Корреляция разных реализаций

Плохой выбор движка может добавить искусственную корреляцию при генерации разных реализаций, фактически это значит что последовательности случайных чисел соответствующие разным зернам (1, 2, 3, …) коррелируют, хотя не должны. Это свойственно как правило старым движкам рандома (главным образом LCG), в большинстве сред это не должно быть сильной проблемой.

Ниже я составил небольшую таблицу с листингом дефолтных движков рандома в разных языках/средах, используемых в вычислительной практике.

This trend can be observed rather clearly by looking at the lineup of default PRNGs used in different programming languages:

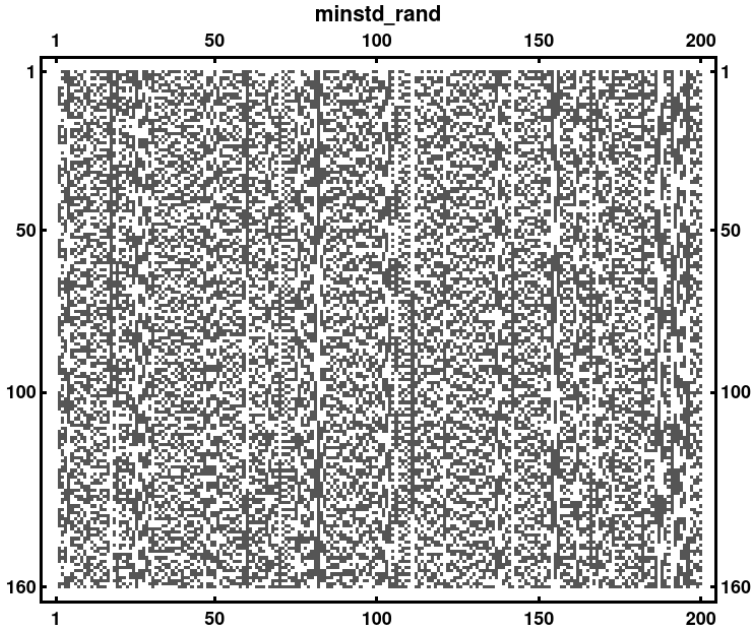
Language	Fist release	Default PRNG	Quality
`C`	1972	LCG	Bad
`Matlab`	1979	Mersenne Twister	Decent
`C++`	1985	Mersenne Twister / LCG	Decent / Bad
`Python`	1991	Mersenne Twister	Decent
`GNU Octave`	1993	Mersenne Twister	Decent
`Python NumPy`	2006	Mersenne Twister (below v.1.17), PCG64 (above v.1.17)	Decent / Good
`Julia`	2012	Xoshiro256++	Good
`Rust`	2015	ChaCha12 / Xoshiro256++	Good

Тут хорошо видна тенденция перехода на более быстрые & качественные движки, однако все что помечено “Decent” в таблицу уже является приемлемо хорошим для большинства моделей.

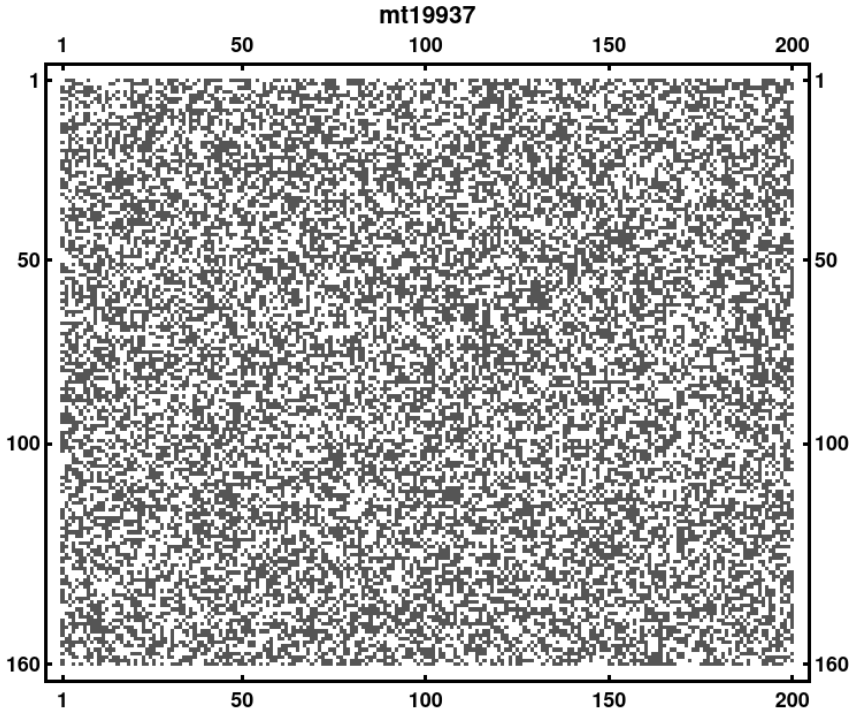
Чаще всего плохой рандом встречается в расчетных программах на **C** в силу использования rand(), который использует крайне старую версию LCG на многих компиляторах. Ниже я построил визуализацию корреляции между разными зернами, заполним матрицу 200x160 случайными нулями и единицами, отобразим её в виде изображения где 0 – белые пиксели, 1 – черные.

При построении матрицы на каждой строке Ni выбираем зерно рандома Ni, таким образом каждая строка пикселей соответствует реализации некоторого случайного процесса, который скачет между 0 и 1.

При использовании “плохого” рандома LCG картинка получается следующая:



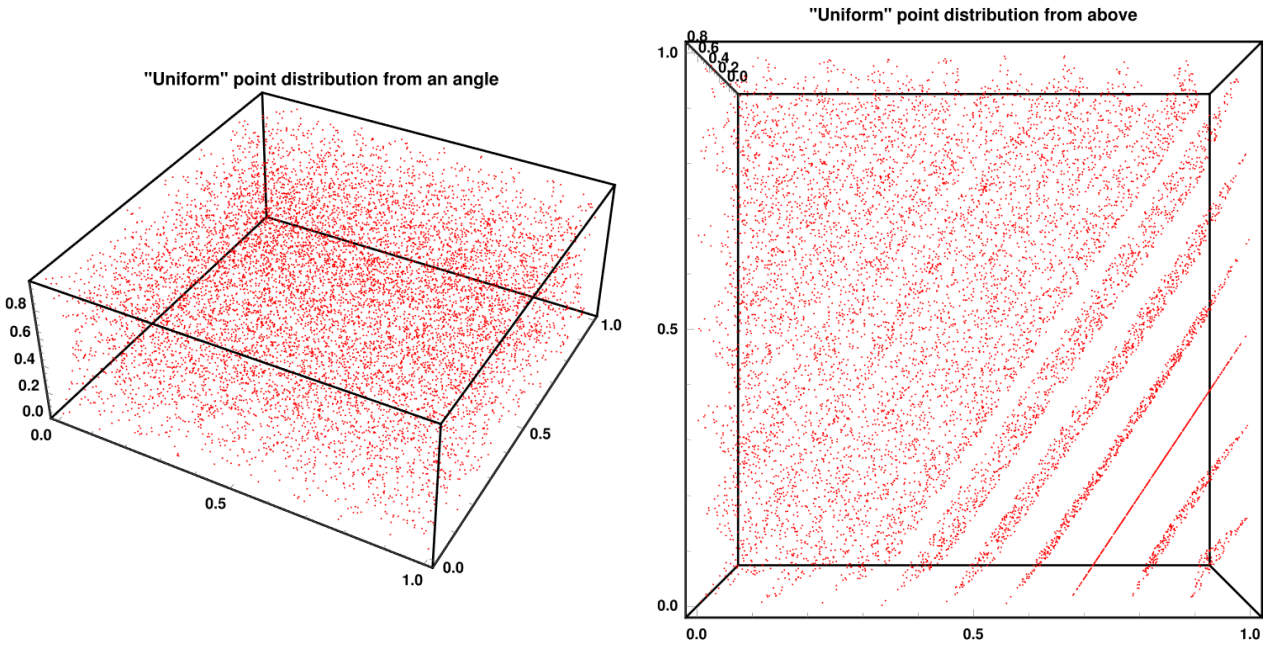
На изображении видим артефакты искусственной корреляции. При использовании рандома с качеством хотя бы “Decent”, таких артефактов наблюдаться не будет:



Примеры совсем плохого рандома

В годах 1970-1980-х очень часто пользовались методом RANDU от IBM, это по сути тот же самый движок LCG (minstd_rand) который был показан как “плохой” выше, только это еще более наивная и примитивная версия.

Если заполнить куб [0,1]x[0,1]x[0,1] случайными тройками точек с помощью RANDU, то увидим, что они все попадают на всего лишь 12 отдельных плоскостей:



Это визуализация того как метод не проходит спектральный PRNG тест для размерности 3, такая проблема свойственная всем LCG методам, просто в их случае размерность слишком высокая чтобы её визуализировать, точки лежат на конечном числе гиперплоскостей в многомерном пространстве.

Семейству Mersenne Twister данная проблема не свойственна, хотя если посмотреть в размерностях 100+, то я видел некоторые экономические модели где геометрия тоже начинала редуцироваться.

Нормальное распределение

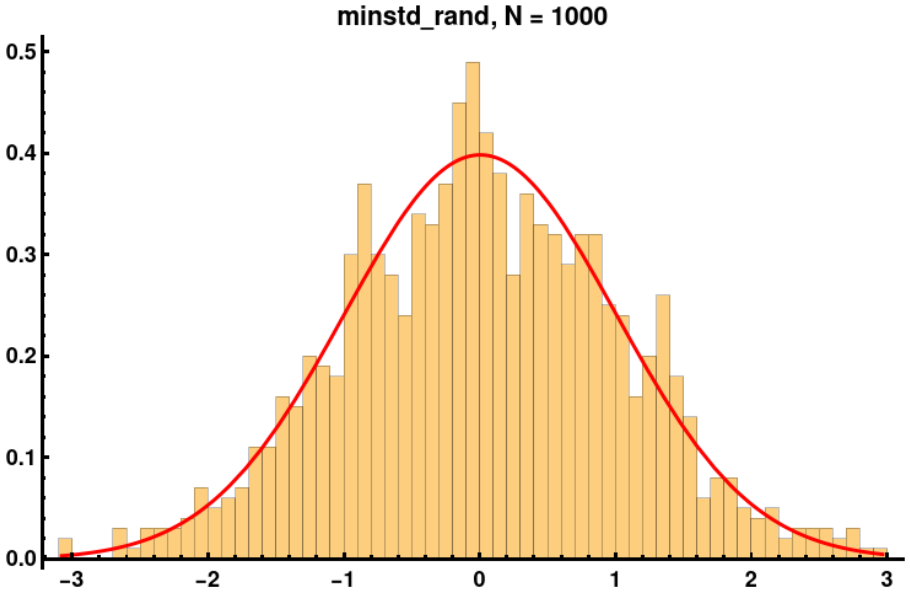
Не похоже чтобы выбор генератора случайных чисел имел сильное влияние на “ровность” распределения при малых N.

Зрительно увидеть эффекты плохого генератора в 1D случае сложно даже для совсем плохих генераторов, как правило дефекты проявляются в виде корреляции разных реализаций или в высоких размерностях.

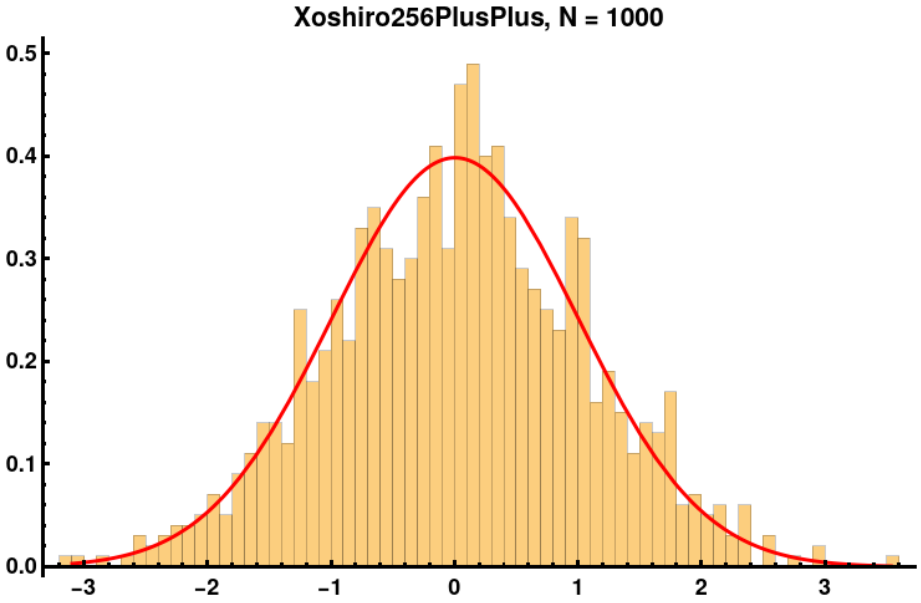
Может приводить к ошибке плохая формула перехода от случайных чисел к нормальному распределению, в C++ это делается через std::normal_distribution<double>, в Matlab’е скорее всего тоже проблем не должно возникнуть.

В некоторых сферах применяют псевдослучайные генераторы, которые учитывают прошлые результаты и генерируют случайные значение “равномернее” чем они на самом деле должны быть статистически, это можно сделать, но это не выглядит как хорошее решение, лучше просто считать с достаточно большими N.

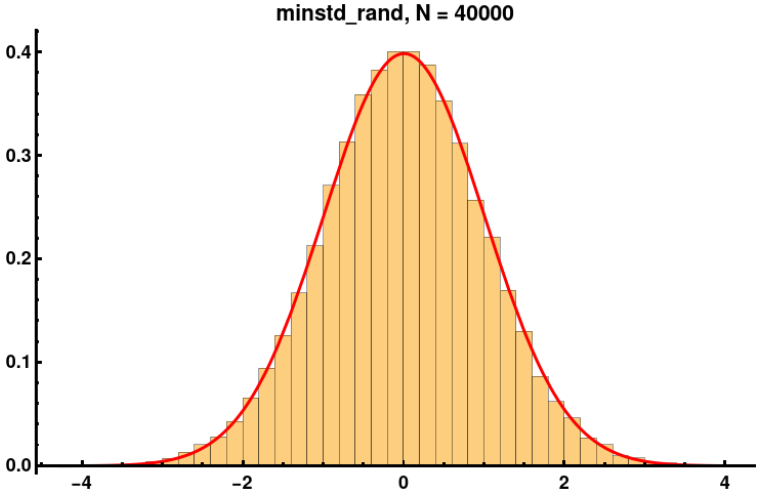
Нормальное распределение, “плохой” метод генерации случайных чисел, малый N:



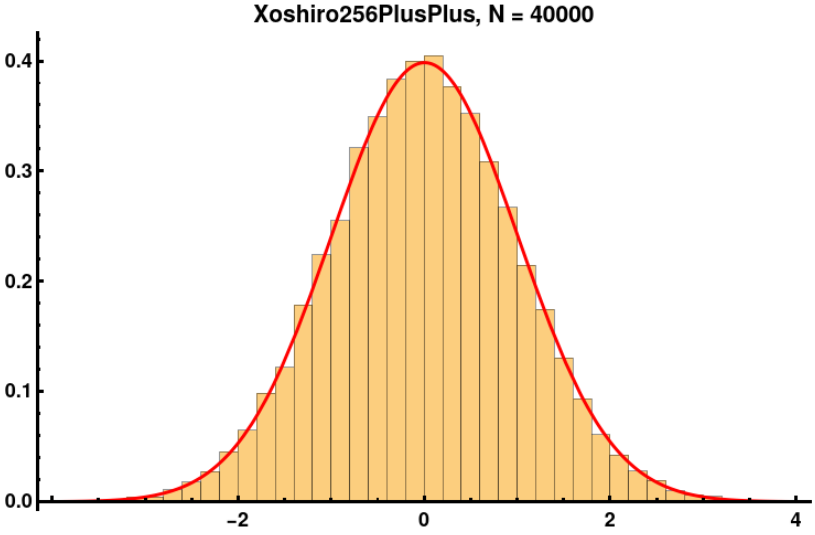
Нормальное распределение, “хороший” метод генерации случайных чисел, малый N:



Нормальное распределение, “плохой” метод генерации случайных чисел, большой N:



Нормальное распределение, “хороший” метод генерации случайных чисел, большой N:



Скорость генераторов

Пожалуй главным выигрышем современных генераторов является их великолепная скорость, после некоторого исследования, тестов и замеров я пришел к реализации **6** разных движков рандома с различным балансом качеств. Они будут перечислены в таблице ниже.

Для диплома это особого смысла на самом деле не имеет, так как генерация случайных чисел занимает не так уж и много времени, но дело оказалось интересное.

В рамках приемлемого качества рандома удалось получить ускорение до 5 раз относительно традиционно используемого Mersenne Twister (360% / 70% ~= 5), в качестве самого надежного и проверенного выбора – Xoshiro256++, который дает ускорение в 2.5 раз, требует в 150 раз меньше памяти чем Mersenne Twister который зашит в Matlab, и при этом имеет даже лучшее качество рандома. Сейчас его часто используют, на момент разработки Matlab такие семейства методов просто еще не были разработаны.

Overview of available PRNGs

Generator	Performance	Memory	Quality	Period	Motivation
`RomuTrio32`	~200% (450%)*	12 bytes	★★★☆☆	$\geq 2^{53}$	Fastest 32-bit PRNG
`JSF32`	~200% (360%)*	16 bytes	★★★☆☆	$\approx 2^{126}$	Fast yet decent quality 32-bit PRNG
`RomuDuoJr`	~195%	16 bytes	★★☆☆☆	$\geq 2^{51}$	Fastest 64-bit PRNG
`JSF64`	~180%	32 bytes	★★★★☆	$\approx 2^{126}$	Fast yet decent quality 64-bit PRNG
`Xoshiro256PlusPlus`	~175%	32 bytes	★★★★☆	$2^{256}-1$	Best all purpose 64-bit PRNG
`Xorshift64Star`	~125%	8 bytes	★★★☆☆	$2^{64}-1$	Smallest state 64-bit PRNG
`std::minstd_rand`	100%	8 bytes	★☆☆☆☆	$2^{31}-1$	
`std::mt19937`	~70%	5000 bytes	★★★☆☆	$2^{19937}-1$	
`std::ranlux48`	~4%	120 bytes	★★★★☆	$\approx 2^{576}$	

[*] A lot of CPUs lacks 64-bit `rotl` instructions, which can make 32-bit versions offer up to **300–500% speedup**.

*движки были реализованы в рамках своей библиотеки утилит C++, таблицы взяты и её документации

Итог

Дело интересное, в диплом определенно можно многое добавить. Практического смысла в этом на самом деле не сильно много, это скорее вопрос разработки новых программных комплексов, но с точки зрения сторогости и современности подхода реализовать свой генератор хорошо.