Wprowadzenie do symulacji i metod Monte Carlo
Projekt 1

Zuzanna Brzóska, anonymous second person

22 maja 2023

Spis treści

1 Wprowadzenie

Celem niniejszego projektu jest opisanie wybranych generatorów liczb pseudo-losowych (czyli liczb w pewien sposób przypominających liczby losowe) oraz sprawdzenie ich własności i jakości, bazujące na testach statystycznych.

2 Generatory

Na początek skupimy się na przedstawieniu poszczególnych generatorów liczb pseudolosowych, czyli algorytmów zwracających sekwencje liczb (lub bitów), które wydają się losowe. Poza generatorem *XORG* zwracającym ciąg bitów, wszystkim generatorom liczbowym można w naszej implementacji przekazać argument informujący czy chcemy otrzymać liczby z przedziału $(0, 1)$ czy ze zbioru $\{0, \ldots, M-1\}$ dla pewnego M (zwykle będącego potęgą 2).

2.1 LCG

Liniowy generator kongruentny $LCG(M, a, c)$ jest jednym z prostszych sposobów generowania liczb pseudolosowych postaci $u_n = x_n/M$. Generator zdefiniowany jest poprzez rekurencję

$$x_{n+1} = (ax_n + c) \bmod M,$$

gdzie a stanowi mnożnik, c przyrost, M moduł, a x_0 jest wartością początkową i każdy z parametrów pochodzi z przezdziału $[0, M)$. Liczba pseudolosowa jest resztą z dzielenia przez M, zatem przyjmować może wartości od 0 do $M-1$. Generowana sekwencja będzie miała okres o długości co najwyżej M.
W testach za (M, a, c) przyjmować będziemy $(13, 1, 5)$, $(2^{10}, 3, 7)$ oraz $(2^{10}, 5, 7)$. W ostatnim przypadku parametry spełniają twierdzenie pozwalające określić kiedy okres wygenerowanego ciągu ma długość równą M, a mianowicie:
$\rightarrow$ c jest względnie pierwsze z M,
$\rightarrow$ $a-1$ jest wielokrotnością każdej liczby pierwszej dzielącej M,
$\rightarrow$ jeśli $4|M$, to $a = 1 \bmod 4$.

2.2 GLCG

$GLCG(M, (a_1, a_2, \ldots, a_k))$ stanowi uogólnienie poprzedniego generatora, gdzie liczby generowane są na podstawie rekurencji

$$x_n = (a_1 x_{n-1} + \ldots + a_k x_{n-k}) \bmod M,$$

dla danych $x_0, x_1, \ldots, x_{k-1}$.
Dla odpowiednio dobranych stałych a_j, przy czym M - liczba pierwsza, możemy otrzymać okres długości M^k. W projekcie przetestujemy $GLCG$ dla parametrów $M = 2^{10}, k = 3, (a_1, a_2, a_3) = (3, 7, 68)$, gdzie spodziewamy się zdecydowanie krótszego cyklu oraz dla $M = 2^{13} - 1, k = 3, (a_1, a_2, a_3) = (3, 7, 68)$ (zauważmy, że $2^{13} - 1$ jest liczbą pierwszą, więc istnieje większa szansa, na dłuższy okres ciągu).

2.3 Lagged Fibonacci generator

Opóźniony generator Fibonacciego *LFG* jest ulepszeniem liniowego generatora kongruencyjnego i opiera się na uogólnieniu ciągu Fibonacciego. Kolejne liczby wyrażają się następującym wzorem

$$x_n = x_{n-k} + x_{n-l} \bmod M \quad (n \geq \max(k, l)).$$

Zbadamy również inną modyfikację tzw. *subtractive lagged Fibonacci generator* zadaną rekurencją

$$x_n = x_{n-k} - x_{n-l} \bmod M \quad (n \geq \max(k, l)).$$

Powyższe generatory przetestujemy z parametrami $2^{10}, 5, 12$ oraz z parametrami zaproponowanymi przez Knutha - $2^{30}, 100, 37$, a pierwszy z nich dodatkowo z parametrami $2^{10}, 1, 2$, co odpowiada generatorowi Fibonacciego bez opóźnienia.

2.4 RC4(32)

Generator $RC4(m)$ należy do nieco bardziej zaawansowanych generatorów, ponieważ tak na prawdę składa się z dwóch algorytmów: KSA oraz PRGA. Parametrem przyjmowanym przez niego jest pewien ciąg liczb (klucz) ze zbioru $1, \ldots, m-1$, który podlega permutacjom, do czego służy algorytm KSA, działający zgodnie z poniższą ideą:

`KSA(`K`)`

for $i := 0$ to $m-1$ **do**
  $S[i] := i$
end for

$j := 0$
for $i := 0$ to $m-1$ **do**
  $j := j + S[i] + K[i \bmod L]$
  swap$(S[i], S[j])$
end for
$i, j := 0$

gdzie S - początkowo permutacja identycznościowa, K - podany klucz o długości L. Następnie, na podstawie wyniku KSA, algorytm PRGA używając operacji XOR, zwraca liczby pseudo-losowe, również ze zbioru $\{1, ..., m-1\}$, zgodnie z poniższym:

`PRGA`

while $r \in \mathcal{N}_+$ **do**
  $i := i + 1$
  $j := j + S[i]$
  swap$(S[i], S[j])$
  $Y_r \leftarrow S[S[i] + S[j]]$
end while

Przetestujemy powyższy generator dla kluczy $\{0, 1, 2, \ldots, 31\}$ oraz $\{15, 10, 21, 13, 24\}$.

2.5 Exclusive OR Generator

Jest to generator zwracający bity (liczby 0 lub 1). $XORG$ opiera się na ciągu 127 bitów (nazywanym ziarnem), a kolejne liczby są postaci

$$x_n = x_{n-1} \oplus x_{n-127} \quad j \geq 128.$$

(Dla przypomnienia: operacja XOR daje wynik prawdziwy $\iff$ nieparzysta liczba zdań na wejściu jest prawdziwa). W projekcie opieramy się na ziarnie zaproponowanym w [1].

3 Testy

W tej sekcji przedstawimy wybrane testy statystyczne. W każdym z testów interesuje nam wyliczona dzięki niemu p-wartość, która informuje nas o prawdopodobieństwie zajścia hipotezy zerowej - w przypadku testowania generatorów H_0 zakłada, że ciągi pochodzą w rozkładu jednostajnego.

3.1 χ^2 test

Jest to bardzo popularny test statystyczny, w naszej implementacji przyjmuje liczby z przedziału $[0, 1]$. Test zakłada, że otrzymujemy n liczb, z których każda należy do jednej z k kategorii - tutaj są to równe podprzedziały odcinka jednostkowego. Liczymy liczbę obserwacji z kategorii s, czyli zmienną Y_s, która dla większych ciągów powinna dążyć do iloczynu np_s, gdzie p_s jest prawdopodobieństwem wpadnięcia obserwacji do kategorii s. Następnie liczymy statystykę

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

a także p-wartość korzystając z faktu, że powyższa statystyka powinna mieć rozkład χ^2 z $k-1$ stopniami swobody.

3.2 Kolmogorow-Smirnov test

W implementacji do niniejszego projektu zaproponowany przez nas test przyjmuje ciąg liczb z odcinka $[0, 1]$. Test sprawdza, czy rozkład w populacji dla pewnej zmiennej losowej, różni się od założonego rozkładu teoretycznego, gdy znana jest jedynie pewna skończona liczba obserwacji tej zmiennej. Dystrybuanta empiryczna dana jest wzorem

$$\hat{F}_X(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}(X_i \leq x).$$

Dla $n \to \infty$ statystyka

$$\hat{D}_n = \sqrt{n} \sup_{x \in \mathbb{R}} |\hat{F}_X(x) - F_X(x)|$$

dąży do rozkładu Kołmogorowa - Smirnova

$$P(\hat{D}_n \leq t) \to 1 - \sum_{i=1}^{\infty} (-1)^{i-1} e^{-2i^2 t}.$$

3.3 Frequency monobit test

Jest to test przyjmujący dane w postaci ciągu bitów, który sprawdza w nim proporcje 0 i 1. Po prawdziwie losowym ciągu spodziewamy się, że proporcje te powinny być zbliżone. W celu wykonania tego testu, przekształcamy bity 0 na -1. Test działa na podstawie Centralnego Twierdzena Granicznego: gdy mamy niezależne zmienne losowe $X_1, X_2, \ldots$ o rozkładzie $P(X_i = -1) = P(X_i = 1) = \frac{1}{2}$, to na podstawie CTG widzimy, że rozkład statystyki danej wzorem

$$S_n = \frac{1}{n} \sum_{i=1}^{n} X_i$$

dąży do $N(0, 1)$. Teoretyczna p-wartość dla $S_n = x$ będzie równa

$$P(|m| > |x|) = 2(1 - \phi(|S_n|))$$

(gdzie ϕ jest dystrybuantą rozkładu normalnego) i tak też ją liczymy.

3.4 Runs test

Runs test to drugi w tym projekcie test działający na ciągach bitów. Opiera się on na wyznaczeniu liczby nieprzerwanych sekwencji identycznych bitów w badanym ciągu. Przebieg długości k składa się z k identycznych bitów i z obu stron jest ograniczony bitem o przeciwnej wartości. Celem testu jest określenie, czy liczba przebiegów jedynek i zer o różnej długości jest zgodna z oczekiwaniami dla losowego ciągu. Do porównania z innymi wynikami i określeniem wyniku testu potrzebna jest również p-wartość, którą otrzymujemy ze wzoru

$$p = \boldsymbol{erfc}\left(\frac{|V_n - 2n\pi(1-\pi)|}{2\sqrt{2n}\pi(1-\pi)}\right),$$

dla V_n będącego statystyką liczoną jako

$$V_n = \sum_{k=1}^{n-1} r(k) + 1,$$

gdzie $r(k) = 0$, gdy sąsiednie bity są takie same, oraz $r(k) = 1$ w przeciwnym wypadku. Test dokładniej opisany jest w [2].

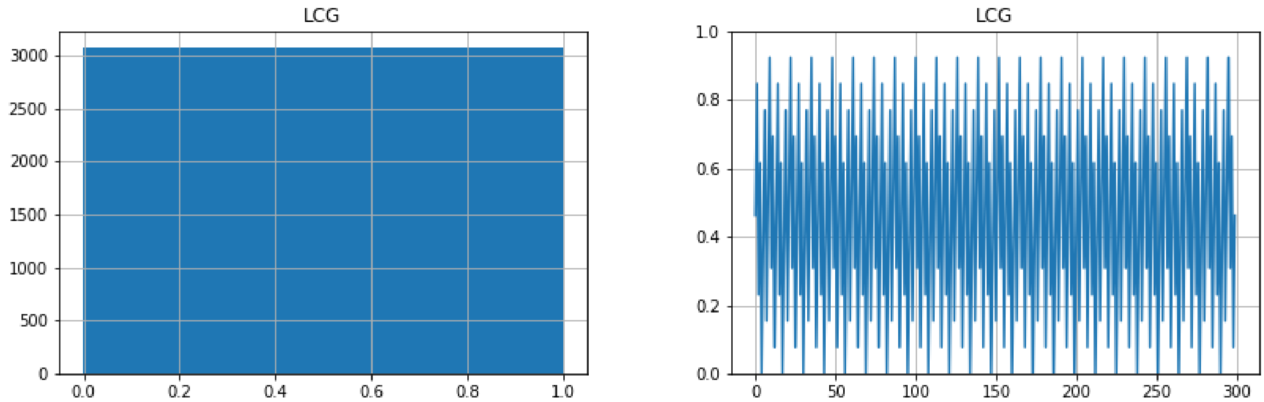
4 Second-level testing

Second-level testing polega na sprawdzeniu czy p-wartości otrzymywane z wcześniej opisanych testów również mają rozkład jednostajny na $(0, 1)$. Za pomocą wybranego generatora uzyskujemy ciąg liczb/bitów (mamy nadzieję, że pseudo-losowych) i dzielimy go na R podciągów, każdy o długości N. Dla każdego z otrzymanych podciągów liczymy wybranym testem p-wartość. Następnie ciąg p-wartości testujemy ponownie, podobnie jak we wcześniejszych testach ciągów liczb - w przypadku second-level testing zwykle używamy wtedy jako drugiego testu χ^2. Ostatecznie otrzymujemy znów jedną p-wartość, która sugeruje, czy uzyskane p-wartości były także rozłożone losowo.

5 Wyniki

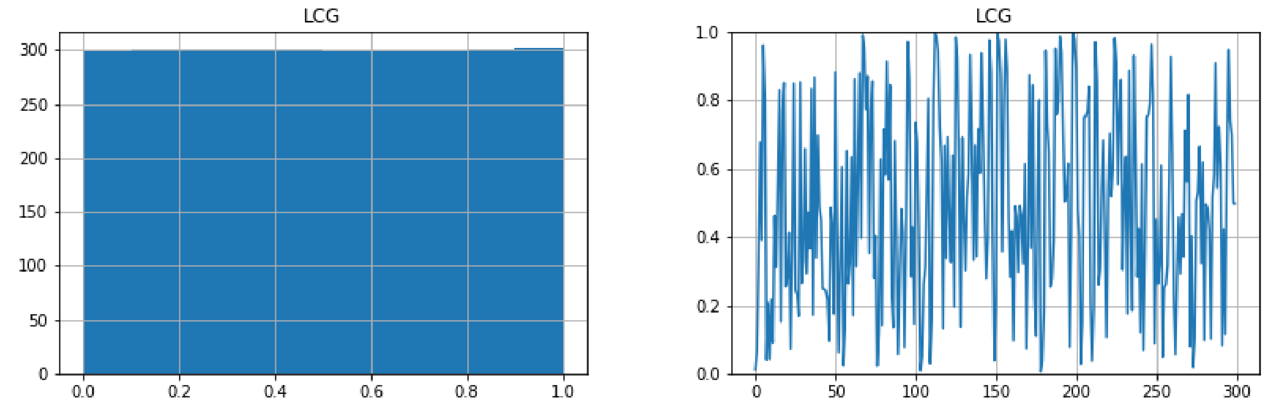
Zobaczymy jak prezentują się histogramy i wykresy dla poszczególnych generatorów. W każdym histogramie rozważamy ciągi o dobranej przez nas długości, a dla każdego wykresu ograniczamy sekwencję do pierwszych 300 liczb, aby zachować przejrzystość prezentowanych wyników.

5.1 LCG



Rysunek 1: $LCG(13, 1, 5)$

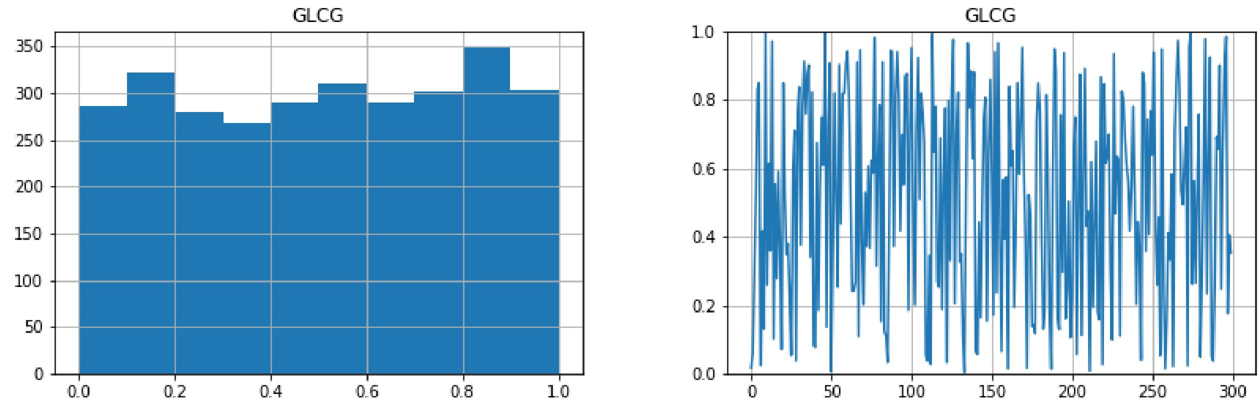
Słupki histogramu nie różnią się co sugeruje, że mamy do czynienia z rozkładem jednostajnym. Problem w tym, że wyniki przedstawione na wykresie są cykliczne. Ze względu na krótki okres powyższego generatora zauważamy pewien wzorzec w wygenerowanej ścieżce.



Rysunek 2: $LCG(2^{10}, 5, 7)$

Dla innych parametrów otrzymany histogram jest podobny do poprzedniego i ponownie przypomina prostokąt. Różnicę zauważamy w wykresie, dla którego tym razem liczby nie utworzyły wyraźnego wzoru. Wobec tego, wygenerowana sekwencja wygląda na jednostajną i niezależną.

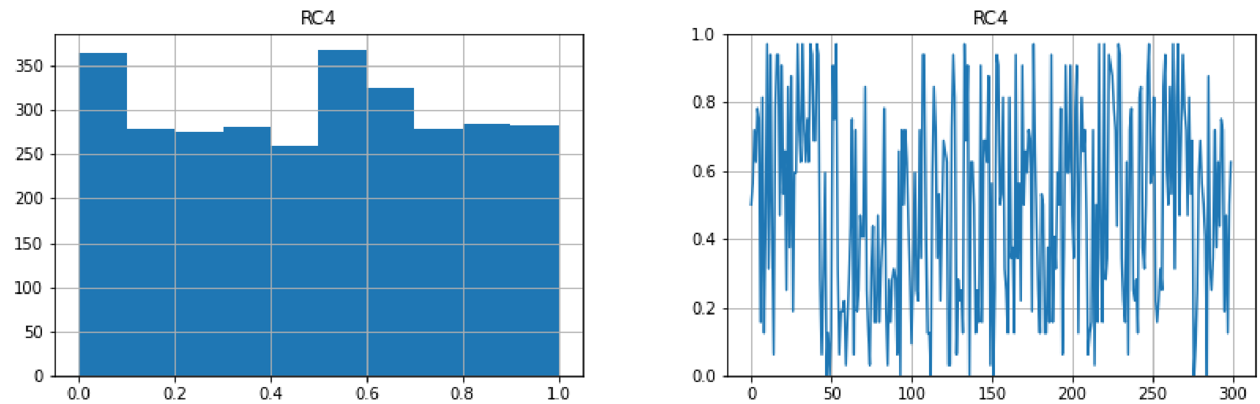
5.2 GLCG



Rysunek 3: $GLCG(2^{13}, 1, 5)$

Tutaj histogram wygląda gorzej - zauważamy różnice w wysokości słupków. Wygenerowana sekwencja liczb nie rozkłada się jednostajnie i wydaje się być niezależna. Na wykresie nie widać konkretnie zarysowanej tendencji.

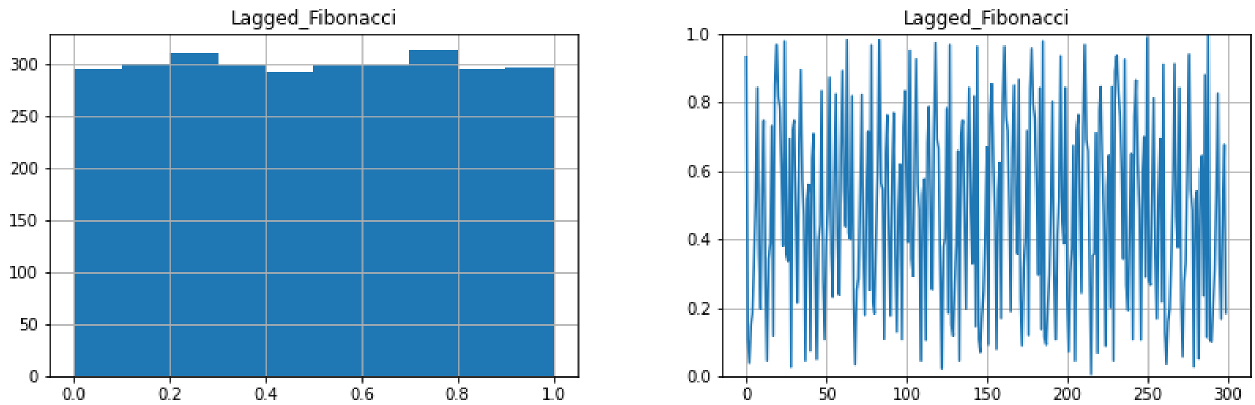
5.3 RC4(32)



Rysunek 4: $RC4(32, \{0, \ldots, 31\})$

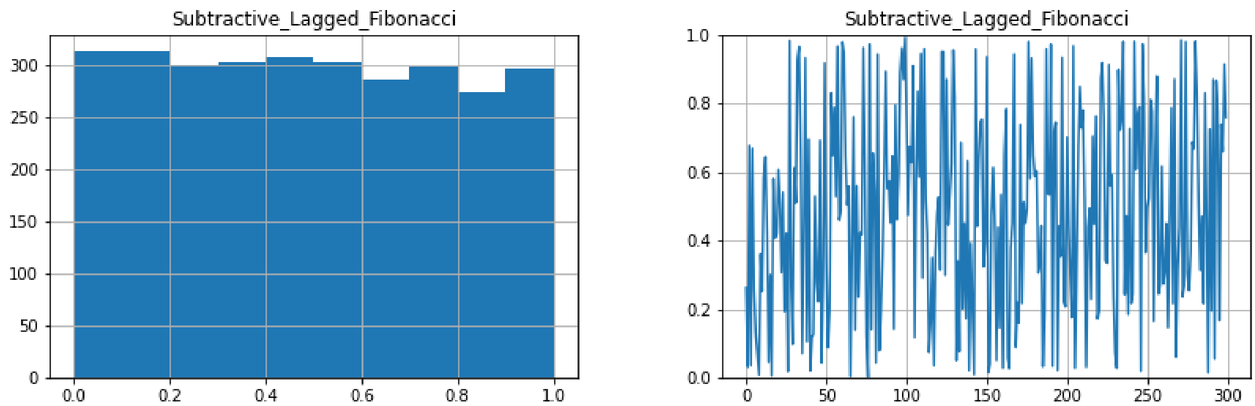
Histogram odbiega od idealnego, co wynikać może z niewłaściwego doboru kluczy. Otrzymana ścieżka nie ma wzorca.

5.4 Lagged Fibonacci generator



Rysunek 5: $LFG(2^{10}, 1, 2)$

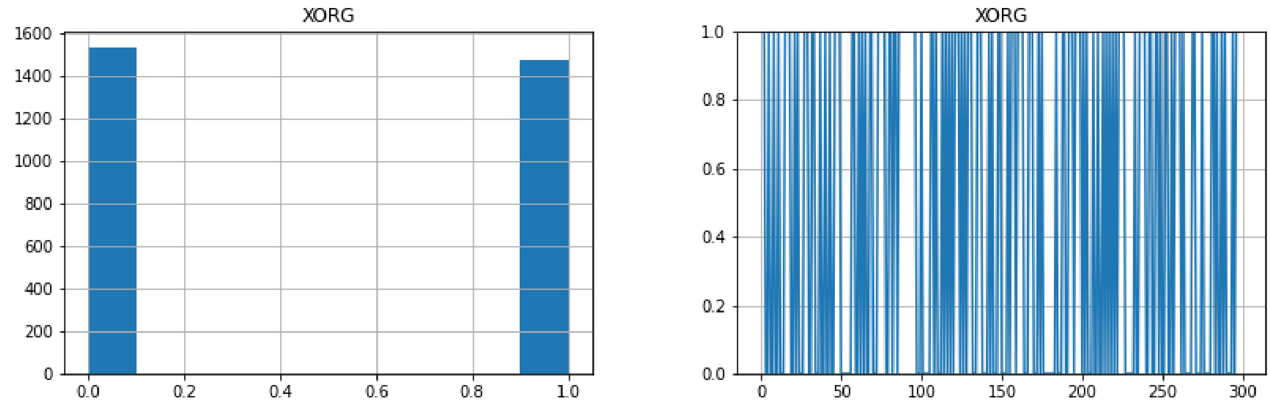
Otrzymane wyniki są zgodne z naszymi oczekiwaniami - wyglądają na losowe i jednostajne, pomimo drobnych różnic wysokości słupków.



Rysunek 6: $SLFG(2^{30}, 100, 37)$

Ponownie zauważamy drobne odstępstwa w histogramie i niezależność wygenerowanych liczb.

5.5 Exclusive OR Generator



Rysunek 7: *XORG*

W tym przypadku generowane są tylko zera lub jedynki, co widać na otrzymanym histogramie. W wygenerowanej ścieżce liczby nie utworzyły żadnego wzoru.

5.6 $\Pi, e, \sqrt{2}$

Zestawienie wyników dla liczb niewymiernych

	Irrational number	RUNS	FM	CHI2	KS	second level RUNS	second level FM	second level CHI2	second level KS
0	pi	0.422674	0.613721	0.746874	0.554221	0.759756	0.554420	0.897763	0.678686
1	e	0.610927	0.926876	0.594775	0.521477	0.021999	0.834308	0.191687	0.494392
2	sqrt2	0.279069	0.817751	0.111701	0.310881	0.978072	0.514124	0.437274	0.366918

Dla *Runs test* oraz *Frequency Monobit test* uzyskane p-wartości są w miarę duże, w związku z czym nie mamy podstaw do odrzucenia hipotezy zerowej, mówiącej o braku losowości rozwinięć binarnych badanych liczb niewymiernych. Dla testu χ^2 oraz *Kołmogorowa-Smirnova* p-wartości są zdecydowanie większe od przyjętego poziomu istotności $\alpha = 0.01$ zatem możemy uznać, iż obserwacje pochodzą z rozkładu jednostajnego. Owe spostrzeżenia dotyczą również *second-level testing*.

6 Porównanie

Na podstawie wykonanych testów statystycznych zbadamy wybrane generatory i porównamy je poprzez zestawienie odpowiednich p-wartości.

6.1 Testy generatorów

Wyniki dla ciągu o długości 100

	generator	CHI2	KS	RUNS	FM
0	LCG(13, 1, 5)	0.122325	0.531679	0.091938	0.870568
1	LCG(2^10, 3, 7)	0.834308	0.480274	0.007944	0.800282
2	LCG(2^10, 5, 7)	0.851383	0.680495	0.713641	0.527089
3	GLCG(2^10, (3, 7, 68))	0.983453	0.965181	0.977183	0.026857
4	GLCG(2^13-1, (3, 7, 68))	0.924076	0.986711	0.827991	0.571608
5	Lagged Fibonacci(2^10, 1, 2)	0.699313	0.758658	0.654634	0.375921
6	Lagged Fibonacci(2^10, 5, 12)	0.514124	0.554344	0.206077	0.184126
7	Lagged Fibonacci(2^30, 100, 37)	0.514124	0.766764	0.715885	0.144127
8	Subs Lag Fibonacci(2^10, 5, 12)	0.657933	0.899192	0.038082	0.899343
9	Subs Lag Fibonacci(2^30, 100, 37)	0.249284	0.078423	0.598871	0.770196
10	RC4(32, {0,...,31})	0.350485	0.198081	0.530285	0.858028
11	RC4(32, {15, 10, 21, 13 ,24})	0.983453	0.895555	0.535663	0.179712
12	XORG	0.759756	0.189280	0.231510	0.841481

Dla ciągu długości 100 każdy generator (poza $LCG(2^{10}, 3, 7)$ dla testu *Runs*) przechodzi każdy z przeprowadzonych przez nas testów.

Wyniki dla ciągu o długości 1000

	generator	CHI2	KS	RUNS	FM
0	LCG(13, 1, 5)	0.000000	0.000014	1.090012e-07	9.862868e-01
1	LCG(2^10, 3, 7)	0.999996	1.000000	2.285921e-23	8.728811e-01
2	LCG(2^10, 5, 7)	1.000000	1.000000	7.188201e-01	9.521556e-01
3	GLCG(2^10, (3, 7, 68))	0.998169	0.991017	0.000000e+00	4.704481e-11
4	GLCG(2^13-1, (3, 7, 68))	0.877083	0.561450	8.278778e-01	1.258271e-04
5	Lagged Fibonacci(2^10, 1, 2)	0.867692	0.691580	0.000000e+00	7.274494e-10
6	Lagged Fibonacci(2^10, 5, 12)	0.223648	0.634569	2.499282e-01	4.999579e-02
7	Lagged Fibonacci(2^30, 100, 37)	0.848027	0.601450	3.312205e-01	5.674682e-02
8	Subs Lag Fibonacci(2^10, 5, 12)	0.707513	0.283399	2.441494e-01	3.270861e-01
9	Subs Lag Fibonacci(2^30, 100, 37)	0.002428	0.396478	6.178156e-01	8.173613e-01
10	RC4(32, {0,...,31})	0.016149	0.027737	6.465415e-01	5.153446e-01
11	RC4(32, {15, 10, 21, 13 ,24})	0.029011	0.144373	8.873002e-02	1.620954e-02
12	XORG	0.090936	0.210607	8.002820e-01	1.000000e+00

Dla dłuższego ciągu generator $LCG(13, 1, 5)$ wypada znacznie gorzej niż poprzednio. Również w przypadku $GLCG(2^{10}, (3, 7, 68))$ i *Lagged Fibonacci*$(2^{10}, 1, 2)$ p-wartości uzyskane w *Runs test* oraz *Frequency Monobit test* są bardzo małe.

6.2 Second level testing

Ciąg długości 100 podzielony na 10 podciągów

	generator	result CHI2	result KS	result FM	result RUNS
0	LCG(13, 1, 5)	0.000040	6.164291e-12	3.250256e-09	0.000954
1	LCG(2^10, 3, 7)	0.000199	3.504852e-01	7.399183e-01	0.035174
2	LCG(2^10, 5, 7)	0.000954	3.517354e-02	2.133093e-01	0.534146
3	GLCG(2^10, (3, 7, 68))	0.213309	5.341462e-01	2.133093e-01	0.213309
4	GLCG(2^13-1, (3, 7, 68))	0.008879	5.341462e-01	5.341462e-01	0.534146
5	Lagged Fibonacci(2^10, 1, 2)	0.066882	1.223252e-01	5.341462e-01	0.534146
6	Lagged Fibonacci(2^10, 5, 12)	0.534146	1.223252e-01	2.133093e-01	0.534146
7	Lagged Fibonacci(2^30, 100, 37)	0.004301	5.341462e-01	9.114125e-01	0.122325
8	Subs Lag Fibonacci(2^10, 5, 12)	0.066882	5.341462e-01	5.341462e-01	0.534146
9	Subs Lag Fibonacci(2^30, 100, 37)	0.350485	5.341462e-01	1.223252e-01	0.122325
10	RC4(32, {0,...,31})	0.008879	5.341462e-01	6.688159e-02	0.534146
11	RC4(32, {15, 10, 21, 13 ,24})	0.017912	3.504852e-01	5.341462e-01	0.911413
12	XORG	0.122325	7.399183e-01	3.964659e-05	0.122325

Ogólnie, najgorzej wypada generator LCG, który dla $(13, 1, 5)$ nie przechodzi żadnego testu, a dla parametrów $(2^{10}, z, 7)$ - testu χ^2. W pozostałych przypadkach nie mamy podstaw do odrzucenia H_0 na poziomie $\alpha = 0.01$, chociaż nieco niższe wyniki od przyjętego α zauważamy dla kilku generatorów w teście χ^2. Dodatkowo, p-wartość w teście *Frequency Monobit* dla $XORG$ jest niewielka.

Ciąg długości $1000 \cdot 2^{10}$ podzielony na 1000 podciągów

	generator	result CHI2	result KS	result FM	result RUNS
0	LCG(13, 1, 5)	0.000000	0.000000	0.000000	0.000000
1	LCG(2^10, 3, 7)	0.000000	0.000000	0.000000	0.000000
2	LCG(2^10, 5, 7)	0.000000	0.000000	0.000000	0.000000
3	GLCG(2^10, (3, 7, 68))	0.000000	0.000000	0.000000	0.000000
4	GLCG(2^13-1, (3, 7, 68))	0.697257	0.680755	0.000000	0.000000
5	Lagged Fibonacci(2^10, 1, 2)	0.000000	0.000000	0.000000	0.000000
6	Lagged Fibonacci(2^10, 5, 12)	0.856359	0.024688	0.000449	0.000021
7	Lagged Fibonacci(2^30, 100, 37)	0.589341	0.994944	0.723804	0.587274
8	Subs Lag Fibonacci(2^10, 5, 12)	0.061260	0.371941	0.000209	0.502247
9	Subs Lag Fibonacci(2^30, 100, 37)	0.053286	0.018668	0.870856	0.459717
10	RC4(32, {0,...,31})	0.000000	0.000000	0.827279	0.705466
11	RC4(32, {15, 10, 21, 13 ,24})	0.000000	0.000000	0.469232	0.506194
12	XORG	0.073417	0.628790	0.000109	0.336111

W tym przypadku tylko *Lagged Fibonacci* oraz *Subtractive Lagged Fibonacci* dla $(2^{30}, 100, 37)$ spełniają wszystkie testy. Zatem owe generatory dla tak przyjętych parametrów są najlepsze. Dodatkowo, generator $RC4$ z różnym doborem kluczy zdaje się być losowy.

Literatura

[1] Tomasz Rolski, *Symulacje stochastyczne i teoria Monte Carlo*, `http://www.math.uni.wroc.pl/~rolski/Zajecia/sym.pdf`

[2] NIST, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, `http://www.math.uni.wroc.pl/~lorek/teaching/files/nistspecialpublication800-22r1a.pdf`

[3] Projekt, *Introduction to simulations and Monte Carlo methods. Project nr 1*, `https://moodle.math.uni.wroc.pl/pluginfile.php/17083/mod_resource/content/2/2022_monte_carlo_project1.pdf`