PRNG - Statistical testing

Maciej Szczutko

2024-12-22

Statistical test description

In this short report I would like to perform some statistical test to check whether the sequence is random or not. In general the common part for each test is null hypothesis.

H_0 : The sequence is random H_A : The sequence is not random.

In most cases the interfere will be based on p-value. It's convienient to perform the 2nd level testing later.

χ^2 test (goodnes of fit)

Here we divide some interval into buckets and check distribution within bucket. Usually we will use equally length buckets. The test statistic is

$$\hat{\chi}^2 = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i}.$$

- Test parameters
  - k - number of categories/buckets
  - p_i - probability that value belong to i-th bucket ($i = 1, 2, \ldots, k$)
- Statistic params explanation
  - n - sample size
  - Y_i - number of elements in sample belong to i-th bucket

Implementation in *chi_square_module*. The implementation make assumption that sample lives in $(0, 1)$ for simplicity.

Frequency monobit test

The test is designed for generator yielding binary sequences. Suppose we have a bit sequence $b_1, b_2, \ldots, b_i \in \{0.1\}$. We convert the sequence to $x_1, x_2, \ldots$ with values $\{-1, 1\}$ via $x_i = 2b_i - 1$. Test statistic is

$$s_n(obs) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} x_i.$$

Under H_0 test statistic is approximetly $N(0, 1)$ by CLT.

KS-test

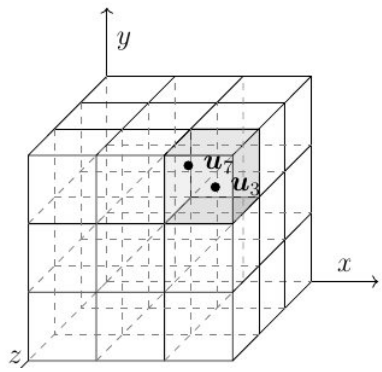
This test is based on empircal cdf function and described in project document. For KS test I will use the implementation from **scipy** module.

scipy.kstest

Serial test (m, L)

Serial test is quite easy to understand test that incorporate relation between sequence's elements. Roughly speaking we split vector into r equally length vectors with dimension m and check how they are distributed within $[0,1]^m$ hybercube. We divide the hybercube into L^m smaller cubes (see picture below). Assuming that sequence is uniformly distributed we expect the distribution within cubes should be also uniform. It multidimensional equivalent for ordinary χ^2 test (if $m \geq 2$).

The implementation is **serial_test** module.



Test parameters and statistic description

- Test parameters
  - m - dimension of hypercube
  - L - "number of bucket for each dimension" control the granularity
- Statistic paramas explanation
  - r - number of vectors (dimension m) obtained from sample
  - $k = L^m$ - number of sub hypercube, granularity (should be significant lower than r)
  - O_i - vector count in each i-th subhypercube

$$X^2(obs) = \sum_{i=1}^{k} \frac{(O_i - r/k)^2}{r/k}.$$

Under H_0 statistic has χ^2_{k-1} distribution.

Second level testing

The idea of second level testing is also straightforward. We assume we have quite big sample and we can split it into many smaller. Then for each subsample evalute the test and calculate p_value. Then use χ^2 test and check whether p_values fit $U[0,1]$. NIST recommend use 10 equal length bucket.

Testing binary expansion of constans

In this section we perform frequency monobit test for number $\pi, e, \sqrt{2}$. More formally we will use their binary expansion as the random bit sequence. We use provided files with binary expansions. For inference we will follow instruction from official **NIST** report.

Some important notes from report about most basic test.

> 2.1.5 Decision Rule (at the 1% Level)
>
> If the computed P-value is < 0.01, then conclude that the sequence is non-random. Otherwise, conclude that the sequence is random.

2.1.7 Input Size Recommendation

It is recommended that each sequence to be tested consist of a minimum of 100 bits (i.e., $n \geq 100$).

Constant name	p_{value}	Input Size
π	0.612315825298478	1004858
e	0.928460306674579	1004858
$\sqrt{2}$	0.817749242838411	1004859

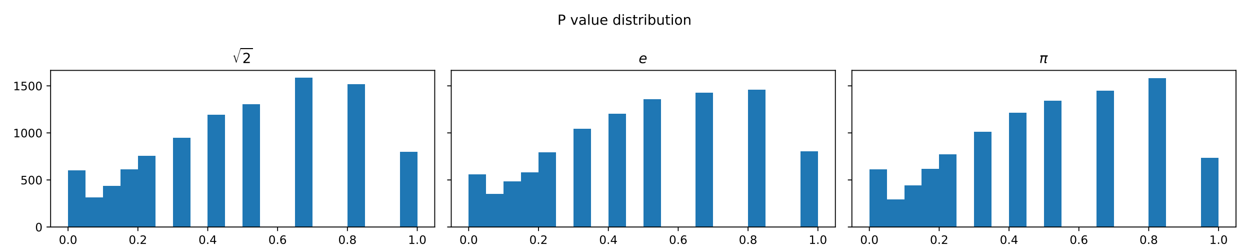
The size of out data is aligned with **NIST** recommendations. Authors recommend 0.01 as significance level for PRNG testing. From above table we conclude that binary expansion of each mentioned constants could be considered as random bit sequence.

Second level testing for bits

Because the size of sample are quite big we can try another approach. We split the sequence into a lot of smaller samples. We use the smallest recommended $n = 100$ for this test. The split method will be very straightforward. We simply take first 100 bits for first sample, another 100 for second sample and so on.

In below table we can see what fraction of samples pass the single test.

Constant name	Fraction that pass
π	0.9867635
e	0.9876592
$\sqrt{2}$	0.9874602



We can take a look at the histogram of p values. For me it doesn't look like uniformly distributed. I expect the second level test fail (not fit to $U[0,1]$ and thus not random sequence).

Constant name	Statistic	P-value
π	18.1958957	0
e	16.4493193	0
$\sqrt{2}$	18.4055295	0

As I expect the χ^2 test show that the p-value obtained from frequency monobit test are not uniformly distributed. The p values for all constants are extremely low so we reject hypothesis. This is only for one setup (the way I obtained the sample from constans), but I think is rationale to not use constans expansion as PRNG.

Generator description

First we give some basic idea and formulas of generators.

Linear congurent

The most basic one using linear dependence as function of previous element. Need the single number as seed.

$$x_{n+1} = (ax_n + c) \mod M.$$

Generalized linear congurent

Natural extension on above. Now the output is generated based on last k elements in sequence. Note that the seed is now k element sequence instead of single number.

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + \ldots + a_k x_{n-k} + c) \mod M.$$

Excell

Weird one. Probably used in earlier MS Office implementation (but hard to find some information). This is LCG with non-integeer coefficients. Yielding number from $[0, 1]$.

$$u_i = (0.9821 u_{i-1} + 0.211327) \mod 1$$

MT19937

Implementation from numpy module.

To do. Add description.

RC(32)

The RC(32) random generator is a computational tool designed to produce sequences of random or pseudo-random numbers within a defined range. It typically employs algorithms optimized for speed and uniform distribution, making it suitable for simulations, cryptography, and data sampling. With its 32-bit architecture, the generator can provide high precision and a large range of outputs, ensuring versatility for various applications. Modern implementations often focus on enhancing randomness quality, reducing predictability, and adhering to statistical randomness tests like DIEHARD or TestU01. The RC(32) is favored in environments where reliable randomness is critical, such as in secure key generation or randomized algorithms. Additionally, its efficiency ensures minimal computational overhead, making it well-suited for embedded systems and high-performance computing tasks.

I provide some naive python implementation in *PRGA* and *KSA* modules. I will use my birthday (from year, month and day) as key (mod 32). So the key is [15, 8, 29].

```
## Elapsed generation time of sample size = 1000000 is 3.0716331005096436s
```

We can check how the sequence begin 25, 15, 1, 22, 14, 4, 11, 26, 31, 1, 7, 28, 14, 15, 20, 3, 16, 21, 16, 16, 0, 21, 31, 19, 12, 15, 22, 4, 28, 1, 25, 9, 29, 20, 9, 21, 17, 17, 24, 21, 19, 22, 15, 31, 1, 20, 8, 31, 31, 12…

LCG$(13, 1, 5)$ used seed $= 42$

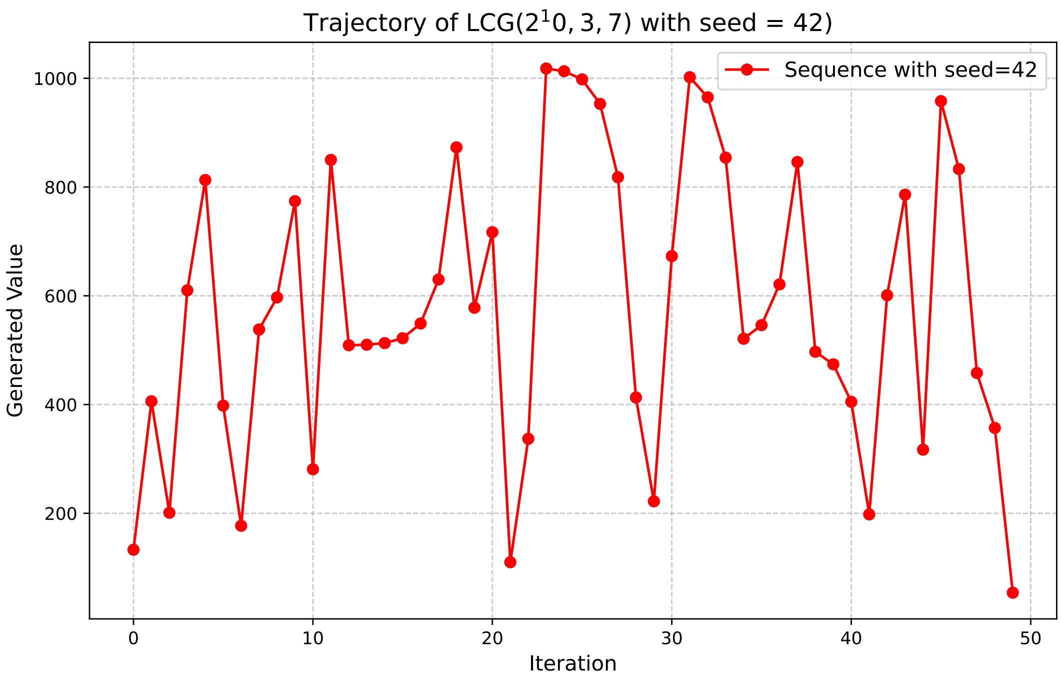
We can check how the sequence begin 8, 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, 8, 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3…

For this example we can observe period of generator due relative small parameter M. The period can't be greater than M, so here just by looking on first 26 elements we can see the how the sequence repeat.

LCG($2^{10}, 3, 7$) used seed $= 42$

We can check how the sequence begin 133, 406, 201, 610, 813, 398, 177, 538, 597, 774, 281, 850, 509, 510, 513, 522, 549, 630, 873, 578, 717, 110, 337, 1018, 1013, 998, 953, 818, 413, 222, 673, 1002, 965, 854, 521, 546, 621, 846, 497, 474, 405, 198, 601, 786, 317, 958, 833, 458, 357, 54…

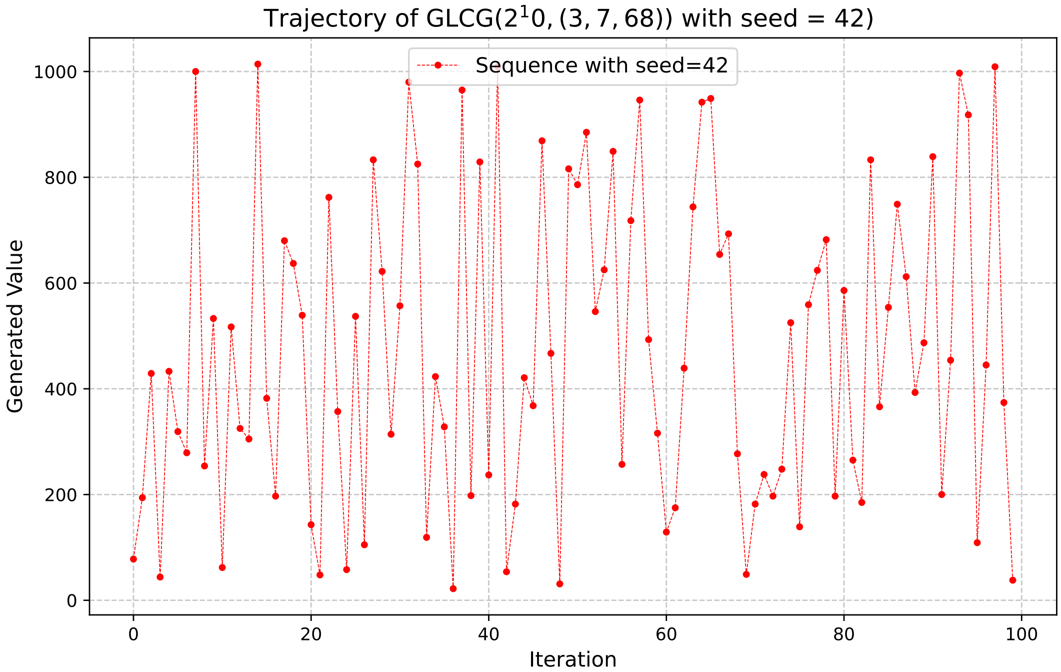
Comparing with previous example we can't determine the period by manual inspection now.



GLCG($2^{10}, 3, 7, 68$)

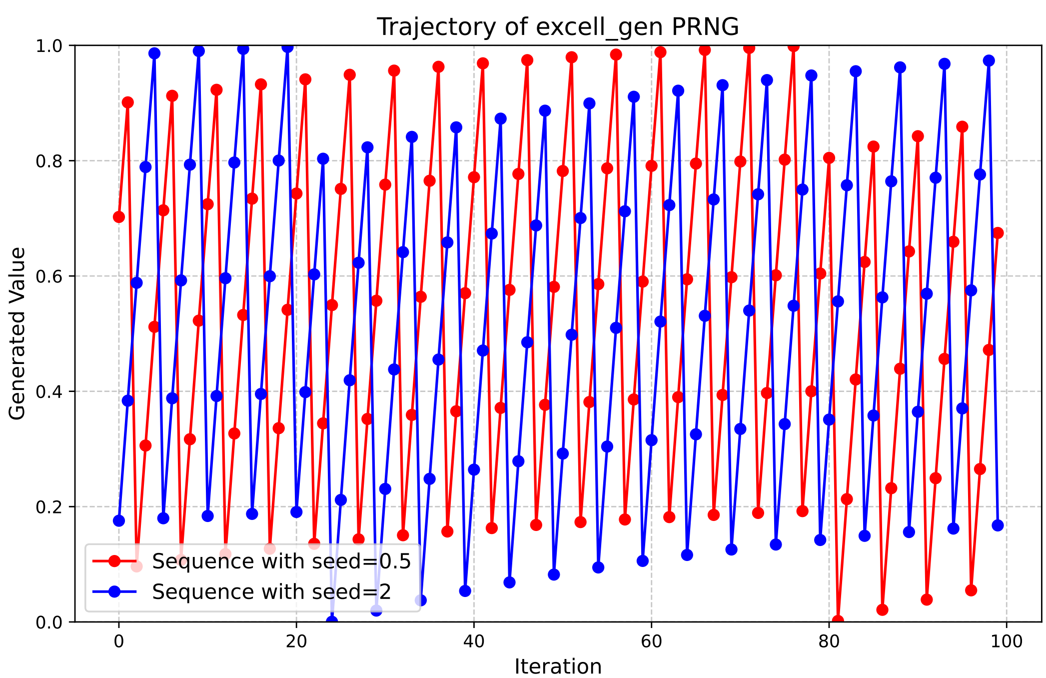
We can check how the sequence begin 78, 194, 429, 44, 433, 319, 279, 1000, 254, 533, 62, 517, 325, 305, 1014, 382, 197, 680, 637, 539, 143, 48, 762, 357, 58, 537, 105, 833, 622, 314, 557, 980, 825, 119, 423, 328, 22, 965, 198, 829, 237, 1009, 54, 182, 421, 368, 869, 467, 31, 816.

Due complexity of PRNG and relative large M we can't see the pattern now. Even graph inspection won't help.



Excell

```
## (0.0, 1.0)
```



Plotting the trajectory of beginning give us overview of quality of such generator. Maybe it simple and fast but not safe. We can observe pattern easily for difference seeds.