Focus Project: On Pseudo-Random Number Generators

2025S 280522-1: Methods of Computational Astrophysics

Simone Spedicato, BSc

July 2025

1 Introduction

The ability to generate random numbers is a fundamental tool in computer science, numerical analysis, and astrophysics. Randomness underpins a wide range of applications, including Monte Carlo simulations, statistical sampling, and modeling complex physical systems such as star clusters.

However, truly random numbers are inherently difficult to produce deterministically. While hardware devices and quantum processes can provide sources of entropy, most practical applications rely on *pseudo-random number generators* (PRNGs). These are algorithms that, starting from an initial seed, generate sequences that approximate randomness.

In this report, I examine several widely used PRNGs, describe methods to evaluate their quality, and demonstrate how poor parameter choices or inadequate generators can introduce patterns and bias. Finally, I apply these concepts to a simulation of a star cluster, highlighting how an inappropriate choice of PRNG can lead to visibly degraded and unphysical results.

2 Methods

This analysis involves a literature review to identify the most commonly used PRNGs in modern applications, followed by implementing these generators in Python. Finally, I developed a suite of tests to evaluate their properties.

2.1 PRNGs

2.1.1 LCG

The **Linear Congruential Generator** (LCG) is one of the oldest and most widely used methods for generating pseudo-random numbers [1]. It is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m \tag{1}$$

where X is the sequence of pseudo-random values, determined by four parameters:

- **Modulus** $m > 0$
- **Multiplier** $0 < a < m$
- **Increment** $0 \leq c < m$
- **Seed** $0 \leq X_0 < m$

At each step, the generator updates its internal state and produces a new pseudo-random number. LCGs are extremely fast and simple to implement, and are fully deterministic (the same seed always yields the same sequence). However, the statistical quality of the output depends strongly on the choice of parameters. Poor parameter choices can lead to short periods or visible patterns.

Intuitively, the modulo operation keeps the values within a fixed range, while the recurrence relation causes the sequence to move through this range in a way that can approximate randomness when designed carefully.

The generator class is implemented with the following simple code:

```
from collections.abc import Generator

class LCG:
    def __init__(self, seed, a=1664525, c=1013904223, m=2**32):
        self.state = seed
        self.a = a
        self.c = c
        self.m = m

    def next(self):
        self.state = (self.a * self.state + self.c) % self.m
        return self.state / self.m
```

2.1.2 Mersenne Twister

Another very famous and important PRNG is the **Mersenne Twister**. This is the default generator used by Python's `random` module. It produces 53-bit precision floating-point numbers and has an enormous period of $2^{19937} - 1$ [2].

The standard C implementation is both fast and thread-safe, but for educational purposes I have written my own version. The Mersenne Twister is one of the most extensively tested random number generators ever developed. However, because it is completely deterministic, it is not suitable for all applications—especially cryptographic ones.

The **Mersenne Twister (MT19937)** maintains an internal state consisting of 624 32-bit integers.

1. Initialization The state array `mt[0..623]` is initialized from a seed using:

$$mt[i] = \big(f \cdot (mt[i-1] \oplus (mt[i-1] \gg (w-2))) + i\big) \bmod 2^{32}$$

where:

$$f = 1812433253, \quad w = 32.$$

and $\oplus$ means bitwise exclusive OR (XOR).

2. Twisting When all state values have been consumed, the generator *twists* to produce new ones:

$$x = \big(mt[i] \,\&\, \text{upper\_mask}\big) + \big(mt[(i+1) \bmod 624] \,\&\, \text{lower\_mask}\big)$$

$$mt[i] = mt[(i+397) \bmod 624] \oplus (x \gg 1)$$

If x is odd, an additional XOR with a constant a is applied.

3. Tempering Before producing output, a sequence of bit shifts and masks improves the statistical properties:

$$y = y \oplus \big((y \gg u) \,\&\, d\big) \oplus \big((y \ll s) \,\&\, b\big) \oplus \big((y \ll t) \,\&\, c\big) \oplus (y \gg l)$$

The tempered value is then scaled into the interval $[0, 1)$.

2.1.3 Common Entropy Sources

While pseudo-random number generators (PRNGs) are deterministic algorithms, they often rely on external **entropy sources** for their initial seed. These sources provide unpredictable data to ensure that the generated sequences vary between runs and are not easily guessable [3].

Most modern operating systems maintain an *entropy pool*, which collects unpredictable information from various hardware and system events, including:

- Timing of user input (keyboard/mouse activity)
- Disk I/O events
- Network packet arrival times
- Hardware interrupts

On Unix-like systems, common interfaces to these entropy pools include:

- `/dev/random`: a blocking interface that waits for high-quality entropy
- `/dev/urandom`: a non-blocking interface that uses a cryptographically secure PRNG seeded from the entropy pool

In pratice, most high-level libraries (e.g., `numpy`, `random`, or `secrets` in Python) automatically seed their internal PRNGs using system entropy. However, for reproducibility in simulations or debugging, it is common to manually set the seed.

2.2 Tests

To evaluate the quality of the implemented pseudo-random number generators (PRNGs), I designed two main sets of statistical tests. These tests assess both the basic properties of the output distribution and the presence of unwanted patterns or correlations.

First Set: General Tests The first set, called `general_tests`, consists of relatively simple checks on a large number of generated samples. All of the analyzed PRNGs are expected to pass these baseline tests:

- **Moments Test:** Verify that the sample mean and variance are close to the theoretical values for a uniform distribution over [0,1]:

$$\text{Expected mean} = 0.5, \quad \text{Expected variance} = \frac{1}{12} \approx 0.0833.$$

- **Chi-Squared Uniformity Test:** Assess how well the sample histogram matches a uniform distribution. A p-value below 0.05 would indicate significant deviation.
- **Lag-1 Autocorrelation Test:** Measure the correlation between successive values. Ideally, the autocorrelation should be near zero.

Second Set: Advanced Tests The second set of tests explores more subtle structural properties of the generated sequences:

- **2D Embedding Test (X_i vs. X_{i+1}):** Plot pairs of successive values to visually inspect for patterns or clustering that indicate poor mixing.
- **3D Embedding Test (X_i vs. X_{i+1} vs. X_{i+2}):** Similar to the 2D test but in three dimensions, helping detect lattice structures that can arise in some generators.
- **Birthday Spacing Test:** Generate random integers mapped into a fixed number of bins and record the spacing between sorted samples. The distribution of these spacings can reveal hidden correlations or non-uniformity.
- **Period Length Estimation:** Estimate the cycle length by observing how many generated values occur before the sequence repeats. While this test cannot determine the exact period for large PRNGs, it is useful to confirm that the cycle length exceeds practical thresholds.

These combined tests allow a thorough comparison of PRNG behavior, highlighting maybe subtle deficiencies in randomness quality.

2.3 Physical Simulation

The physical simulation to test some of the PRNGs out is based on *Exercise Sheet 7*, in which we simulated a star cluster using the Plummer model [4]. I expect to see basically no differences if the PRNgs are given good starting conditions, but I am interested to see how bad parameters can influence the distribution of stars.

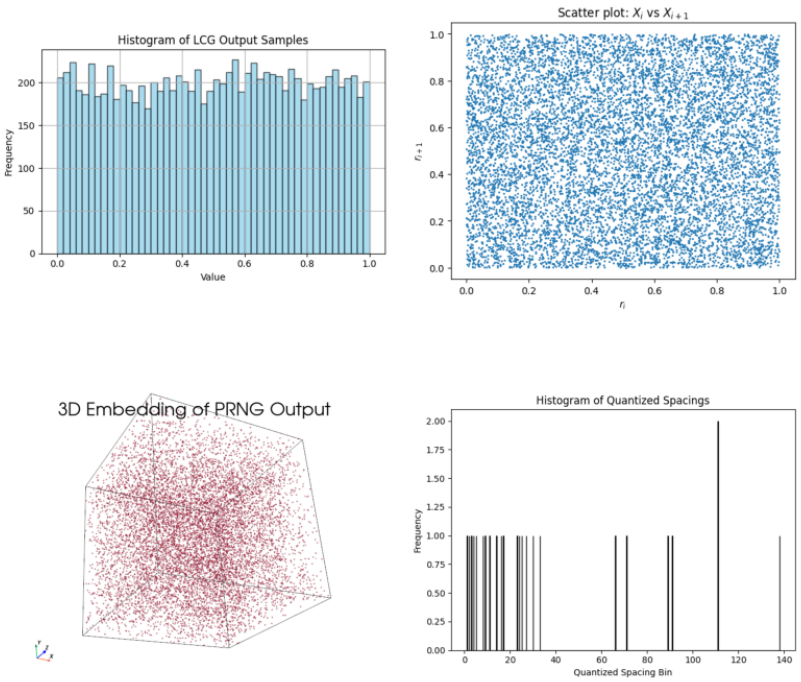


Figure 1: Example results of the test suite applied to the LCG: histogram of samples, 2D and 3D embeddings, and histogram of birthday spacings.

3 Results

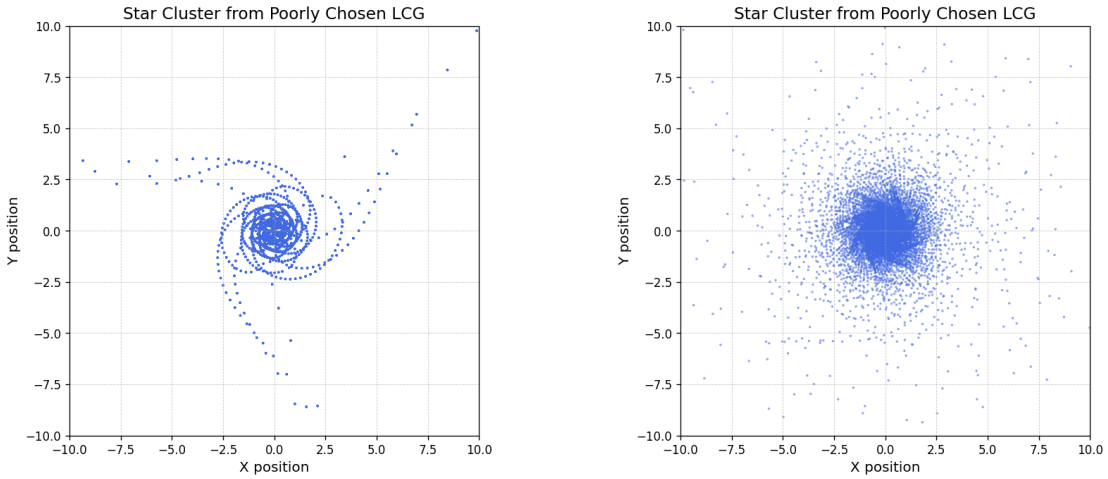
As expected, the methods evaluated—including the Linear Congruential Generator (LCG), the Mersenne Twister, and system-provided entropy sources—all successfully passed the majority of the tests.

Figure 1 shows example outputs of the test suite applied to the LCG, showing the histogram of generated samples, 2D and 3D embeddings, and the histogram of birthday spacings.

The only test where notable deviations appeared was the birthday spacing test when using large sample sizes ($N > 1000$). In those cases, the LCG occasionally produced spacing distributions inconsistent with the uniform hypothesis.

3.1 Simulation Results

When using good or standard parameters, the simulated star cluster distributions appear isotropic and uniform, as expected. However, when intentionally selecting poor parameters for the LCG, clear artifacts and patterns emerge in the generated points, as



(a) LCG with small modulus

(b) LCG with modulus = power of 2

Figure 2: Examples of poor parameter choices in LCG resulting in strong patterns.

it can be seen in Figures 2a and 2b compared with Figure 3.

Specifically:

- With a **small modulus**, the generator quickly repeats values, leading to pronounced clustering and visible repetitions in the spatial distribution.
- With a **modulus that is a power of 2 and zero increment**, many outputs fall on planes or lines in space, creating strong correlations between coordinates.

These examples illustrate why careful parameter selection is essential to avoid hidden regularities in pseudo-random sequences.

4 Discussion & Concluding Remarks

We examined several PRNGs using a custom suite of tests to assess their statistical properties. As expected, the most widely used generators generally passed these evaluations, with failures occurring primarily in more specialized, higher-dimensional tests such as the birthday spacing test.

Our results underscore that parameter selection is critically important. Poorly chosen seeds or parameters can lead to highly degenerate or predictable outputs. To illustrate this, we applied deliberately flawed PRNG configurations to a simulation of a star cluster, clearly demonstrating how inadequate randomness can produce visible artifacts and unrealistic results.

The full project is available on GitHub.

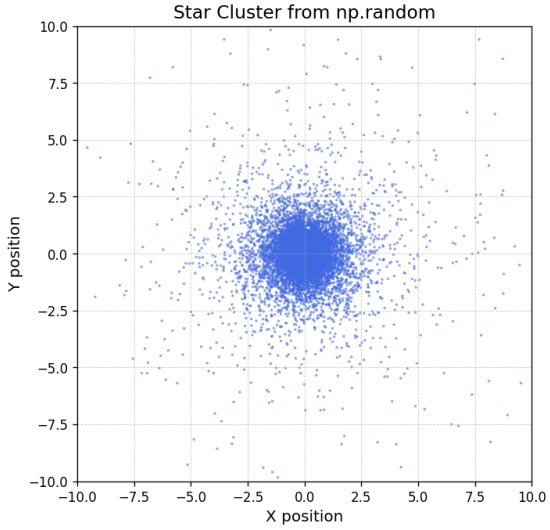


Figure 3: Reference simulation using `numpy.random`, producing a uniform isotropic distribution.

References

[1] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[2] Makoto Matsumoto and Takuji Nishimura. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30.

[3] Meltem Sönmez Turan et al. *Recommendation for the entropy sources used for random bit generation*. Tech. rep. National Institute of Standards and Technology, 2016.

[4] Henry Crozier Plummer. "On the problem of distribution in globular star clusters". In: *Monthly Notices of the Royal Astronomical Society, Vol. 71, p. 460-470* 71 (1911), pp. 460–470.

Prompt List

- **ChatGPT Prompt:** `What are the most important PRNGs that should be analyzed for a project in computational astrophysics?`