - \frac{(1-p_k)p_k}{\beta}\right), \end{aligned} \tag{25}$$

где $\overrightarrow{\mathrm{p}}$ — точка в n-мерном кубе, $0 < p_k < 1$. Если выбрать константу

$$A = \prod_{k=1}^{n} \frac{\beta}{\operatorname{arcctg}\left(\beta - \frac{(1-p_k)p_k}{\beta}\right)}, \tag{26}$$

то значение интеграла должно быть равно единице. Если вычислять интеграл методом Монте-Карло

$$R = \frac{A}{N} \sum_{j=1}^{N} \left(\prod_{k=1}^{n} \frac{1}{\beta^2 + (x_{j,k} - p_k)^2} \right), \tag{27}$$

то статистически значимое отклонение этого значения от 1 может свидетельствовать о дефекте генератора случайных чисел. Дисперсия может

быть вычислена аналитически:

$$\begin{aligned} D(R) &= \frac{A^2}{N}\prod_{k=1}^{n}\left[\frac{x-p_k}{2\beta^2\left[(x-p_k)^2+\beta^2\right]}-\frac{1}{2\beta^3}\operatorname{arcctg}\frac{x-p_k}{\beta}\right]_{x=0}^{x=1}-\frac{1}{N}= \\ &= \frac{1}{N}\left\{\prod_{k=1}^{n}\frac{1}{\operatorname{arcctg}^2\left(\beta-\frac{(1-p_k)p_k}{\beta}\right)}\left[\frac{1-p_k}{2\left[(1-p_k)^2+\beta^2\right]}+\frac{p_k}{2\left[p_k^2+\beta^2\right]}+\right.\right. \\ &\left.\left.+\frac{1}{2\beta}\operatorname{arcctg}\left(\beta-\frac{p_k(1-p_k)}{\beta}\right)\right]-1\right\} \end{aligned} \tag{28}$$

В пределе $\beta \to \infty$ выражение превращается в

$$\begin{aligned} D(R) &\underset{\beta\to\infty}{\approx} \frac{1}{N}\left\{\prod_{k=1}^{n}\left[1+\frac{4-15p_k(1-p_k)}{45\operatorname{tg}^2\beta}\right]-1\right\}\approx \\ &\approx \frac{1}{N}\sum_{k=1}^{n}\frac{4-15p_k(1-p_k)}{45\operatorname{tg}^2\beta}. \end{aligned} \tag{29}$$

В обратном случае $\beta \to 0$, который нас больше интересует, получаем

$$D(R) \underset{\beta\to 0}{\approx} \frac{1}{(2\pi\beta)^n N} \tag{30}$$

Для надёжного обнаружения дефекта нам нужно $D(R) \ll 1$, или $N \gg \frac{1}{(2\pi\beta)^n}$. Для генератора RANDU в трёхмерном пространстве критический размер (расстояние между плоскостями) равен $\beta \sim 0.09$ и $N \gg \left(\frac{1}{0.58}\right)^3 = 5.2$. Для этого же генератора в 10-мерном пространстве необходима статистика $N \gg \left(\frac{1}{2\pi 0.1}\right)^{10} = 168$. Видно, что для этого генератора такой тест оказался бы критическим при вполне доступной статистике. Для генератора DRNDM в 10-мерном пространстве статистика потребуется гораздо больше: $N \gg \left(\frac{1}{2\pi 0.014}\right)^{10} = 4{\cdot}10^{10}$, что уже находится на грани возможного. Посмотрим, как этот тест выглядит на практике. Табл. 11 показывает результаты теста в 3-мерном пространстве.

В таблицу введены также данные по генератору RANLUX, который имеет период ряда 10^{165}, что для генератора Drndm соответствует $M = 550$. По-видимому, для этого генератора дефект при интегрировании функции с узким пиком не должен наблюдаться. Интересно, что для этого генератора особенностью расчёта является повышенная оценка ошибки. Во всех расчётах дисперсия оценивалась по самому распределению, что не очень хорошо при больших дисперсиях. Посмотрим, какую оценку даёт формула (30):

β	0.1	10^{-2}	10^{-3}	10^{-4}	10^{-5}
n	3	3	3	3	3
N	10^6	10^6	10^7	10^8	10^9
σ_R	$2 \cdot 10^{-3}$	0.063	0.63	6.3	63.5

Таблица 11: Результаты интегрирования функции трёх переменных с узким пиком методом Монте-Карло с разными генераторами случайных чисел. $p_k = 0.3$.

Ген-тор	N_{ev}	β	R	$\frac{R-1}{\sigma_R}$
RANDU	10^6	1.0	1.00004 ± 0.00019	0.23
	10^6	0.2	0.9993 ± 0.0014	-0.47
	10^6	0.1	1.0063 ± 0.0031	2.05
	10^6	0.02	1.057 ± 0.025	2.34
	10^6	$\frac{1}{200}$	0.650 ± 0.044	-7.9
RANECU	10^6	10^{-1}	1.0048 ± 0.0031	1.57
	10^6	10^{-2}	0.947 ± 0.056	-0.96
	10^7	10^{-3}	1.54 ± 0.66	0.82
	10^8	10^{-4}	0.131 ± 0.052	-16.9
	10^9	10^{-5}	0.0084 ± 0.0039	-257
RANLUX	10^6	10^{-1}	1.0010 ± 0.0030	0.31
	10^6	10^{-2}	0.956 ± 0.056	-0.78
	10^7	10^{-3}	1.66 ± 0.78	0.85
	10^9	10^{-4}	11.3 ± 7.8	1.3
	10^{10}	10^{-5}	0.042 ± 0.032	-29.5
Drndm (M=32)	10^6	10^{-1}	0.9940 ± 0.0030	-1.98
	10^6	10^{-2}	1.095 ± 0.070	1.35
	10^7	10^{-3}	0.89 ± 0.42	-0.26
	10^8	10^{-4}	0.039 ± 0.011	-84.9
	10^9	10^{-5}	$\frac{1.28 \pm 0.26}{10^3}$	-3886
Drndm (M=63)	10^6	10^{-1}	0.9966 ± 0.0030	-1.13
	10^6	10^{-2}	1.060 ± 0.072	0.83
	10^7	10^{-3}	0.43 ± 0.13	-4.4
	10^8	10^{-4}	0.105 ± 0.028	-31.5
	10^9	10^{-5}	0.0041 ± 0.0015	-676
Drndm (M=80)	10^6	10^{-1}	0.9971 ± 0.0030	-0.96
	10^6	10^{-2}	1.001 ± 0.065	0.016
	10^7	10^{-3}	1.73 ± 0.84	0.88
	10^8	10^{-4}	1.43 ± 0.65	0.65
	10^9	10^{-5}	0.0032 ± 0.0017	-570

Видно, что при очень узких пиках функции (малое β) статистическая точность оценки интеграла становится неудовлетворительной. При увеличении размерности пространства трудности будут только увеличиваться.

4.8 Тест «случайные блуждания»

В английской литературе этот тест называется “Random walk” [6]. Фактически, с помощью исследуемого генератора моделируется дискретное распределение вероятностей величины r и сравнивается с истинным распределением. Параметром распределения является константа $\alpha \in (0, 1)$. Алгоритм генерации:

1. $r = 0$.
2. Выбирается очередное случайное число ξ.
3. Если $\xi > \alpha$, то цикл генерации прекращается. Текущее значение r является новым значением случайной переменной, которую мы моделируем.
4. Если же $\xi < \alpha$, то в r добавляется единица и делается переход на пункт 2.

Возможные значения переменной r — целые числа от 0 до бесконечности. Распределение вероятностей легко вычисляется:

$$W_r = \alpha^r \cdot (1 - \alpha), \quad r = 0, 1, 2, 3, \ldots \tag{31}$$

Если использовать гистограмму распределения по r с полным числом событий N, то удобно ввести меру отклонения R фактического распределения от точного теоретического:

$$R(r) = \frac{n_r}{N \cdot W_r} - 1, \quad \sigma_R = \sqrt{\frac{1 - W_r}{N \cdot W_r}} \tag{32}$$

Возьмем $\alpha = \frac{31}{32}$ и посмотрим, какие будут значения $R(r)$ для $0 < r < 64$ для разных генераторов (рис. 5, 6, 7, 8, 9, 10).

Как видно, по результатам этого теста генераторы Ranecu и DRNDM (M=32, 63,80) выглядят достойно. Некоторые сомнения вызывает генератор Ranlux, а генератор RANDU однозначно не проходит этот тест.

Изменим теперь параметр $\alpha = \frac{63}{64}$ и посмотрим, какие будут значения $R(r)$ для $0 < r < 128$ для разных генераторов (рис. 11, 12, 13, 14, 15, 16).

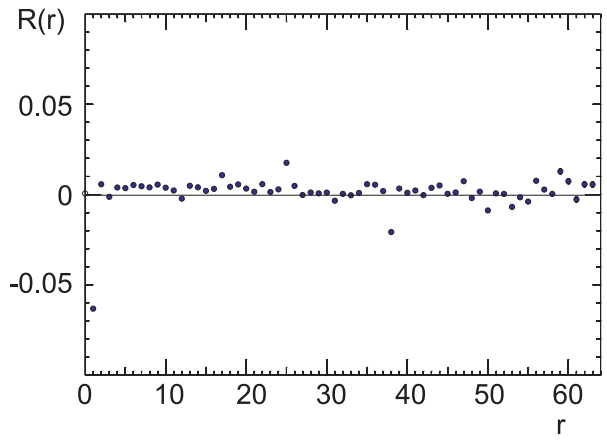

Рис. 5: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 31/32$. Генератор случайных чисел RANDU. $\chi^2/n_D = 14781.8/64$, 10^8 событий.

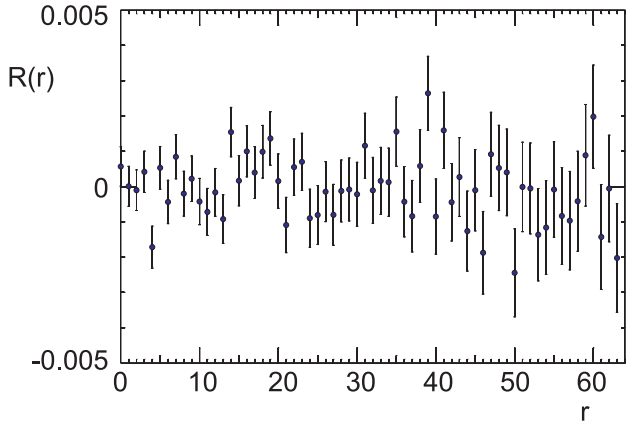

Рис. 6: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 31/32$. Генератор случайных чисел RANECU. $\chi^2/n_D = 66.1/64$, 10^8 событий.

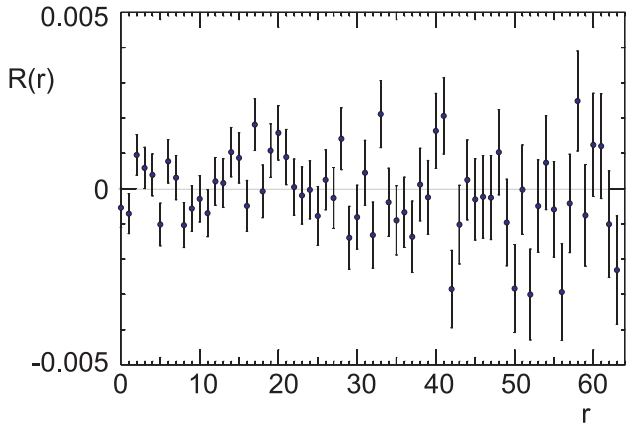

Рис. 7: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 31/32$. Генератор случайных чисел RANLUX. $\chi^2/n_D = 90.0/64$, 10^8 событий.

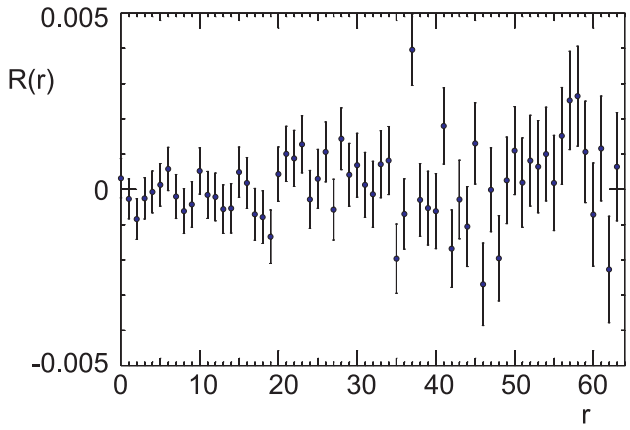

Рис. 8: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 31/32$. Генератор случайных чисел DRNDM, $M = 32$. $\chi^2/n_D = 75.3/64$, 10^8 событий.

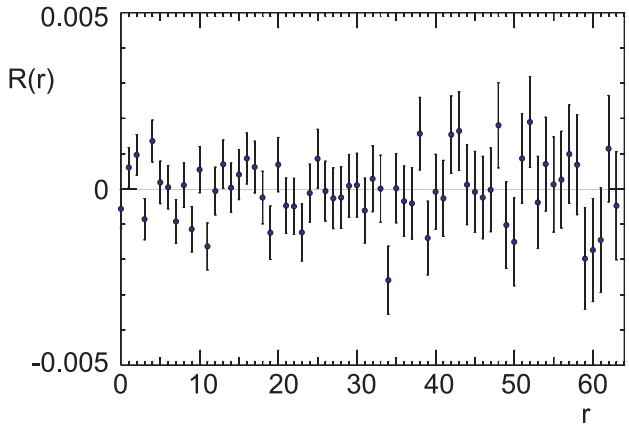

Рис. 9: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 31/32$. Генератор случайных чисел DRNDM, $M = 63$. $\chi^2/n_D = 65.5/64$, 10^8 событий.

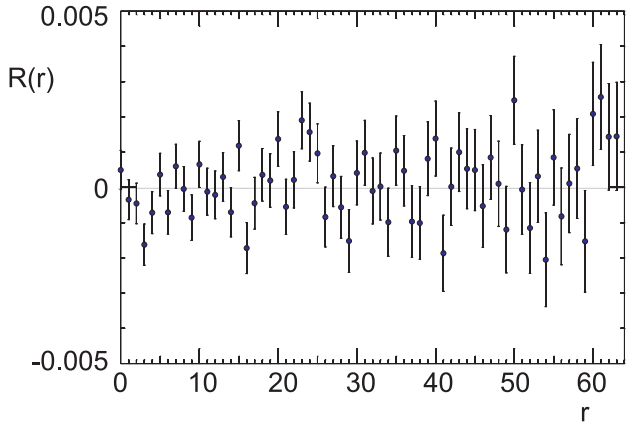

Рис. 10: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 31/32$. Генератор случайных чисел DRNDM, $M = 80$. $\chi^2/n_D = 75.3/64$, 10^8 событий.

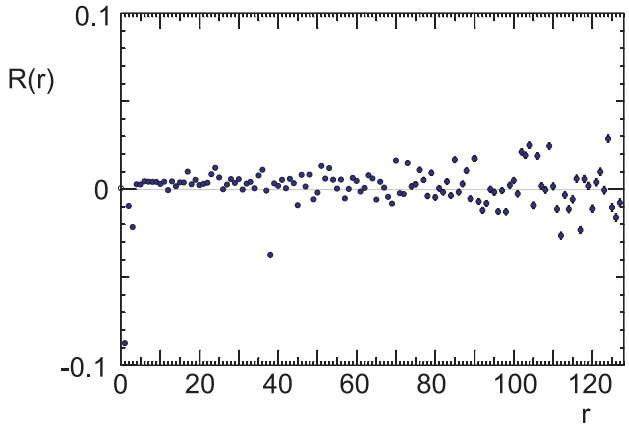

Рис. 11: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 63/64$. Генератор случайных чисел RANDU. $\chi^2/n_D = 18135.7/128$, 10^8 событий.

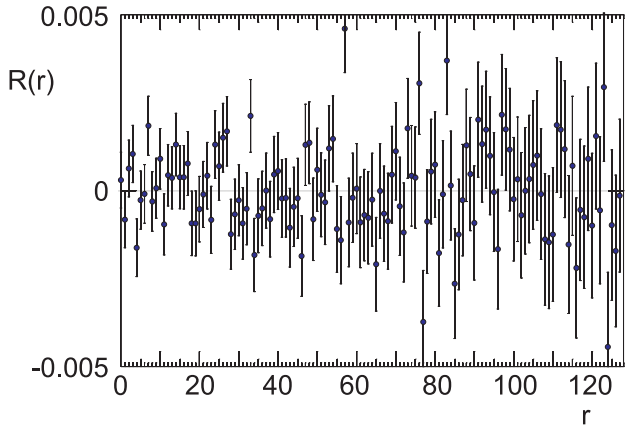

Рис. 12: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 63/64$. Генератор случайных чисел RANECU. $\chi^2/n_D = 121.0/128$, 10^8 событий.

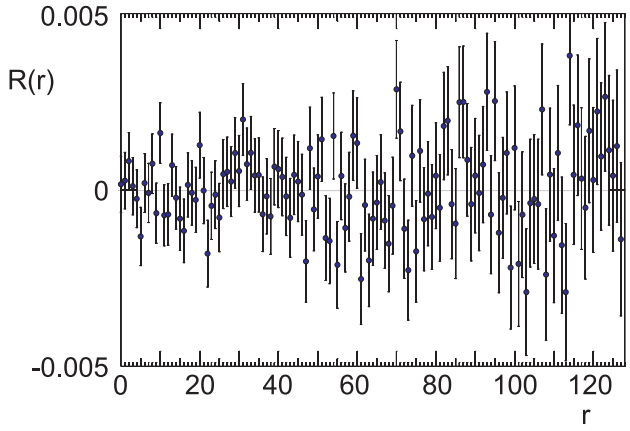

Рис. 13: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 63/64$. Генератор случайных чисел RANLUX. $\chi^2/n_D = 105.0/128$, 10^8 событий.

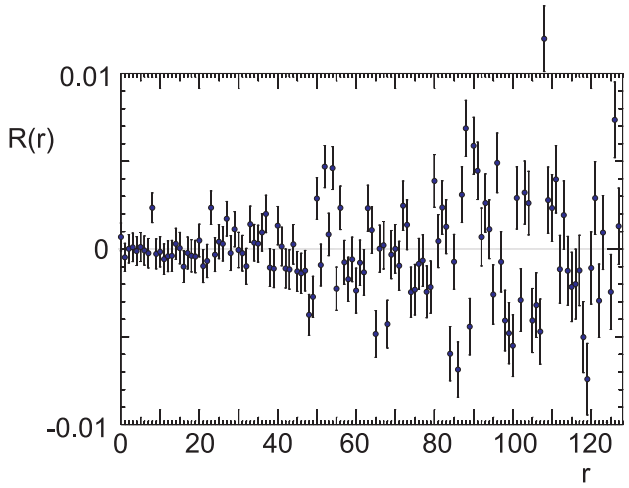

Рис. 14: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 63/64$. Генератор случайных чисел DRNDM, $M = 32$. $\chi^2/n_D = 429.1/128$, 10^8 событий.

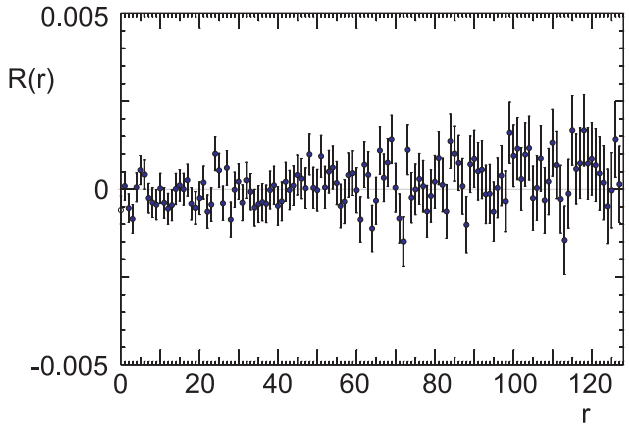

Рис. 15: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 63/64$. Генератор случайных чисел DRNDM, $M = 63$. $\chi^2/n_D = 111.5/128$, 10^8 событий.

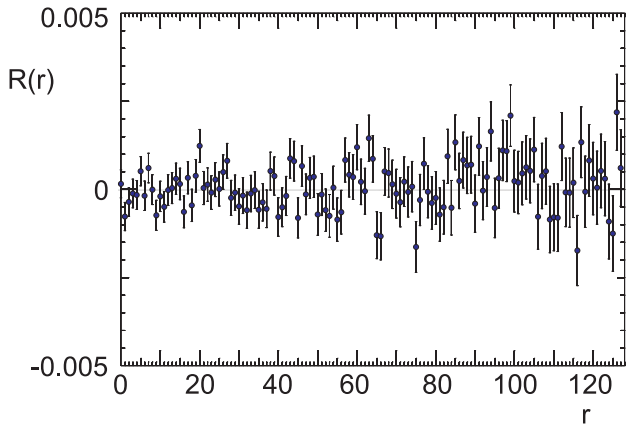

Рис. 16: Отклонение $R(r)$ генерированного по r распределения от теоретического, $\alpha = 63/64$. Генератор случайных чисел DRNDM, $M = 80$. $\chi^2/n_D = 128.6/128$, 10^8 событий.

Как видно, генератор DRNDM при $M = 63, 80$ имеет, наряду с генераторами Ranecu и RANLUX, хорошие результаты прохождения теста «Random walk».

5 Заключение

Разработан генератор случайных чисел RndDrndmEngine на языке C^{++}, по интерфейсу совместимый с генератором RndRanecuEngine и пригодный для использования в программе моделирования детектора СНД в качестве альтернативного.

Статистические свойства этого генератора определяются параметром M — количество бит в целочисленной арифметике. Начиная с $M = 63$, его статистические свойства не хуже, чем у генератора Ranecu. Статистические свойства такого генератора могут быть значительно ухудшены неудачным выбором производящей константы ряда, которая может задаваться по умолчанию, но может быть и введена с помощью функции setGenerConst.

Ухудшить период ряда (уменьшить) неудачной константой невозможно – при вводе производится контроль, чтобы новая константа ряда равнялась 3 или 5 по модулю 8. Для этих констант период равен 2^{M-2}.

Правильность работы генератора при нескольких значениях $M = 16, 32, 63, 80$ проверялась на нескольких статистических тестах. Отсутствие опечаток в программе проверялось, во-первых, по полному совпадению ряда случайных чисел с генератором DRNDM ($M = 63$) на Фортране, который был тщательно изучен ранее и использовался при моделировании СНД программой UNIMOD, во-вторых, непосредственной проверкой всего цикла получения одного очередного случайного числа при $M = 80$.

В принципе, оба генератора можно использовать в GEANT4 для моделирования СНД. Так как любой расчет методом Монте-Карло является тестом статистических свойств генератора, то в некоторых особо важных случаях можно провести расчет дважды с использованием разных генераторов. Если результаты статистически совместны, то в дальнейшем можно выборки событий объединять. Если же результаты статистически несовместимы, то необходимы дальнейшие исследования. Кстати, разработанный генератор можно рассматривать, как набор большого числа разных генераторов, так как при изменении производящей константы ряда свойства генератора существенно меняются.

Автор благодарен Букину Д.А. за полезные обсуждения и помощь в работе.

A Исходные тексты класса RndDrndmEngine

```
#ifndef _RNDUTIL_RNDDRNDMENGINE_H
#define _RNDUTIL_RNDDRNDMENGINE_H 1
/**
 * @file RndUtil/RndDrndmEngine.h
 *
 * Declaration of class RndDrndmEngine
 * and related functions
 *
 * @author Alexander D. Bukin
 * @date   Jan 29, 2007
 *
 * Copyright (C) BINP, 2007
 *
 * $Id: RndDrndmEngine.h,v 1.1 2007/02/26
 * 09:04:59 bukin Exp $
 */

// iostream forward declarations
#include <iosfwd>
#include <string>
#include <vector>
#include <cassert>

// Base classes declarations

#include "RndUtil/RndVEngine.h"

/**
 * Example class.
 *
 * Full description...
 */
class RndDrndmEngine: public RndVEngine
{
 public:
   explicit RndDrndmEngine( size_t M = 63 );
```

```
/// Destructor
virtual ~RndDrndmEngine();

/// Output method
virtual std::ostream&
  print(std::ostream& s) const;

/// set Random Seed
bool setRandomSeed
       ( const std::string& seed );

/// set Generation constant
bool setGenerConst( const std::string& genconst );

typedef enum {
    DEC = 10,
    BIN = 2,
    HEX = 16
} SeedType;

/// get Current random integer number
std::string getCurrentSeed( SeedType = DEC ) const;

/// skip specified number of random numbers
void skip( size_t n1, size_t n2, size_t n3);

/// Output of next random number in (0,1)
virtual double flat();

/// Fills an array "vect" of specified size
//       with flat random values.
virtual void flatArray( const HepInt size,
                        HepDouble* vect );

// Should initialise the status
//  of the algorithm according to seed.
virtual void setSeed( long, HepInt ) {
  std::cout <<
   "RndDrndmEngine: setSeed function not implemented "
```

```
      << std::endl;
    ::abort();
  }

  // Should initialise the status of the algorithm according
  // to the zero terminated array of seeds.
  // It is allowed to ignore one or many seeds in this array.
  virtual void setSeeds( const long*, HepInt );

  // --- extension of the interface

  /// get the number of randoms per single skip.
  /// The values allowed here are 1 -- 10^9
  virtual size_t randomsPerSkip() const {
    return 1;
  }

  /// get the state of the engine as a new vector of longs,
  /// suitable for setSeeds().  Must be deleted []
  virtual long* getCurrentSeeds() const ;

protected:
  /// Auxilliary function for n3 = n1 * n2 mod 2**M
  //       integer arithmetics
  void multiply( const std::vector< size_t >& n1,
         const std::vector< size_t >& n2,
         std::vector< size_t >& n3 );

private:

  /// moved to private, DONT USE IT
  virtual void saveStatus( const char* )
  const {
    ::abort();
  }
  /// moved to private, DONT USE IT
  virtual void restoreStatus( const char* )
  {
    ::abort();
  }
```

```
   /// moved to private, DONT USE IT
   virtual void showStatus() const {
     ::abort();
   }

   /// Assigment operator,
   //  avoid default implementation
   RndDrndmEngine& operator=
           (const RndDrndmEngine&);

   /// Copy constructor,
   //     avoid default implementation
   RndDrndmEngine(const RndDrndmEngine&);

  private:
   size_t M;  // Number of bits in
              // integer arithmetics
   size_t Mb; // Number of words in
   // long integer numbers Mb=(M+15)/16
   std::vector<size_t> Kr; // Generation
                           // constant
   std::vector<size_t> ki; // current
    //  random integer number in (0,2**M)
   mutable std::vector<size_t> kr1;
   // working field - 1
   mutable std::vector<size_t> kr2;
   // working field - 2
   mutable std::vector<size_t> kr3;
   // working field - 3

};

#endif // _RNDUTIL_RNDDRNDMENGINE_H

// Local Variables: ***
// mode: c++ ***
// End: ***
```

RndDrndmEngine.cc

```
/**
 * @file RndUtil/RndDrndmEngine.cc
 *
 * Implementation of class RndDrndmEngine.
 *
 * @see RndUtil/RndDrndmEngine.h
 * @author Alexander D. Bukin
 * @date   Jan 29, 2007
 * Copyright (C) BINP, 2007
 *
 * $Id: RndDrndmEngine.cc,v 1.1 2007/02/26
 *  09:04:58 bukin Exp $
 */

// ----- includes ------

// standard C headers
#include <cstdlib>

// standard C++ headers
#include <iostream>
#include <algorithm>

// Experiment.h must be in any implementation file
#include "Experiment/Experiment.h"

// this class header
//#include "RndUtil/RndDrndmEngine.h"
#include "RndDrndmEngine.h"

// other packages headers

// ----- methods definition ------


namespace {

bool getNumber( const std::string& string,
```

```
        const size_t M,
        const size_t Mb,
        std::vector< size_t >& ki )
{
  // preparation
  // remove spaces
  std::string seed( string.size(),' ');
  std::remove_copy( string.begin(),
    string.end(), seed.begin(), ' ');
  // to lower case
  std::transform( seed.begin(), seed.end(),
                  seed.begin(), ::tolower );

  assert( ! seed.empty() );

  size_t Base = 10;
  size_t i = 0;
  if ( seed[0] == 'z' ) {
    // hexadecimal system
    Base = 16;
    ++i;
  } else if ( seed[0] == 'b' ) {
    Base = 2;
    ++i;
  }

  const std::string
         Figs = "0123456789abcdef";
  for ( size_t j=0; j<Mb; j++) ki[j] = 0;

  for ( ; i < seed.size(); ++i ) {
    // loop on input symbols

    const size_t
           found = Figs.find( seed[i] );
    if ( found == std::string::npos ) {
      // not found
      return false;
    }
```

```
    if ( found >= Base ) {
      // wrong symbol
      return false;
    }

    size_t m = found;
    for( size_t j=0; j<Mb; j++) {
      // taking into account this figure
      size_t k = ki[j] * Base + m;
      m = k / 65536;
      k = k - m * 65536;
      ki[j] = k;
    }

    if(m > 0) {
      // extra bits !
      return false;
    }

  } // loop over seed chars

  size_t k = ki[Mb-1]; // Check the highest digit:
  if((k >> (M - Mb * 16 + 16)) > 0)
    {// Non zero extra bits in highest digit!
      return false;
    }
  else
    {// Okay !
      k = ki[0]; // Lowest digit should be odd!
      size_t m = k / 2;
      k = k - m * 2;
      if(k == 0)
    {// Even number !
      return false;
    }
    }
  return true;
}

} // namespace
```

```
// default constructor
/*
 */

// constructor with specified M
RndDrndmEngine::RndDrndmEngine( size_t Mout ) :
RndVEngine( RndEngineType( RndEngineType::Drndm ) ),
M(Mout),
Mb( (Mout+15)/16 ),
Kr(Mb,0),
ki(Mb,0),
kr1(Mb,0),
kr2(Mb,0),
kr3(Mb,0)
{
  if(Mout < 8 || Mout >1000 )
    {// abort
      std::cerr <<
      "Invalid M for RndDrndmEngine Mout=" << Mout <<
      "(see file " __FILE__ << " )" << std::endl;
      ::abort();
    }
  ki[0] = 1;
  size_t k = M / 4;
  size_t i = 0;
  while( k >= 16 )
    {
      k = k - 16; i++;
    }
  ki[i] = ki[i] | (1 << k);
  if(M > 32)
    {// Generation constant from DRNDM
      Kr[0] = 277; Kr[1] = 16385;
      if(Mb > 2)
    {// More than 2 words
      Kr[2] = 16384;
      if(Mb > 3)
        {// More than 3 words
```

```
          Kr[3] = 0;
          if(M > 63)
        {// More bits than in DRNDM
          Kr[3] = 32768;
          for (size_t i=4; i<Mb; i++)
            {
              Kr[i] = 0x8888;
            }
          int Mx = M / 3; int i = Mb - 1;
          while (Mx > 0){// set to zero extra bits
            if(Mx >= 16){// Current word should be zero
              Kr[i] = 0;
            }
            else {
              Kr[i] = Kr[i] & ( 0xFFFF >> Mx);
            }
            i = i - 1; Mx = Mx - 16;
          }
        }
        }
    }
    }
  else
    {// Generation constant of RNDM from CERN
      Kr[0] = 3533;
      if(Mb > 1)
    {// More than 1 word
      Kr[1] = 1;
    }
    }
  size_t kw = 0xFFFF;
  i = Mb * 16 - M; //  Number of unused bits
  kw = kw >> i;
  Kr[Mb-1] = Kr[Mb-1] & kw;  // Mask used to
                          // reset higher bits
}

// destructor
RndDrndmEngine::~RndDrndmEngine()
{
```

```
}

// just an example -- please, override
std::ostream& RndDrndmEngine::print
            (std::ostream& s) const
{
   return s <<
    static_cast<const void*>(this) <<
    "(file " __FILE__ " )";
}

// Multiplication of long integer numbers
void RndDrndmEngine::multiply(
    const std::vector<size_t>& n1,
    const std::vector<size_t>& n2,
    std::vector<size_t>& n3 )
{
  size_t nw, ns, nr1, nr2;
  for ( size_t i=0; i<Mb; i++)
    {
      n3[i] = 0;
    }
  ns = 0; // Shifted from lower part
  for ( size_t k=0; k<Mb; k++)
    {// Main loop on digits of n3
      for ( size_t i=0; i<= k; i++)
    {// Product of two digits of operands
      ns = ns + n1[i] * n2[k - i];
      if(ns > 65536 || i == k)
        {// Necessary to distribute
                  // accumulated sum
          size_t m = k;
          while (ns > 0 && m < Mb)
        {//
          nr1 = (ns + n3[m]) & 0xFFFF;
          nr2 = (ns + n3[m]) >> 16;
          n3[m] = nr1;
          ns = nr2;
          m = m + 1;
        }
```

```
          ns = 0;
        }
    }
    }// All digits of n3 are defined
  nw = 0xFFFF >> (Mb * 16 - M);
        // Mask for highest digit
  n3[Mb-1] = n3[Mb-1] & nw;
}

// Output of next random number
double RndDrndmEngine::flat()
{// First generation of next integer
                   // random number
  for ( size_t i=0; i<Mb; i++)
    {
      kr1[i] = ki[i];
    }
  multiply(Kr, kr1, ki);
  double Ri;
  int knz = 0; size_t  nw;
  size_t i = 0;
  double Rn = 0.;
  nw = 1 << (M - Mb * 16 + 16);
  double Rb = static_cast<double>( nw );
  while(i<Mb && knz < 4)
    {// Taking into account one digit after another
      nw = ki[Mb - i - 1];
      if(nw > 0 || knz > 0) knz++;
      Ri = static_cast<double>( nw );
      Rn = Rn + Ri / Rb;
      Rb = Rb * 65536.;
      i++;
    }
  return Rn;
}

void RndDrndmEngine::flatArray( const HepInt size,
                                HepDouble* vect )
{
  for ( size_t i = 0; i < size_t(size); ++i ) {
```

```
    vect[i] = flat();
  }
}

// Skip the specified number of
     //   random numbers N=n1*n2*n3
void RndDrndmEngine::skip( size_t n1,
                 size_t n2, size_t n3)
{//
  std::vector< size_t > kIn(3,1);
  if(n1 == 0 || n2 == 0 || n3 == 0 ) return;
  kIn[0] = n1;
  kIn[1] = n2;
  kIn[2] = n3;
  for ( size_t i=0; i< Mb; i++ )
    {
      kr1[i] = 0;
      kr2[i] = Kr[i];
    }
  kr1[0] = 1;
  int nr1;
  for( size_t i=0; i<3; i++)
    {// Loop on input numbers
      nr1 = kIn[i];
      while (nr1 > 0)
    {// Loop on kIn[i]
      if((nr1 & 1) == 1)
        {// This power of Kr should be included
          multiply(kr1, kr2, kr3);
          for (size_t j=0; j < Mb; j++)
                       kr1[j] = kr3[j];
        }
      multiply(kr2,kr2,kr3); // Squared k**n
      for (size_t j=0; j<Mb; j++)
                       kr2[j] = kr3[j];
      nr1 = nr1 >> 1;
    }
      for( size_t j=0; j<Mb; j++)
    {// New base
      kr2[j] = kr1[j]; kr1[j] = 0;
```

```
    }
      kr1[0] = 1;
    }
  // Now kr1 = Kr**(n1*n2*n3)
  multiply(ki,kr2,kr3);
  for ( size_t j=0; j<Mb; j++)
                      ki[j] = kr3[j];
}

/// set Random Seed
bool RndDrndmEngine::setRandomSeed
          (const std::string& string )
{// Set specified Random seed
  bool res = ::getNumber( string,M,Mb,ki);
  if(res) {// OKAY !
    size_t k = ki[0];
    // The number should be odd!
    size_t m = k/2;
    if(m *2 != k){// it is odd
      return true;
    }
  }
  return false;
}

/// set Generation constant
bool RndDrndmEngine::setGenerConst
      ( const std::string& string )
{
  bool res = ::getNumber( string, M, Mb, Kr );
  if ( res ) {
    // Okay !
    size_t k = Kr[0];
    // Check if lowest digit = 3 or 5 mod 8
    size_t m = k / 8;
    k = k - m * 8;
    if ( k == 3 || k == 5 )
      {// Okay !
    return true;
      }
```

```
  }
  return false;
}


/// get Current random integer number
std::string RndDrndmEngine::getCurrentSeed(
            SeedType atype ) const
{
  assert( atype == DEC ||
        atype == BIN || atype == HEX );
  const size_t Base =
       static_cast< size_t >( atype );
  std::ostringstream string;
  const char* Figs = "0123456789ABCDEF";
  kr1 = ki;
  size_t knz = Mb - 1;
  // remove the most-significant zeros
  while (knz > 0 && kr1[knz] == 0) knz = knz - 1;
  while (knz > 0 || kr1[0] > 0) {
    // Extraction of successive digits
      // starting from the lowest ones
    size_t m = 0; // Transfer from higher digits
    int j = (int) knz;
    if(j>0 && kr1[j] == 0) knz = knz - 1;
    while( j >= 0 )
      {// Division by Base
    size_t n1 = kr1[j] + m * 65536;
    size_t n2 = n1 / Base;
    size_t n3 = n1 - n2 * Base;
    kr1[j] = n2;
    m = n3;
    j = j - 1;
      }
    // m equals residue of division by Base
    string << Figs[m];
    }
  if ( Base == 16 ) {
    string << "Z";
  } else if ( Base == 2 ) {
```

```
    string << "B";
  }

  // reverse
  const std::string& seed = string.str();
  std::string res( seed.size(), ' ' );
  std::copy( seed.rbegin(),
             seed.rend(), res.begin() );
  return res;
}
// --> Set seed from long array <-----<<<
void RndDrndmEngine::setSeeds
         ( const long* seeds, HepInt )
{
  for (size_t i=0; i<Mb; i++){
    ki[i] = seeds[i];
  }
  // Check for being odd
  if( ki[0] / 2 * 2 == ki[0]){
    std::cout <<
    "RndDrndmEngine::setSeeds : invalid seed"
      << std::endl
      << " Initial random number must be odd !"
      << std::endl
      << "Entered seed =";
    for(size_t i=0; i<Mb; i++)
        std::cout << " " << seeds[i] ;
    std::cout << std::endl;
    ::abort();
  }
}
// --> get Current Seeds as long array < --- <<
long* RndDrndmEngine::getCurrentSeeds() const
{
  long* rpointer = new long [Mb];
  for(size_t i=0; i<Mb; i++){
    rpointer[i] = ki[i];
  }
  return rpointer;
}
```

Список литературы

[1] *С.И. Середняков.* VEPP-2000 Project: Collider, Detectors, and Physics Program. Physics of Atomic Nuclei, Vol. 67, №3, 2004, pp. 482 – 492. Translated from «Ядерная физика», том 67, №3, 2004, стр. 501 – 511.

[2] *S. Agostinelli, J. Allison, K, Amako et al.* GEANT4: A Simulation Toolkit.Nuclear Instuments and Methods A506 (2003) 250 – 303.

[3] *P. L'Ecuyer.* Efficient and Portable Random Number Generators. Comm. ACM 31 (1988) 742.

[4] *А.Д. Букин.* Корреляции псевдослучайных чисел мультипликативной последовательности. Препринт ИЯФ 89-86, Новосибирск, 1989.

[5] *G. Marsaglia.* Random numbers fall mainly in the planes. Proc. Nat. Acad. Sci. 61 (1968) 25 – 28.

[6] *L.N.Shchur, J.R.Herringa, and H.W.J.Blöte.* Phisica A, 241, 579 (1997). *L.N.Shchur and Paolo Butera.* The RANLUX generator: resonances in a random walk test. Mod. Phys. C: 9 (1998), pp. 607–624. e-Print: hep-lat/9805017.

Содержание

А.Д. Букин

Разработка альтернативного генератора случайных чисел для моделирования детектора СНД

A.D. Bukin

Development of random number generator for simulation of the detector SND

ИЯФ 2007-21

Ответственный за выпуск А.М. Кудрявцев
Работа поступила 4.07.2007 г.

Сдано в набор 5.07.2007 г.
Подписано в печать 6.07.2007 г.
Формат бумаги 60×90 1/16 Объем 3.2 печ.л., 2.6 уч.-изд.л.
Тираж 100 экз. Бесплатно. Заказ № 21

Обработано на IBM PC и отпечатано на
ротапринте ИЯФ им. Г.И. Будкера СО РАН
Новосибирск, 630090, пр. академика Лаврентьева, 11.