Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Matematyczny

Grzegorz Łoś

Zastosowanie błądzenia przypadkowego do testowania generatorów liczb pseudolosowych

Praca magisterska
napisana pod kierunkiem
dr. Pawła Lorka

Wrocław, 31 sierpnia 2015

Spis treści

Wprowadzenie

Wiele współczesnych technologii opiera się na randomizacji. W informatyce losowość pojawia się na każdym kroku i często nie zdajemy sprawy jak bardzo jesteśmy od niej uzależnieni. Programiści korzystają z niej na co dzień, często zupełnie nieświadomie, na przykład używając bibliotecznych implementacji algorytmu quicksort lub tablic haszujących. Randomizacja jest niezbędnym elementem w wielu innych specjalistycznych dziedzinach. Przykładowo w finansach ważną rolę odgrywają metody Monte Carlo polegające na wielokrotnej symulacji rozwoju rynku. Metody optymalizacji oparte o metaheurystyki lub algorytmy ewolucyjne nie miałyby bez losowości racji bytu.

W podanych powyżej przykładach drobne wady generatora liczb pseudolosowych (GLP), na których oparte są wspomniane metody, mogą obniżyć efektywność działania lub dokładność wyników, ale nie rujnują algorytmów całkowicie. Są jednak dziedziny, w których jakość generatora ma zasadnicze znaczenie. Dobrym przykładem jest kryptografia. Zauważalne odstępstwa od losowości mogą istotnie zwiększyć szanse złamania protokołu kryptograficznego, odkrycia klucza prywatnego, itp. Wynika stąd potrzeba zidentyfikowania tych GLP, na których można polegać.

Początki analizy generatorów sięgają lat pięćdziesiątych. Już wtedy John von Neumann zauważył, że trudniej jest testować ciągi pseudolosowych liczb niż je produkować. Od tego czasu badacze nieustannie udoskonalają narzędzia służące testowaniu GLP. To zagadnienie poruszał już Donald Knuth w swoim dziele *Sztuka programowania*. Zaproponował on kilka testów statystycznych traktujących liczby otrzymane z GLP jak zmienne losowe. Przy założeniu ich jednostajności i niezależności, pewne funkcje tych zmiennych powinny mieć znane rozkłady, co sprawdzamy np. testem chi-kwadrat. To podejście było następnie dalej rozwijane – znane paczki testów opracowali George Marsaglia (w 1995), Pierre L'Ecuyer i Richard Simard (w 2007), a obecnie ważną rolę odgrywa zestaw rozwijany przez amerykańską agencję National Institute of Standards and Technology.

W niniejszej pracy przedstawiamy nowe metody testowania GLP: test arcusa sinusa i test iterowanego logarytmu. Wymagają one traktowania wyjścia generatora jako strumienia binarnych danych. Rozstrzygnięcie czy dany GLP jest wystarczająco solidny sprowadza się do odpowiedzi na pytanie czy sekwencja otrzymana z GLP jest nieodróżnialna od ciągu prawdziwie losowego. Interpretując wygenerowane bity jako +1 oraz -1, możemy łatwo zobaczyć, że wyjście GLP odpowiada realizacji błądzenia przypadkowego. Ten proces stochastyczny jest dobrze zbadany i opisany w literaturze, stąd znamy wiele jego własności. Testowanie GLP polega na sprawdzeniu czy jego wyjście również je posiada.

Opracowana metoda silnie polega na teoretycznych własnościach błądzenia przypadkowego,

dlatego w części 1 przedstawiamy niezbędne matematyczne podstawy. Przypominamy definicję błądzenia losowego oraz przytaczamy dwa kluczowe twierdzenia. Pierwsze z nich, prawo iterowanego logarytmu, oszacowuje wielkość odchyleń od zera jakich należy spodziewać się obserwując proces błądzenia. Drugie, prawo arcusa sinusa, mówi, że jest bardziej prawdopodobne, iż błądzenie przypadkowe zdecydowaną większość czasu spędzi nad osią OX, niż że rozkład czasu spędzony po obu stronach osi będzie w miarę równy.

Część 2 poświęcona jest przybliżeniu obiektu naszego zainteresowania. Precyzujemy pojęcie generatora liczb pseudolosowych. Przedstawiamy kilka rodzajów GLP, np. generatory wykorzystujące kongruencje liniowe. GLP zwracają nam pseudolosowe liczby, a do testów arcusa sinusa i iterowanego logarytmu potrzebujemy pseudolosowych sekwencji bitów. Dlatego odnotowujemy kilka uwag dotyczących interpretowania wyjścia generatora jako ciągu zerojedynkowego. Następnie przywołujemy kilka popularnych metod testowania GLP.

W części 3 opisujemy dokładniej jak wykorzystać przytoczone własności błądzenia przypadkowego do testowania GLP. Autorzy [10] zauważyli użyteczność prawa iterowanego logarytmu do testowania generatorów. W niniejszej pracy proponujemy podobną metodę, opartą o prawo arcusa sinusa. Z grubsza rzecz biorąc, postępujemy tak jak w statystyce matematycznej, choć obserwacje są dość nietypowe, bo są nimi długie ciągi zerojedynkowe. Wykorzystujemy GLP, by otrzymać m sekwencji bitów. Każdą z nich interpretujemy jako realizację błądzenia przypadkowego i na jej podstawie obliczamy pewną wartość, którą będziemy nazywać charakterystyką ciągu. W ten sposób dostajemy empiryczny rozkład tej charakterystyki. Natomiast dzięki twierdzeniom z części 1 znamy jej rozkład teoretyczny. Wykonując test χ^2 sprawdzamy zgodność tych rozkładów. Alternatywnie możemy skorzystać ze znanych funkcji odległości miar (rozkład prawdopodobieństwa nie jest niczym innym jak miarą zdefiniowaną na prostej rzeczywistej). Jeśli wartość statystyki testowej lub odległość miary teoretycznej i empirycznej jest duża, to możemy powiedzieć, że GLP niezbyt dobrze imituje losowość, w przeciwnym razie nie ma podstaw by go zdyskredytować.

Metoda opisana w części 3 została zaprogramowana w języku Julia. W części 4 przedstawiamy tę implementację. Omawiamy różne praktyczne aspekty, które wpłynęły na architekturę programu. Prezentujemy jak użyć wykonanego narzędzia do przetestowania własnych GLP. Następnie przedstawiamy wyniki testowania kilku stosowanych generatorów. Sprawdzamy jakość funkcji `rand` z języka C, w kilku popularnych środowiskach. Inne przetestowane generatory to RANDU, Minstd, CMRG oraz Mersenne Twister. Warto zwrócić uwagę również na ostatni przykład, wprawdzie nieco sztuczny, jednak obrazujący zaletę testów arcusa sinusa i iterowanego logarytmu w stosunku do podejścia stosowanego przykładowo przez NIST Test Suite.

CZĘŚĆ I

Błądzenie przypadkowe

Jak napisaliśmy we wprowadzeniu, własności błądzenia przypadkowego (inaczej: losowego) będą kluczowe dla testowania GLP. Zebrane w tej części wiadomości opracowane są na podstawie [2]. Większość oznaczeń jest również wzorowane na tej książce.

Wyjście generowane przez GLP zawsze można traktować jako ciąg zerojedynkowy. Dlatego ważnym pojęciem będzie dla nas ciąg niezależnych prób Bernoulliego (inaczej: proces Bernoulliego) $(B_i)_{i\in\mathbb{N}}$. Dla ustalonego $p \in [0,1]$ oznaczamy w ten sposób ciąg niezależnych zmiennych losowych o jednakowym rozkładzie, taki że

$$\mathbb{P}(B_1 = 1) = p = 1 - \mathbb{P}(B_1 = 0).$$

Możemy postrzegać i-ty bit wygenerowany przez GLP jako wynik i-tej próby Bernoulliego. Dobry generator powinien z takim samym prawdopodobieństwem losować 0 oraz 1, dlatego ograniczymy się do przypadku $p = \frac{1}{2}$.

Często będzie nam wygodniej posługiwać się ciągiem prób $(X_i)_{i\in\mathbb{N}}$, który przyjmuje wartości -1 zamiast 0, czyli

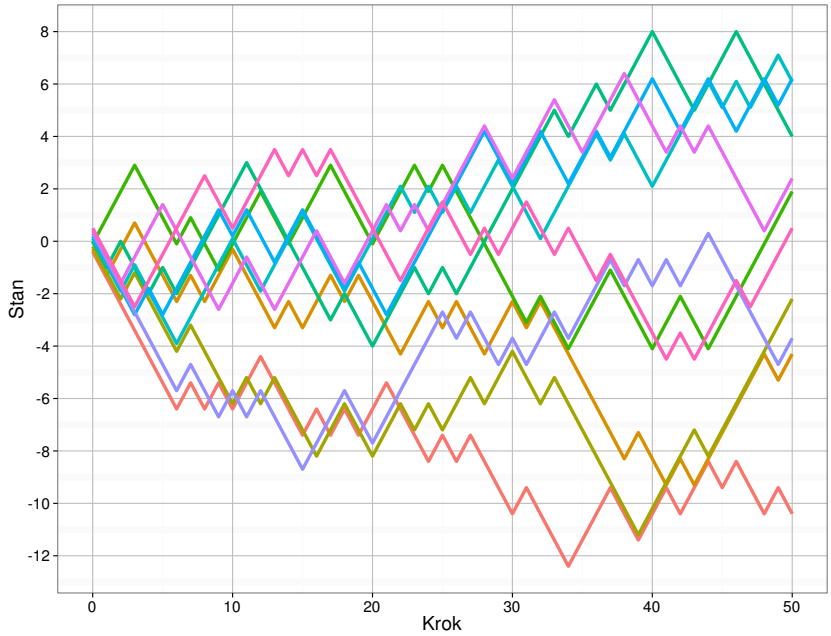
$$X_i \stackrel{D}{=} 2B_i - 1.$$

Ponadto oznaczmy

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

Tak zdefiniowany proces $(S_i)_{i\in\mathbb{N}}$ jest nazywany błądzeniem przypadkowym. Ciąg ten w każdym kolejnym kroku zmienia swoją wartość o 1 lub -1. Czasem wygodnie jest go postrzegać jako wynik następującej gry. Dwóch graczy rzuca idealną monetą. Jeśli wypada orzeł, to pierwszy gracz otrzymuję złotówkę od drugiego, w przeciwnym przypadku pierwszy płaci złotówkę drugiemu. Proces S przedstawia zysk ustalonego gracza.

Sporo miejsca w rachunku prawdopodobieństwa poświęcono badaniu własności błądzenia przypadkowego, z których dwie omawiamy poniżej. Ideą testów, które przedstawiamy w części 3 jest sprawdzanie czy wyjście GLP zachowuje się tak jak to wynika z praw rachunku prawdopodobieństwa.



Rysunek 1.1: Przykładowe trajektorie procesu S.

1.1 Prawo iterowanego logarytmu

Jest jasne, że $|S_n| \leqslant n$. Można się jednak domyślać, choćby na podstawie Rysunku 1.1, że duże wartości $|S_n|$ są jednak bardzo mało prawdopodobne i w praktyce z dużym prawdopodobieństwem wartości S_n znajdą się w znacznie węższym przedziale niż $[-n, n]$. Słabe i mocne prawo wielkich liczb (SPWL i MPWL) mówią nam, że

$$\frac{S_n}{n} \xrightarrow{\mathbb{P}} 0, \text{ a nawet } \frac{S_n}{n} \xrightarrow{p.n.} 0.$$

Jest więc jasne, że odchylenia procesu S od zera rosną znacznie wolniej niż liniowo. Z drugiej strony centralne twierdzenie graniczne (CTG) mówi nam, że $\frac{S_n}{\sqrt{n}} \xrightarrow{D} \mathcal{N}(0,1)$ co jest w pewnym sensie oszacowaniem fluktuacji S_n od dołu – będą one wychodzić poza przedział $[-\sqrt{n}, \sqrt{n}]$, mamy bowiem

Fakt 1.1. *Błądzenie przypadkowe S_n z prawdopodobieństwem 1 spełnia*

$$\limsup_{n\to\infty} \frac{S_n}{\sqrt{n}} = \infty.$$

Dowód. Z prawa 0-1 Kołmogorowa wynika, że dla dowolnego ciągu zmiennych losowych (X_i) i.i.d., zdarzenia typu $\left\{\limsup_{n\to\infty} X_n > M\right\}$ mają prawdopodobieństwo równe 0 lub 1 (patrz [4],

§7.2, zadanie 1). Weźmy dowolnie duże M. Mamy

$$\begin{aligned}\mathbb{P}\left(\limsup_{n\to\infty}\frac{S_n}{\sqrt{n}}>M\right)&=\mathbb{P}\left(\bigcap_{n=1}^{\infty}\bigcup_{k\geqslant n}\left\{\frac{S_k}{\sqrt{k}}>M\right\}\right)\\&=\lim_{n\to\infty}\mathbb{P}\left(\bigcup_{k\geqslant n}\left\{\frac{S_k}{\sqrt{k}}>M\right\}\right)\\&\geqslant\lim_{n\to\infty}\mathbb{P}\left(\frac{S_n}{\sqrt{n}}>M\right)\\&=1-\Phi(M)>0.\end{aligned}$$

Czyli $\mathbb{P}\left(\limsup_{n\to\infty}\frac{S_n}{\sqrt{n}}>M\right)=1$, co wobec dowolności M oznacza, że

$$\mathbb{P}\left(\limsup_{n\to\infty}\frac{S_n}{\sqrt{n}}=\infty\right)=1.$$

□

Okazuje się, że fluktuacje S można oszacować precyzyjniej, mówi o tym

Twierdzenie 1.2 (**Prawo iterowanego logarytmu**). *Błądzenie przypadkowe S_n z prawdopodobieństwem 1 spełnia*

$$\limsup_{n\to\infty}\frac{S_n}{\sqrt{2n\log\log n}}=1.$$

Dowód można znaleźć w [2], rozdział VIII, §5. Oczywiście ze względu na symetrię mamy analogiczne własności do Faktu 1.1 i Twierdzenia 1.2 dla $\liminf$.

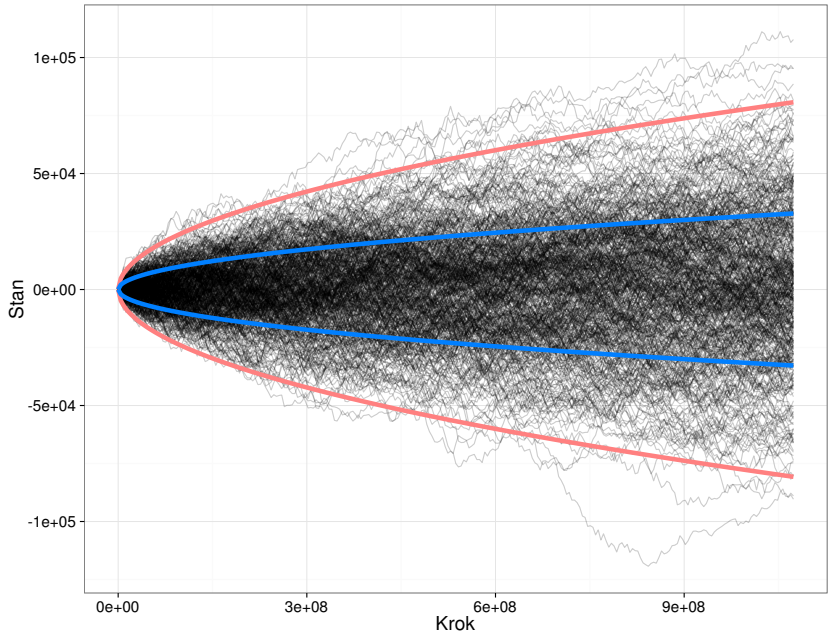
Jak widać n było zbyt dużym dzielnikiem, a $\sqrt{n}$ zbyt małym – odchylenia S_n od zera rosną proporcjonalnie do $\sqrt{n\log\log n}$. Można zatem powiedzieć, że prawo iterowanego logarytmu (PIL) „działa pomiędzy" prawem wielkich liczb i centralnym twierdzeniem granicznym. Te trzy twierdzenia dają nam własności błądzenia przypadkowego, które zebrano w Tabeli 1.1.

Przyjrzyjmy się Rysunkowi 1.2. Widać, że funkcja $\sqrt{2n\log\log n}$ z grubsza odpowiada fluktuacjom procesu S_n. Można jednak zauważyć, że kilka trajektorii po około miliardzie kroków ciągle nie mieści się w przedziale $[-\sqrt{2n\log\log n},\sqrt{2n\log\log n}]$. Prawo iterowanego logarytmu mówimy nam, że dla odpowiednio dużych n trajektorie nie będą wykraczać poza ten zakres z prawdopodobieństwem 1. Wniosek jaki możemy wyciągnąć z tego obrazka jest taki, że mowa tu o naprawdę olbrzymich wartościach n.

Choć nie będzie przydatna w dalszej części pracy, jeszcze jedna ciekawa własność narzuca się by o niej wspomnieć. Niech $S_n^{lil}=\frac{S_n}{\sqrt{2n\log\log n}}$. Z PIL wynika, że wielkość S_n^{lil} nie zbiega punktowo do żadnej stałej. Zachodzi natomiast zbieżność do 0 według prawdopodobieństwa. Ustalmy więc dowolnie małe $\varepsilon>0$ i zastanówmy się jak często S_n^{lil} opuści epsilonowy pasek wokół zera. Możemy przyjąć $p<1$ dowolnie bliskie jedności, a mimo to dla prawie wszystkich n możemy powiedzieć, że z prawdopodobieństwem p wielkość S_n^{lil} nie wyjdzie poza przedział $(-\varepsilon,\varepsilon)$. Tymczasem PIL równocześnie mówi nam, że ten epsilonowy pasek opuścimy nieskończenie wiele razy. Ta niesamowita, pozorna sprzeczność pokazuje jak bardzo nasza intuicja zawodzi, gdy myślimy o zjawiskach zachodzących w nieskończoności.

Tabela 1.1: Wnioski dotyczące błądzenia przypadkowego wynikające ze znanych twierdzeń.

	Zbieżność według prawdop.	Zbieżność prawie na pewno	Wartość limes superior prawie na pewno	Wartość limes inferior prawie na pewno
PWL	$\frac{S_n}{n} \xrightarrow{\mathbb{P}} 0$	$\frac{S_n}{n} \xrightarrow{p.n.} 0$	$\limsup_{n\to\infty} \frac{S_n}{n} = 0$	$\liminf_{n\to\infty} \frac{S_n}{n} = 0$
PIL	$\frac{S_n}{\sqrt{2n\log\log n}} \xrightarrow{\mathbb{P}} 0$	$\frac{S_n}{\sqrt{2n\log\log n}} \overset{p.n.}{\nrightarrow} 0$	$\limsup_{n\to\infty} \frac{S_n}{\sqrt{2n\log\log n}} = 1$	$\liminf_{n\to\infty} \frac{S_n}{\sqrt{2n\log\log n}} = -1$
CTG	$\forall x\ \frac{S_n}{\sqrt{n}} \overset{\mathbb{P}}{\nrightarrow} x$	$\forall x\ \frac{S_n}{\sqrt{n}} \overset{p.n.}{\nrightarrow} x$	$\limsup_{n\to\infty} \frac{S_n}{\sqrt{n}} = \infty$	$\liminf_{n\to\infty} \frac{S_n}{\sqrt{n}} = -\infty$



Rysunek 1.2: Ilustracja prawa iterowanego logarytmu. Przedstawia ona 500 trajektorii błądzenia losowego, długości 2^{30}. Im ciemniejszy jest obszar wykresu, tym większe jest w nim zagęszczenie trajektorii. Niebieska krzywa to wykresy funkcji $\sqrt{x}$ oraz $-\sqrt{x}$, zaś czerwona funkcji $\sqrt{2x\log\log x}$ oraz $-\sqrt{2x\log\log x}$.

1.2 Prawo arcusa sinusa

Kolejna własność błądzenia przypadkowego, którą postaramy się wykorzystać do testowania GLP jest znana jako prawo arcusa sinusa. Odpowiada ono na pytanie przez jaką frakcję czasu

ustalony gracz będzie na prowadzeniu. Spodziewalibyśmy się, że w przypadku bardzo długiej gry, obaj gracze będą na prowadzeniu przez mniej więcej tyle samo czasu. Jednak pokażemy, że również w tym przypadku nasza intuicja płata nam figla.

Powiemy, że bilans gry w k-tym kroku ($k \geqslant 1$) był dodatni, jeżeli $S_k > 0$ lub $S_{k-1} > 0$. Pomijamy tu remisy przyjmując, że w przypadku wystąpienia równej liczby reszek i orłów przewagę ma ten, kto miał ją w poprzedniej chwili. Geometrycznie oznacza to, że odcinek wykresu błądzenia losowego przebiegający pomiędzy odciętymi $k-1$ oraz k, musi znajdować się nad osią x-ów.

Wprowadźmy następujące oznaczenia:

- U_n – zdarzenie, że w n-tym kroku nastąpił powrót do zera,
- F_n – zdarzenie, ze w n-tym kroku nastąpił *pierwszy* powrót do zera,
- $u_n = \mathbb{P}(U_n)$, $f_n = \mathbb{P}(F_n)$.
- $p_{k,n}$ – prawdopodobieństwo, że przez k spośród pierwszych n kroków gry, bilans był dodatni.

Łatwo zauważyć, że powrót do zera może nastąpić tylko w parzystym kroku, zatem

$$\forall n \in \mathbb{N} \quad u_{2n-1} = f_{2n-1} = 0,$$

$$\forall k, n \in \mathbb{N} \quad p_{2k-1,2n} = 0,$$

Ponadto przyjmujemy, że $p_{0,0} = u_0 = 1$. Zachodzi również

Lemat 1.3. *Dla każdego $n \in \mathbb{N}$ spełnione są poniższe tożsamości:*

$$u_{2n} = \binom{2n}{n} 2^{-2n} \tag{1.1}$$

$$u_{2n} = \sum_{r=1}^{n} f_{2r} u_{2n-2r} \tag{1.2}$$

$$f_{2n} = \frac{1}{2n} u_{2n-2} \tag{1.3}$$

$$f_{2n} = u_{2n-2} - u_{2n} \tag{1.4}$$

Dowód. Wzór (1.1) wynika stąd, że wszystkich dróg długości $2n$ jest 2^{2n}, a drogi wracające na końcu do zera odpowiadają ustawieniu n orłów i n reszek na $2n$ miejscach – co robimy na $\binom{2n}{n}$ sposobów.

Tożsamość (1.2) wynika wprost ze wzoru na prawdopodobieństwo całkowite:

$$u_{2n} = \mathbb{P}(U_{2n}) = \sum_{r=1}^{n} \mathbb{P}(U_{2n}|F_{2r})\mathbb{P}(F_{2r}) = \sum_{r=1}^{n} \mathbb{P}(U_{2n-2r})\mathbb{P}(F_{2r}) = \sum_{r=1}^{n} u_{2n-2r} f_{2r}$$

Dla dowodu (1.3) wprowadźmy dodatkowe oznaczenia:

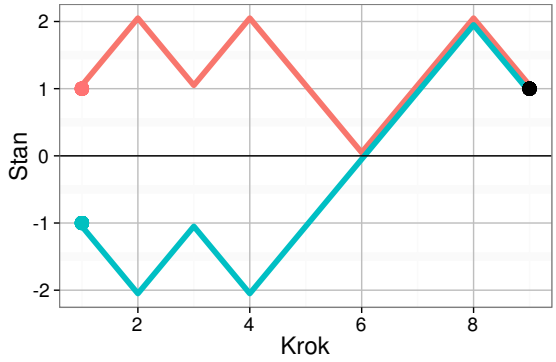
- $N_n(a, b)$ – liczba ścieżek od stanu a do stanu b w n krokach,

- $N_n^{\neq 0}(a,b)$ – jak $N_n(a,b)$, ale ścieżki nie mogą dotykać 0 (za wyjątkiem co najwyżej końców),
- $N_n^{=0}(a,b)$ – jak $N_n(a,b)$, ale ścieżki muszą dotknąć lub przeciąć 0.

Łatwo zauważyć, że $N_n(a,b) = \binom{n}{(n+b-a)/2}$ oraz $N_n(a,b) = N_n^{\neq 0}(a,b) + N_n^{=0}(a,b)$. Wartość f_{2n} to oczywiście stosunek $N_{2n}^{\neq 0}(0,0)$ do liczby wszystkich ścieżek od stanu 0 do stanu 0 w $2n$ krokach. Dlatego liczymy

$$N_{2n}^{\neq 0}(0,0) = N_{2n-1}^{\neq 0}(1,0) + N_{2n-1}^{\neq 0}(-1,0) = 2N_{2n-1}^{\neq 0}(1,0) = 2N_{2n-2}^{\neq 0}(1,1)$$

Patrząc na Rysunek 1.3 łatwo zauważyć, że $N_{2n-2}^{=0}(1,1) = N_{2n-2}(-1,1)$, jest to szczególny przypadek tzw. zasady odbicia. Zatem



Rysunek 1.3: Ilustracja faktu $N_n^{=0}(1,1) = N_n(-1,1)$. Łatwo zobaczyć jednoznaczną odpowiedniość między oboma rodzajami ścieżek. Aż do momentu pierwszego powrotu do zera ścieżka jednego rodzaju jest odbiciem symetrycznym względem osi odciętych ścieżki drugiego rodzaju, zaś dalej ścieżki się pokrywają.

$$\begin{aligned} N_{2n-2}^{\neq 0}(1,1) &= N_{2n-2}(1,1) - N_{2n-2}^{=0}(1,1) = N_{2n-2}(1,1) - N_{2n-2}(-1,1) \\ &= \binom{2n-2}{n-1} - \binom{2n-2}{n} = \binom{2n-2}{n-1} - \frac{n-1}{n}\binom{2n-2}{n-1} \\ &= \frac{1}{n}\binom{2n-2}{n-1} = \frac{2^{2n-2}}{n} u_{2n-2} \end{aligned}$$

Ostatecznie

$$f_{2n} = \frac{N_{2n}^{\neq 0}(0,0)}{2^{2n}} = \frac{2N_{2n-2}^{\neq 0}(1,1)}{2^{2n}} = \frac{\frac{2^{2n-1}}{n} u_{2n-2}}{2^{2n}} = \frac{u_{2n-2}}{2n}$$

Formuła (1.4) to prosta konsekwencja (1.1) i (1.3), bo

$$\begin{aligned} u_{2n-2} - u_{2n} &= u_{2n-2} - \binom{2n}{n} 2^{-2n} = u_{2n-2} - \binom{2n-2}{n-1} \frac{(2n-1)2n}{4n^2} 2^{-(2n-2)} = \\ &= u_{2n-2}\left(1 - \frac{(2n-1)}{2n}\right) = \frac{1}{2n} u_{2n-2} = f_{2n}. \end{aligned}$$

□

Tożsamości z Lematu 1.3 intensywnie wykorzystujemy w dowodzie następującego, kluczowego faktu.

Twierdzenie 1.4. *Dla wszystkich* $k, n \in \mathbb{N}$

$$p_{2k,2n} = u_{2k}u_{2n-2k} = \binom{2k}{k}\binom{2n-2k}{n-k}2^{-2n} \tag{1.5}$$

Dowód. Niech q_{2n} oznacza prawdopodobieństwo, że w pierwszych $2n$ krokach gry ani razu nie doszło do remisu. Wzór (1.4) daje nam

$$q_{2n} = 1 - f_2 - f_4 - \cdots - f_{2n} = 1 - (1 - u_2) - (u_2 - u_4) - \cdots - (u_{2n-2} - u_{2n}) = u_{2n}.$$

Udowodnimy teraz indukcyjnie, że

$$p_{0,2n} = u_{2n}. \tag{1.6}$$

Łatwo sprawdzić, że $p_{0,2} = \frac{1}{2} = u_2$. Załóżmy, że $p_{0,2\tilde{n}} = u_{2\tilde{n}}$ dla $\tilde{n} < n$. Zauważmy, że aby spędzić całą grę na minusie, musieliśmy w pierwszym kroku pójść w dół, co dzieje się z prawdopodobieństwem $\frac{1}{2}$. Dalej musiała zajść jedna z dwóch możliwości. Z prawdopodobieństwem q_{2n} mogliśmy ani razu nie wrócić do zera. Mogło się też zdarzyć, że dla pewnego r wróciliśmy do zera po raz pierwszy w kroku $2r$ (z prawdopodobieństwem f_{2r}), ale resztę czasu mimo tego spędziliśmy „pod kreską" (z prawdopodobieństwem $p_{0,2n-2r}$). Te rozważania, założenie indukcyjne oraz wzór (1.2) dają

$$\begin{aligned} p_{0,2n} &= \frac{1}{2}\left(q_{2n} + \sum_{r=1}^{n} f_{2r}p_{0,2n-2r}\right) = \frac{1}{2}\left(u_{2n} + \sum_{r=1}^{n} f_{2r}u_{2n-2r}\right) \\ &= \frac{1}{2}\left(u_{2n} + u_{2n}\right) = u_{2n}, \end{aligned}$$

co chcieliśmy pokazać.

Teraz uogólniamy ten wynik postępując również indukcyjnie. Twierdzenie 1.4 jest w oczywisty sposób prawdziwe dla $n = 0$. Załóżmy teraz, że dla wszystkich $\tilde{n} < n$ zachodzi $\forall 0 \leqslant k \leqslant \tilde{n}\ \ p_{2k,2\tilde{n}} = u_{2k}u_{2\tilde{n}-2k}$ i pokażemy, że $\forall 0 \leqslant k \leqslant n\ \ p_{2k,2n} = u_{2k}u_{2n-2k}$. Wiemy już, że teza jest prawdziwa dla $k = 0$ oraz $k = n$, gdyż

$$p_{2n,2n} = p_{0,2n} = u_{2n} = u_{2n}u_0.$$

Dlatego weźmy dowolne k, takie że $0 < k < n$. Aby zaszło rozważane zdarzenie, błądzenie musi przechodzić przez 0. Załóżmy, że pierwszy raz dzieje się to w pewnym punkcie $2r$. Jeżeli w pierwszym kroku poszliśmy w górę (co dzieje się z prawdopodobieństwem $\frac{1}{2}$), to po powrocie musimy spędzić „nad kreską" jeszcze $2k - 2r$ kroków, a szanse tego zdarzenia wynoszą $p_{2k-2r,2n-2r}$. W przeciwnym razie po powrocie ciągle musimy być na plusie przez $2k$ kroków, co zdarzy się z prawdopodobieństwem $p_{2k,2n-2r}$. Stąd

$$p_{2k,2n} = \frac{1}{2}\left(\sum_{r=1}^{k} f_{2r}p_{2k-2r,2n-2r} + \sum_{r=1}^{n-k} f_{2r}p_{2k,2n-2r}\right) = (\bigstar)$$

Z założenia indukcyjnego

$$p_{2k-2r,2n-2r} = u_{2k-2r}u_{2n-2r-(2k-2r)} = u_{2k-2r}u_{2n-2k}$$

oraz

$$p_{2k,2n-2r} = u_{2k}u_{2n-2r-2k},$$

zatem

$$\begin{aligned}(\bigstar) &= \frac{1}{2}\left(\sum_{r=1}^{k} f_{2r}u_{2k-2r}u_{2n-2k} + \sum_{r=1}^{n-k} f_{2r}u_{2k}u_{2n-2r-2k}\right)\\ &= \frac{1}{2}\left(u_{2n-2k}\sum_{r=1}^{k} f_{2r}u_{2k-2r} + u_{2k}\sum_{r=1}^{n-k} f_{2r}u_{2n-2r-2k}\right)\\ &= \frac{1}{2}\left(u_{2n-2k}u_{2k} + u_{2k}u_{2n-2k}\right) = u_{2k}u_{2n-2k},\end{aligned}$$

co było do okazania. Korzystając z (1.1) otrzymujemy tezę. □

Dzięki Twierdzeniu 1.4 możemy obliczać dokładne prawdopodobieństwa frakcji przewagi. Na Rysunku 1.4 przedstawiony jest ich rozkład dla $n = 20$. Widać wyraźnie, że równomierny podział czasu na przewagę jednego i drugiego gracza jest najmniej prawdopodobny. Najbardziej prawdopodobna jest dominacja jednego z graczy przez większość czasu. Przykładowo prawdopodobieństwo, że po 100 rzutach

- jeden z graczy wygrywa przez 90-100% czasu, wynosi 44%.
- jeden z graczy ani razu nie wyjdzie na prowadzenie, wynosi 16%.
- ustalony gracz będzie prowadził przez 40-60% czasu, wynosi 14%.

Wzór (1.5) jest dokładny, ale często nieporęczny. Spróbujmy znaleźć rozsądne przybliżenie. Zakładając $k \to \infty,\ n-k \to \infty$ i korzystając ze wzoru Stirlinga ($n! \approx \sqrt{2\pi n}\,(\frac{n}{e})^n$), dostajemy

$$\binom{2k}{k} = \frac{(2k)!}{k!k!} \approx \frac{\sqrt{4\pi k}\left(\frac{2k}{e}\right)^{2k}}{2\pi k\left(\frac{k}{e}\right)^{2k}} = \frac{2^{2k}}{\sqrt{\pi k}},$$

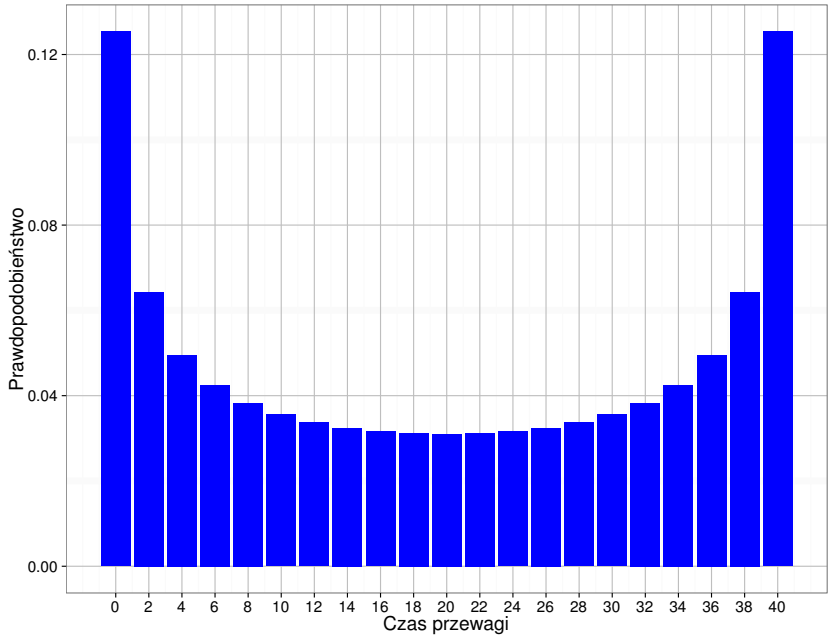
i podobnie

$$\binom{2n-2k}{n-k} \approx \frac{2^{2n-2k}}{\sqrt{\pi(n-k)}}.$$

Podstawiając to do wzoru (1.5) otrzymujemy

$$p_{2k,2n} \approx \frac{1}{\pi\sqrt{k(n-k)}} \tag{1.7}$$

Odpowiemy teraz na następujące pytanie: **jaka jest szansa, że w bardzo długiej grze byliśmy na prowadzeniu przez co najwyżej frakcję x czasu?** $(0 < x < 1)$



Rysunek 1.4: Rozkład czasu prowadzenia ustalonego gracza przy 40 rzutach monetą.

Niech $P_{2n}(x)$ oznacza szukane prawdopodobieństwo przy $2n$ rzutach monetą. Załóżmy na początek, że $x > \frac{1}{2}$. Wtedy

$$P_{2n}(x) = \sum_{k:\ \frac{k}{n}<x} p_{2k,2n} = \underbrace{\sum_{k:\ \frac{k}{n}\leqslant\frac{1}{2}} p_{2k,2n}}_{(\spadesuit)} + \underbrace{\sum_{k:\ \frac{1}{2}<\frac{k}{n}<x} p_{2k,2n}}_{(\clubsuit)}$$

Pamiętając o symetryczności rozkładu można zauważyć, że $(\spadesuit) \longrightarrow \frac{1}{2}$ (można to też uzasadnić inaczej – jeden z graczy musi być na prowadzeniu przez co najwyżej połowę czasu). Przy $n \to \infty$ i $\frac{1}{2} < \frac{k}{n} < x < 1$ zachodzi również $k \longrightarrow \infty$ oraz $n - k \longrightarrow \infty$. Dlatego drugą sumę możemy estymować korzystając z (1.7) oraz definicji całki Riemanna

$$\begin{aligned}(\clubsuit) &\approx \sum_{k:\ \frac{1}{2}<\frac{k}{n}<x} \frac{1}{\pi\sqrt{k(n-k)}} = \sum_{k:\ \frac{1}{2}<\frac{k}{n}<x} \frac{1}{\pi n} \frac{1}{\sqrt{\frac{k}{n}(1-\frac{k}{n})}} \\ &\xrightarrow[n\to\infty]{} \frac{1}{\pi}\int_{1/2}^{x} \frac{dt}{\sqrt{t(1-t)}} = \frac{2}{\pi}\arcsin(\sqrt{x}) - \frac{1}{2},\end{aligned}$$

czyli

$$P_{2n}(x) \xrightarrow[n\to\infty]{} \frac{2}{\pi}\arcsin(\sqrt{x}).$$

W celu znalezienia $P_{2n}(x)$ dla $0 < x < \frac{1}{2}$ skorzystamy ze znanych własności funkcji cyklome-

trycznych: $\arcsin x + \arccos x = \frac{\pi}{2}$ oraz $\arccos x = \arcsin(\sqrt{1-x^2})$.

$$P_{2n}(x) = 1 - P_{2n}(1-x) \xrightarrow[n\to\infty]{} 1 - \frac{2}{\pi}\arcsin(\sqrt{1-x}) = 1 - \frac{2}{\pi}\arccos(\sqrt{x})$$
$$= 1 - \frac{2}{\pi}\left(\frac{\pi}{2} - \arcsin(\sqrt{x})\right) = \frac{2}{\pi}\arcsin(\sqrt{x}).$$

W ten sposób udowodniliśmy

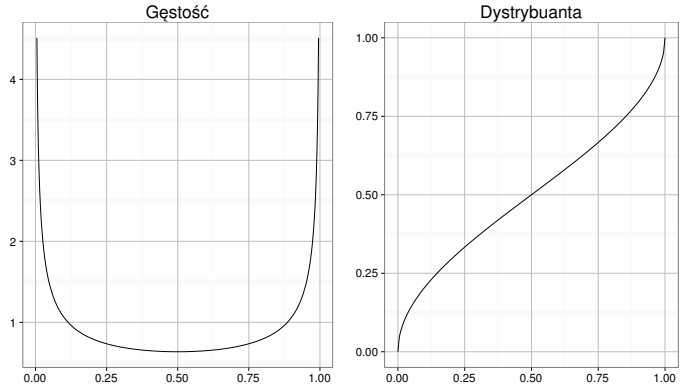
Twierdzenie 1.5 (Prawo arcusa sinusa). *Prawdopodobieństwo, że w n krokach frakcja czasu x ($0 \leqslant x \leqslant 1$), w której ustalony gracz ma przewagę (stan błądzenia przypadkowego jest dodatni), dąży przy $n \longrightarrow \infty$ do*

$$\frac{1}{\pi}\int_0^x \frac{dt}{\sqrt{t(1-t)}} = \frac{2}{\pi}\arcsin(\sqrt{x})$$

Innymi słowy w bardzo długiej grze frakcja czasu x spędzona "na plusie" ma rozkład arcusa sinusa. Oto jego podstawowe własności:

- gęstość: $f(t) = \frac{1}{\pi\sqrt{t(1-t)}}$,
- dystrybuanta: $F(t) = \frac{2}{\pi}\arcsin(\sqrt{t})$,
- wartość oczekiwana: $\frac{1}{2}$,
- wariancja: $\frac{1}{8}$.

Wykres gęstości i dystrybuanty przedstawia Rysunek 1.5. Funkcja gęstości w kształcie litery U pokazuje, że nierówny podział czasu przewagi jest zdecydowanie bardziej prawdopodobny niż względnie równomierny.



Rysunek 1.5: Rozkład arcusa sinusa.

Ludzka intuicja silnie podpowiada, że w grze z symetryczną monetą, każdy z graczy powinien być na plusie przez około połowę czasu. Wydaje się to logiczne – wiadomo, że liczba powrotów błądzenia przypadkowego do zera jest nieskończona w nieskończenie długiej grze. Zatem obaj gracze mają mniej więcej tyle samo fal kiedy są na plusie. Ponadto, jak wynikałoby z MPWL, średnia długość dodatniej fali powinna być dla obu graczy zbliżona. Co z kolei prowadzi do wniosku, że obaj powinni być na prowadzeniu przez podobną frakcję czasu. Gdzie tkwi błąd w tym rozumowaniu? Otóż nie możemy tu zastosować MPWL. Dotyczy ono zmiennych o skończonej wartości oczekiwanej. Tymczasem oczekiwany czas powrotu do zera w błądzeniu przypadkowym okazuje się być nieskończony, co kompletnie zmyla nasze intuicje.

CZĘŚĆ II

Generatory liczb pseudolosowych

Każdy intuicyjnie rozumie czym jest generator liczb pseudolosowych. Jednak dla pełności matematycznego opisu zaczniemy od przedstawienia jego ścisłej definicji. Przedstawiona tu teoria opiera się na książce [1]. Na jej podstawie opisujemy również kilka rodzajów generatorów liczb pseudolosowych. Sposób otrzymania z GLP ciągu bitów o dobrych własnościach również wymaga pewnego komentarza, o czym piszemy dalej. Na końcu tej części krótko omówimy znane metody testowania generatorów liczb pseudolosowych.

2.1 Definicja GLP

Istnieją metody otrzymania liczb „prawdziwie losowych". Najprostszym sposobem jest wielokrotne rzucenie kostką do gry lub monetą i stablicowanie otrzymanych wyników. Lepszym sposobem jest obserwowanie cząsteczek emitowanych przez próbkę radioaktywnego pierwiastka – uważa się, że rozkład radioaktywny jest dobrze modelowany przez proces Poissona. Kolejnym pomysłem jest wykorzystanie szumu atmosferycznego, z tej metody korzysta strona www.random.org, którą wykorzystamy w testach GLP. Źródła takiej losowości są jednak zazwyczaj zbyt wolne, trudno dostępne lub mają pewne inherentne wady (moneta może być niesymetryczna, detektor cząstek nie rejestruje zgłoszeń o zbyt krótkim odstępie, itp.). Wynika stąd potrzeba utworzenia deterministycznych algorytmów które imitowałyby losowość. Takie algorytmy nazywamy generatorami liczb pseudolosowych. Aby były praktyczne, muszą być szybkie i obliczalne na zwykłych komputerach. Liczby przez nie generowane jedynie „udają" losowe, dlatego nazywamy je pseudolosowymi.

Definicja 2.1. Generator liczb pseudolosowych jest to piątka[1] $\langle E, V, s_0, f, g\rangle$, gdzie E jest skończoną przestrzenią stanów, V jest zbiorem wartości zwracanych przez generator, s_0 jest to tzw. ziarno, czyli początkowy stan w ciągu stanów $(s_i)_{i=0}^{\infty}$, funkcja $f : E \to E$ opisuje przejścia między kolejnymi stanami: $s_n = f(s_{n-1})$, zaś $g : E \to V$, odwzorowuje stan generatora w wartość przez niego zwracaną.

[1] Jest to nieco inna definicja niż w [1], dostosowana do naszych potrzeb. W książce Asmussena przyjmuje się $V = [0, 1]$, zaś zamiast s_0 w definicji znajduje się μ – rozkład prawdopodobieństwa początkowego stanu.

Najczęściej przyjmuje się $V = (0, 1)$ lub $V = \bar{M}$ dla pewnego M (dla $n \in \mathbb{N}$ symbol $\bar{n}$ oznacza zbiór $\{0, 1, \ldots, n-1\}$). U nas będzie zachodzić właśnie ta druga możliwość.

Zauważmy, że każdy GLP prędzej lub później „zapętla się", tzn. musi istnieć takie d, że dla pewnego l zachodzi $s_{l+d} = s_l$ (wynika to ze skończoności przestrzeni stanów). Minimalne d o tej własności nazywane jest *okresem* generatora. Dobre generatory powinny mieć jak najdłuższe okresy, optymalnie równe $|E|$.

Poniżej przedstawiamy popularne rodzaje GLP.

LCG

Generatory LCG (od ang. *linear congruential generator*) zmieniają swój stan zgodnie z rekurencją

$$s_n = (as_{n-1} + c) \mod M. \tag{2.1}$$

Generator tej klasy jest określony przez moduł M, mnożnik a oraz przyrost c, co oznaczamy $LCG(M, a, c)$. Zauważmy, że $LCG(M, a, c)$ spełnia definicję GLP z $E = \bar{M}$, $V = \bar{M}$, $f(x) = (ax + c) \mod M$ oraz $g(x) = x$.

Dobranie wartości M, a, c o dobrych własnościach przysparza sporych problemów. Jak się jednak okazuje, istnieje kryterium ułatwiające zapewnienie generatorowi długiego okresu.

Twierdzenie 2.1. *Przy poniższych warunkach $LCG(M, a, c)$ ma okres równy M:*

- *c oraz M są względnie pierwsze,*
- *jeśli p jest liczbą pierwszą i $p|M$, to $p|(a-1)$,*
- *jeśli $4|M$, to $4|(a-1)$.*

Powyższe twierdzenie pochodzi z pracy [3].

Zauważmy jednak, że znalezienie LCG o pełnym okresie nie gwarantuje, że generator będzie dobrej jakości. Łatwo zauważyć następujący

Fakt 2.2. *Niech $M = 2^k$. Wówczas d najmniej istotnych bitów $LCG(M, a, c)$ ma okres równy co najwyżej 2^d.*

W tym przypadku nie ma więc mowy o niezależności liczb generowanych przez LCG. Niektóre pakiety korzystające z LCG częściowo obchodzą ten problem zwracając tylko najbardziej znaczące bity wygenerowanych liczb.

MCG

MCG (od ang. *multiplicative congruential generator*) znany jest też jako GLP Lehmera lub GLP Parka-Millera. Jest to szczególny przypadek LCG, w którym $c = 0$, czyli kolejne stany opisuje rekurencja

$$s_n = as_{n-1} \mod M. \tag{2.2}$$

MCG o parametrach M oraz a oznaczamy $MCG(M, a)$. Aby $MCG(M, a)$ mogło mieć dobre własności, M powinno być liczbą pierwszą lub jej potęgą, a powinno być generatorem grupy $\mathbb{Z}_M^*$, a ziarno s_0 powinno być względnie pierwsze z M.

GLCG

Wyżej opisane GLP dzielą pewną wadę – mają stosunkowo krótkie okresy. W przypadku gdy potrzebujemy dłuższych okresów przydatne mogą być uogólnione LCG (od ang. *generalized linear congruential generator*). Postępują one zgodnie z rekurencją

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + \ldots + a_k x_{n-k}) \mod M. \tag{2.3}$$

GLCG zmieniający stany w ten sposób oznaczamy $GLCG(M, (a_i)_{i=1}^k)$. Jest to ciągle GLP w myśl definicji 2.1, gdzie $E = \bar{M}^k$, $s_n = \langle x_n, x_{n-1}, \ldots x_{n-k+1} \rangle$, $g(s_n) = x_n$. Dobry dobór parametrów może dać okres równy $M^k - 1$.

Generatory mieszane

Dobrym pomysłem na ulepszenie GLP jest połączenie kilku generatorów w jeden. Załóżmy, że mamy dane k GLP $\langle E_j, V_j, s_{j,0}, f_j, g_j \rangle$, $1 \leqslant j \leqslant k$, gdzie j-ty generator zmienia stan według zależności

$$s_{j,n} = f_j(s_{j,n-1}).$$

Możemy teraz zdefiniować mieszany generator w taki sposób, aby ciąg jego stanów spełniał

$$s_n = \langle s_{1,n}, s_{2,n}, \ldots s_{k,n} \rangle$$

Niech ponadto d_j oznacza okres j-tego „składowego" generatora. Jak pokazał L'Ecuyer ([7], Lemma 2) mieszany generator ma okres $d = NWW(d_1, d_2, \ldots, d_k)$.

Szczególnym przypadkiem generatorów mieszanych są **CMCG** (od ang. *combined multiplicative congruential generator*). Składa się on k generatorów $MCG(M_j, a_j)$, gdzie M_j są liczbami pierwszymi, czyli funkcją przejścia jest

$$s_{j,n} = a_j s_{j,n-1} \mod M_j.$$

Wyjście generatora mieszanego otrzymujemy ze wzoru

$$g(s_n) = \left(\sum_{j=1}^{k} (-1)^{j-1} s_{j,n} \right) \mod M_1 - 1$$

Ponadto jeśli liczby $\frac{M_j - 1}{2}$ są względnie pierwsze, to CMCG ma optymalny okres wynoszący $\frac{1}{2^k}(M_1 - 1) \cdot \ldots \cdot (M_k - 1)$.

LFSR

LFSR (od ang. *Linear feedback shift register*) to, z grubsza rzecz ujmując, generator produkujący liczby pseudolosowe na podstawie obwodu bramek logicznych. Stanem takiego generatora jest sekwencja bitów, które przekazywane są na wejście wybranych bramek. Wyjścia tych bramek stanowią wejście innych bramek, te z kolei przekazują swoje wyjścia kolejnym bramkom, itd. Wyjścia ustalonych bramek mogą zmieniać stan generatora lub być zwracane jako rezultat pracy generatora.

Mersenne Twister

Mersenne Twister (MT19937) został zaproponowany przez Matsumoto i Nashimurę w [9]. Nie jest to wprawdzie klasa generatorów, ale przykład konkretnego GLP, jednak ze względu na jego ogromną popularność warto go opisać. Jest on standardowym generatorem w wielu narzędziach programistycznych, między innymi w R, Python, MATLAB, Julia. Został dołączony również do standardowej biblioteki C++11.

Do opisu generatora wykorzystamy notację, w której zapis $d[i..j]$ oznacza bity o indeksach od i do j w liczbie d, symbol $\oplus$ to operacja XOR na kolejnych bitach, symbol & to operacja AND na kolejnych bitach, zaś symbole $\ll$ oraz $\gg$ to operacja bitowego przesunięcia odpowiednio w lewo i w prawo.

Stan generatora opisany jest przez 624 32-bitowe liczby

$$x_k, x_{k+1}, \ldots x_{k+623}.$$

Kolejne stany otrzymujemy ze wzoru

$$x_{k+624} = \begin{cases} x_{397+k} \oplus (0, x_k[0], x_{k+1}[1..30]) & \text{jeśli } x_{k+1}[31] = 0 \\ x_{397+k} \oplus (0, x_k[0], x_{k+1}[1..30]) \oplus a & \text{jeśli } x_{k+1}[31] = 1, \end{cases}$$

gdzie $a = (9908B0D)_{16}$. Przy k-tym wywołaniu MT19937 zwraca jako wyjście wartość $t(x_{623+k})$, przy czym

$$t(x) = y_3 \oplus (y_3 \gg 18),$$

gdzie

$$\begin{aligned} y_3 &= y_2 \oplus ((y_2 \ll 15) \ \& \ (EFC60000)_{16}) \\ y_2 &= y_1 \oplus ((y_1 \ll 7) \ \& \ (9D2C5680)_{16}) \\ y_1 &= x \oplus (x \gg 11) \end{aligned}$$

Generator uzyskany w ten sposób ma okres równy $2^{19937} - 1$, co wyjaśnia jego skrótową nazwę.

2.2 Generowanie sekwencji bitów

Komputery operują tylko i wyłącznie na ciągach bitów, dlatego wyjście każdego programu, w szczególności GLP, może być traktowane jako ciąg zerojedynkowy. My potrzebujemy jednak ciągów specyficznych: każdy bit musi być generowany niezależnie i z jednakowym prawdopodobieństwem przyjmować wartości zero i jeden. Dlatego dla wygody języka wprowadźmy poniższy termin.

Definicja 2.2. *Idealnym ciągiem losowych bitów* nazywamy proces Bernoulliego z prawdopodobieństwem sukcesu $p = \frac{1}{2}$

Algorytm 2.1 Generowanie sekwencji bitów przy użyciu GLP.

```
1: procedura GenerujCiągBitów(glp)
2:     s ← ϵ                                   ▷ ϵ to słowo puste
3:     dopóki s nie jest wystarczająco długi wykonuj
4:         a ← następna liczba z glp
5:         b ← binarny zapis a na ⌈log_2 M⌉ bitach
6:         s ← s ⊙ b                           ▷ ⊙ to operator konkatenacji
7:     zwróć s.
```

Poniżej opisujemy jak zmienić generator liczb pseudolosowych w „generator pseudolosowych ciągów zerojedynkowych".

Mamy dany GLP generujący liczby całkowite ze zbioru $\bar{M} = \{0, 1, \ldots, M-1\}$. Kolejne wywołania powinny dawać niezależne wyniki. Aby wykorzystać GLP do wygenerowania idealnego ciągu losowych bitów możemy użyć Algorytmu 2.1. Jeśli M jest potęgą dwójki, to zadziała on dobrze, mamy bowiem

Lemat 2.3. *Niech $M = 2^k$. Jeżeli GLP w każdym kroku generuje liczby niezależnie i jednostajnie w zbiorze $\bar{M}$, to Algorytm 2.1 generuje idealny ciąg losowych bitów.*

Dowód. W przypadku $M = 2^k$ mamy wzajemnie jednoznaczną odpowiedniość pomiędzy zbiorem $\bar{M}$ oraz układami k bitów. Oznacza to, że w jednym kroku otrzymujemy z GLP każdy możliwy układ k bitów z jednakowym prawdopodobieństwem $\frac{1}{2^k}$. Łatwo zauważyć, że wtedy każdy generowany bit ma równe szanse bycia jedynką i zerem, oraz jest niezależny od pozostałych. □

Jednak jeśli M nie jest potęgą dwójki, to procedura nie działa – przykładowo dla $M = 5$ w każdym kroku doklejamy jedną z sekwencji $\{000, 001, 010, 011, 100\}$. Wówczas w wygenerowanym ciągu spotkanie jedynki jest mniej prawdopodobne niż zera – jedynki stanowią tylko około $\frac{1}{3}$ wszystkich wygenerowanych bitów. Ponadto nie ma niezależności – wystąpienie jedynki na bicie o indeksie podzielnym przez 3 oznacza, że kolejne dwa bity będą zerami.

Jak widać, gdy M nie jest postaci 2^k nie możemy w wyjściowym ciągu tak po prostu umieścić binarnego zapisu wygenerowanej liczby, gdyż na najbardziej znaczących bitach zera mogą znacząco przeważać. Prostym obejściem tego problemu jest ograniczenie się do mniej znaczących bitów generowanych liczb. Wprawdzie one również nie mają idealnego rozkładu, co można łatwo zauważyć licząc prawdopodobieństwo wystąpienia jedynki na najmniej znaczącym bicie, jednak odstępstwa są stosunkowo niewielkie.

Inne podejście przedstawia Algorytm 2.2. Jego dodatkową zaletą jest możliwość modyfikacji, tak by działał dla GLP zwracających liczby z odcinka $(0, 1)$.

Algorytm 2.2 Generowanie sekwencji bitów przy użyciu GLP.

1: **procedura** GenerujCiągBitów(`glp`, d)
2: $s \leftarrow \epsilon$
3: **dopóki** s nie jest wystarczająco długi **wykonuj**
4: $a \leftarrow$ następna liczba z `glp`
5: $b \leftarrow d$ pierwszych bitów rozwinięcia dwójkowego $\frac{a}{M}$
6: $s \leftarrow s \odot b$
7: **zwróć** s.

2.3 Popularne metody testowania GLP

Poniżej przedstawiamy kilka wybranych metod testowania GLP. Lista z pewnością jest daleka od kompletności, zwłaszcza, że zagadnienie testowania GLP cieszy się dużą popularnością.

Do testów używany jest ciąg liczb wygenerowanych przez GLP. Poniżej zazwyczaj będzie nam wygodnie przyjmować, że jest to ciąg liczb

$$U_1, U_2, U_3, \ldots$$

pretendujący do miana ciągu niezależnie i równomiernie rozłożonego na odcinku $(0, 1)$.

Zgodność z rozkładem jednostajnym

Pierwszym nasuwającym się sposobem sprawdzenia jakości liczb generowanych przez GLP jest zastosowanie znanego aparatu statystycznego. Możemy użyć testów zgodności z rozkładem jednostajnym, np. **testu Kołmogorowa-Smirnowa**. Niech n będzie długością ciągu (U_i). Dystrybuanta empiryczna jest zdefiniowana jako

$$F_n(t) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}(U_i < t).$$

Niech

$$D_n = \sup_{0 \leqslant t \leqslant 1} |F_n(t) - F(t)|.$$

Zauważmy, że obliczenie D_n nie stanowi problemu, gdyż F_n jest funkcją schodkową zmieniającą wartość w punktach $U_1, U_2, \ldots$. Twierdzenie Kołmogorowa mówi, że $\sqrt{n} D_n \xrightarrow{D} K$, gdzie K jest zmienną losową o rozkładzie Kołmogorowa. Korzystając z tablic lub pakietów statystycznych możemy znaleźć p-wartość tego testu.

Można też użyć **testu χ^2 Pearsona**. Polega on na podzieleniu odcinka na r części. Niech E_i będzie oczekiwaną liczbą zmiennych U_i, których wartość wpada to i-tego odcinka, zaś O_i obserwowaną liczbą.

Wówczas statystyka

$$T = \sum_{i=1}^{r} \frac{(O_i - E_i)^2}{E_i} \tag{2.4}$$

dąży do rozkładu χ^2 z $(r-1)$ stopniami swobody, oznaczanego $\chi^2(r-1)$. Tak jak w przypadku testu Kołmogorowa-Smirnowa możemy poznać p-wartość testu korzystając z odpowiednich narzędzi.

Wiele testów zgodności opracowano dla rozkładu normalnego, do najbardziej znanych należą **test Shapiro-Wilka**, **test Jarque-Bera**, **test Andersona-Darlinga**. Aby móc z nich skorzystać, wystarczy zmapować ciąg (U_i) na ciąg (N_i) zmiennych losowych o rozkładzie normalnym, np. przy użyciu transformacji Boxa-Mullera.

Powyższe testy sprawdzają jedynie zgodność z rozkładem jednostajnym. Do sprawdzenia niezależności teoretycznie można ponownie wykorzystać test χ^2. Dla ustalonego t dzielimy ciąg (U_i) na bloki

$$(U_1, \ldots, U_t), (U_{t+1}, \ldots, U_{2t}), \ldots \tag{2.5}$$

Otrzymujemy w ten sposób obserwacje, które powinny być jednostajnie rozłożone w hiperkostce $(0,1)^t$. Możemy ją podzielić na mniejsze kostki i skorzystać ze statystyki (2.4), aby stwierdzić czy wpada do nich odpowiednio wiele obserwacji. W praktyce jednak, oczekiwana liczba obserwacji w pojedynczej kostce spada do zera tak szybko, że T nie jest dobrze przybliżane przez rozkład χ^2.

Zgodność z twierdzeniami rachunku prawdopodobieństwa

Wiele faktów w rachunku prawdopodobieństwa opiera się na ciągach niezależnych zmiennych losowych o jednakowym rozkładzie. Dzięki temu na podstawie ciągu (U_i) jesteśmy w stanie otrzymać kolejne zmienne losowe, których teoretyczne rozkłady są znane. W [6] zaproponowanych jest kilka praw, które można wykorzystać w ten sposób. Oto niektóre z nich:

- **Test odstępów.** Dla ustalonego przedziału (α, β) mierzymy czasy oczekiwania na kolejne U_i wpadające do tego przedziału. Otrzymane wartości powinny mieć rozkład geometryczny, co sprawdzamy testem χ^2.

- **Test permutacyjny.** Podzielmy ciąg (U_i) na bloki jak w (2.5), przy niezbyt dużym t. W każdym bloku zachodzi jedno z $t!$ możliwych uporządkowań. Rozkład na uporządkowaniach powinien być jednostajny, co ponownie weryfikujemy testem χ^2.

- **Test kolizji.** Stanowi rozwiązanie, gdy mamy n obserwacji wpadających do m „pudełek", przy czym $n < m$. Jak powiedzieliśmy wcześniej, w takiej sytuacji nie możemy zastosować testu χ^2. Jednak da się wyliczyć teoretyczne prawdopodobieństwo otrzymania k kolizji (kolizją jest trafienie obserwacji do pudełka, w którym jest już inna obserwacja). Jeśli zaobserwowana liczba kolizji nie mieści się w pewnych ramach, to możemy stwierdzić, że ciąg nie jest losowy.

Metody opisane tutaj mają pewną zaletę w stosunku do przedstawionych wcześniej testów zgodności (U_i) z rozkładem jednostajnym – niejawnie testują również niezależność.

Zestawy testów

Rozwinięciem podejścia z poprzedniego paragrafu jest tworzenie paczek testowych. Zawierają one kilkanaście lub więcej testów opartych o fakty rachunku prawdopodobieństwa, które powinien spełniać idealny ciąg losowych bitów. Znane przykłady to:

- **Diehard tests.** Zestaw opracowany przez George'a Marsaglia w 1995 r. Obecnie uważany już za przestarzały.

- **TestU01.** Następca zestawu Diehard, który opracowali Pierre L'Ecuyer oraz Richard Simard w 2007 r.

- **NIST Test Suite.** Zestaw opracowany przez organizację *National Institute of Standards and Technology* i ciągle rozwijany.

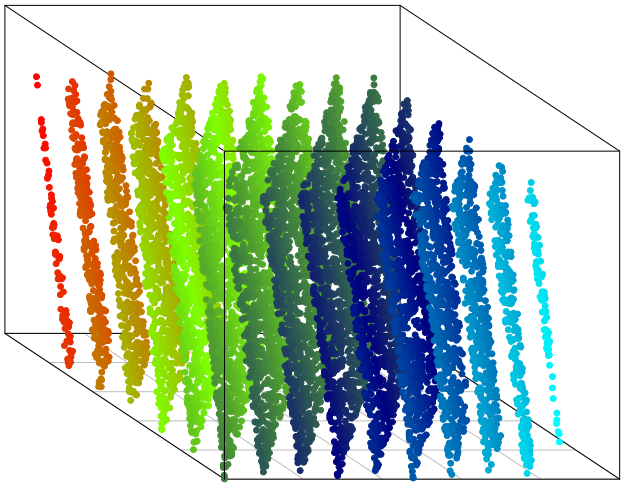
Powiemy więcej o tym ostatnim zestawie. Zawiera on kilkanaście testów rozstrzygających losowość ciągów zerojedynkowych. Hipotezą zerową jest stwierdzenie, że testowana sekwencja jest realizacją idealnego ciągu losowych bitów. Testy badają m.in.: stosunek liczby jedynek do długości ciągu, liczbę jednocyfrowych podciągów, długość najdłuższego podciągu zawierającego same jedynki.

Jakość GLP oceniana jest w systematyczny sposób. Dla każdego testu w NIST Test Suite postępujemy następująco. Generujemy m sekwencji bitów. Po kolei dla każdej z nich przeprowadzamy wybrany test na ustalonym poziomie istotności $\alpha = 0.01$. Test zwraca nam p-wartość, i jeżeli $p > \alpha$, to uznajemy, że dany ciąg bitów jest losowy. Przyjmuje się, że GLP zaliczył wykonywany test jeżeli około 97% lub więcej sekwencji zostało uznanych za losowe (oczywiście nie można wymagać, żeby wszystkie ciągi zostały uznane za losowe, bo nawet spośród idealnych ciągów losowych bitów około α z nich zostanie odrzucona).

Takie podejście wydaje się rozsądne, ma jednak zasadniczą wadę. Wyobraźmy sobie, że mamy znakomity GLP g_1. Na jego podstawie tworzymy nowy GLP g_2 w taki sposób, że co setny ciąg bitów otrzymanych z g_2 , a w pozostałych przypadkach g_2 deleguje wygenerowanie ciągu do g_1. NIST Test Suite prawdopodobnie uznałby generator g_2 za dobry. Metoda opisana w części 3 ma potencjał do wykrywania tego rodzaju nieprawidłowości.

Test spektralny

Testowi spektralnemu Donald Knuth poświęcił kilkanaście stron w swoim dziele [6], czego nie sposób tutaj streścić. Idea polega na spostrzeżeniu, że punkty w t-wymiarowej przestrzeni, utworzone z kolejnych wyrazów ciągu (U_i) wygenerowanego przez LCG, leżą na stosunkowo niewielkiej liczbie $(t-1)$-wymiarowych hiperpłaszczyzn. Zagłębiając się w ten temat można dojść do dość skomplikowanej metody testowania LCG. Jednak w niektórych przypadkach widać gołym okiem, że generator jest zły. Takim przykładem jest niechlubny RANDU (jest to $MCG(2^{31}, 2^{16}+3)$). Na Rysunku 2.1 przedstawiono punkty w przestrzeni otrzymane z tego generatora. Widać wyraźnie, że układają się one na 15 płaszczyznach.



Rysunek 2.1: Ilustracja rozmieszczenia w przestrzeni punktów generowanych przez RANDU. Każdy punkt został utworzony z trzech kolejnych liczb otrzymanych z generatora i przeskalowanych na odcinek $(0, 1)$

Złożoność Kołmogorowa

Zacznijmy od przykładu. Rozważmy ciągi binarne długości 32

$$01010101010101010101010101010101$$

oraz

$$00110111001101001101101000110110.$$

Choć wylosowanie obydwu z nich jest równie prawdopodobne, ten drugi uznajemy za bardziej „losowy". Dzieje się tak dlatego, że pierwszy ciąg można jednoznacznie opisać w znacznie mniejszej liczbie znaków:

$$16\times 01,$$

zaś najkrótszym opisem drugiego ciągu jest prawdopodobnie przepisanie go całego.

Tego typu intuicje próbujemy wyjaśniać za pomocą, tzw. złożoności Kołmogorowa. Służy ona do mierzenia stopnia skomplikowania ciągów znaków. Ustalmy dowolny język programowania L. Złożonością Kołmogorowa łańcucha znaków s jest długość najkrótszego programu P w języku L, takiego że P wypisuje s.

Rzadko kiedy jesteśmy w stanie znaleźć dokładną wartość złożoności łańcucha s. Dlatego może się wydawać, że powyższe podejście jest czysto teoretyczne. Okazuje się jednak, że można złożoność s oszacować. Niech P będzie programem implementującym ustalony algorytm kompresji, a $\tilde{P}$ odpowiadającym mu programem dekompresji. Ponadto niech $\tilde{s}$ będzie skompresowanym łańcuchem s. Wówczas złożoność łańcucha s jest oszacowana z góry przez sumę długości $\tilde{P}$ oraz $\tilde{s}$. Odrzucamy hipotezę o losowości s, gdy otrzymana wartość jest znacznie mniejsza od długości s.

Zagadnienie ruiny gracza

Bardzo interesująca metoda testowania GLP została przedstawiona w pracy [5]. Rozważane jest błądzenie po grupie $\mathbb{Z}_n$. Autorzy badają czas dojścia do stanu 0, co odpowiada zbiciu majątku wysokości n lub bankructwu gracza ze znanego zagadnienia. W pracy opisane są trzy warianty zastosowanej metody, tutaj przedstawiamy podstawową z nich.

Ustalmy $p \in (0,1)$. Ciąg (U_i) otrzymany z generatora wykorzystywany jest do poruszania się po grupie $\mathbb{Z}_n$. Jeśli w kroku i-tym byliśmy w stanie s, to przechodzimy do stanu $s+1$, gdy $U_{i+1} < p$, a do $s-1$ w przeciwnym przypadku. Dla każdego $x \in \mathbb{Z}_n$, $x \neq 0$ niech T_x oznacza czas dojścia do 0 ze stanu x. N-krotnie z punktu x rozpoczynamy błądzenie po grupie. W ten sposób otrzymujemy N replikacji zmiennej T_x. Oznaczmy ich średnią przez $\overline{T}_x$. Niech μ oraz σ^2 oznaczają wartość oczekiwaną oraz wariancję T_x, gdy prawdopodobieństwa przejść do stanów $s+1$ oraz $s-1$ wynoszą odpowiednio p oraz $1-p$. Wartości μ i σ^2 można wyznaczyć teoretycznie. Niech

$$Z_x = \frac{\overline{T}_x - \mu}{\sigma\sqrt{N}}.$$

Przy założeniu hipotezy, o niezależności i rozkładzie jednostajnym zmiennych (U_i), statystyka Z_x ma w przybliżeniu rozkład normalny $\mathcal{N}(0,1)$. W związku z tym przy dużych wartościach $|Z_x|$ należy stwierdzić, że generator nie przeszedł testu w punkcie x.

Błądzenie przypadkowe

Testom opartym na własnościach błądzenia przypadkowego poświęcone są kolejne dwie części pracy.

CZĘŚĆ III

Metoda testowania oparta na błądzeniu przypadkowym

W części 1 przedstawiliśmy teorię przydatną do testowania GLP. W tej części pokazujemy jak zastosować ją w praktyce. Opisujemy w jaki sposób sprawdzać czy wygenerowane ciągi odpowiadają prawdziwie losowym realizacjom błądzenia przypadkowego. Przedstawiona tu metoda wykorzystująca prawo iterowanego logarytmu pochodzi z pracy [10]. Dodatkowo proponujemy podobną metodę opartą o prawo arcusa sinusa. W wykonywanych obliczeniach często polegamy na aproksymacjach, dlatego w ostatnim paragrafie tej części analizujemy wielkość popełnianego błędu.

3.1 Opis metody

Ogólna idea testów, które omawiamy w tej części pracy, nie jest skomplikowana. Metoda polega na obliczeniu pewnych charakterystyk ciągów wygenerowanych przez GLP i porównaniu ich empirycznych rozkładów z rozkładami, które są znane dla idealnego ciągu losowych bitów.

Przykładowo, w części poświęconej prawu arcusa sinusa uzasadniliśmy, że bardziej prawdopodobna jest długa przewaga liczby jedynek nad liczbą zer niż równomierny rozkład prowadzenia. Jeśli generator sztucznie wyrównuje częstość zer i jedynek, to zauważymy odstępstwa od tej reguły. Zgodność z prawem arucsa sinusa sprawdzają testy oparte o zdefiniowaną poniżej charakterystykę S_n^{asin}.

Podobnie ktoś mógłby pomyśleć, że czymś pozytywnym byłyby niewielkie różnice między liczbą jedynek i zer w ciągu bitów otrzymanym z GLP. Moglibyśmy zdecydować się na jakąś „rozsądną" stałą, powiedzmy 100, i uznać, że generator jest dobry jeśli różnica liczby zer i jedynek w ciągu nie przekroczy 100. Wszak duże różnice mogłyby sugerować, że mamy różne prawdopodobieństwa wystąpienia zer i jedynek. Jednak prawo iterowanego logarytmu pokazuje, że to rozumowanie jest błędne. *Należy* spodziewać się fluktuacji i odstępstw od zera, a ich brak oznacza, że GLP nie generuje idealnego ciągu losowych bitów. Tę obserwację wykorzystują testy oparte o charakterystykę S_n^{lil}.

3.1.1 Test arcusa sinusa

Niech

$$D_k = \mathbb{1}\left(S_k > 0 \vee S_{k-1} > 0\right), k = 1, 2, \ldots, n. \tag{3.1}$$

Zmienna D_k przyjmuje wartość 1, gdy w k-tym kroku błądzenia przypadkowego zachodzi przewaga liczby jedynek nad zerami (traktując remisy tak jak w paragrafie 1.2), zaś 0 w przeciwnym przypadku. Zdefiniujmy charakterystykę S_n^{asin} wzorem

$$S_n^{asin} = \frac{1}{n}\sum_{k=1}^{n} D_k. \tag{3.2}$$

Zatem S_n^{asin} jest to frakcja czasu podczas której jedynki dominowały nad zerami. Wiemy z paragrafu 1.2, że

$$\mathbb{P}\left(S_n^{asin} = \frac{k}{n}\right) = p_{k,n},$$

zaś korzystając z prawa arcusa sinusa

$$\begin{aligned}\mathbb{P}\left(S_n^{asin} \in (a,b)\right) &\approx \int_a^b \frac{dt}{\sqrt{t(1-t)}} \\ &= \frac{2}{\pi}\arcsin(\sqrt{b}) - \frac{2}{\pi}\arcsin(\sqrt{a}).\end{aligned} \tag{3.3}$$

W celu przetestowania GLP generujemy przy jego użyciu m ciągów zerojedynkowych długości n. Otrzymujemy w ten sposób m realizacji zmiennej S_n^{asin}, j-tą replikację oznaczamy $S_{n,j}^{asin}$. Ustalamy partycję prostej rzeczywistej i dla każdego odcinka w tej partycji zliczamy ile realizacji S_n^{asin} do niego wpadło.

W testach opartych o wielkość S_n^{asin} korzystamy z $(s+2)$-elementowej partycji postaci $\mathcal{P}_s^{asin} = \{P_0^{asin}, P_1^{asin}, \ldots, P_{s+1}^{asin}\}$, gdzie

$$\begin{aligned}P_0^{asin} &= \left(-\infty, -\frac{1}{2s}\right), \\ P_i^{asin} &= \left[\frac{2i-3}{2s}, \frac{2i-1}{2s}\right), \quad 1 \leqslant i \leqslant s, \\ P_{s+1}^{asin} &= \left[1 - \frac{1}{2s}, \infty\right).\end{aligned}$$

Możemy teraz zdefiniować dwie miary określone na partycji $\mathcal{P}_s^{asin}$. Pierwsza z nich, μ_n^{asin}, reprezentuje teoretyczny rozkład rozważanej charakterystyki:

$$\mu_n^{asin}\left(P_i^{asin}\right) = \mathbb{P}\left(S_n^{asin} \in P_i^{asin}\right), \quad 0 \leqslant i \leqslant s+1. \tag{3.4}$$

Powyższą wartość możemy wyliczyć ze wzoru (3.3).

Druga miara, ν_n^{asin}, reprezentuje rozkład empiryczny, wyznaczony w testach. Określamy

$$\nu_n^{asin}\left(P_i^{asin}\right) = \frac{\left|\{j :\ S_{n,j}^{asin} \in P_i^{asin}, 1 \leqslant j \leqslant m\}\right|}{m}, \quad 0 \leqslant i \leqslant s+1. \tag{3.5}$$

Jeżeli testowany GLP jest dobry, to obie miary powinny być „mniej więcej takie same". Ściślej, odległość między otrzymanymi miarami powinna być mała. Korzystamy ze znanych funkcji odległości *total variation distance* oraz *separation distance*. Są one zdefiniowane następująco dla dowolnych miar μ oraz ν:

$$d^{tv}(\mu,\nu) = \sup_{A\subseteq\mathbb{R}} |\mu(A) - \nu(A)|, \tag{3.6}$$

$$d^{sep}(\mu,\nu) = \sup_{A\subseteq\mathbb{R}} \left(1 - \frac{\mu(A)}{\nu(A)}\right) \tag{3.7}$$

Ponieważ nie jesteśmy w stanie wyznaczyć supremum na całej prostej, skorzystamy z nieco uproszczonych definicji. Dla partycji $\mathcal{P}$ prostej rzeczywistej, definiujemy:

$$d^{tv}_{\mathcal{P}}(\mu,\nu) = \frac{1}{2}\sum_{A\in\mathcal{P}} |\mu(A) - \nu(A)| = \sum_{\substack{A\in\mathcal{P},\\ \mu(A)>\nu(A)}} \Big(\mu(A) - \nu(A)\Big) \tag{3.8}$$

$$d^{sep}_{\mathcal{P}}(\mu,\nu) = \max_{A\in\mathcal{P}} \left(1 - \frac{\mu(A)}{\nu(A)}\right) \tag{3.9}$$

Inne podejście polega na testowaniu hipotezy o zgodności rozkładów. Korzystając z terminologii statystycznej wielkości $S^{asin}_{n,j}$ będziemy nazywać obserwacjami. Hipotezą zerową jest stwierdzenie, że obserwacje mają rozkład μ^{asin}_n, czyli de facto, że GLP generuje idealny ciąg losowych bitów. Niech O_i oznacza liczbę obserwacji wpadających do przedziału P^{asin}_i, tzn.

$$O_i = |\{j:\ S^{lil}_{n,j} \in P^{asin}_i, 1 \leqslant j \leqslant m\}|, \quad 0 \leqslant i \leqslant s+1, \tag{3.10}$$

oraz E_i oznacza *oczekiwaną* liczbę obserwacji wpadających do tego przedziału, czyli

$$E_i = m \cdot \mathbb{P}\left(S^{asin}_n \in P^{asin}_i\right), \quad 0 \leqslant i \leqslant s+1. \tag{3.11}$$

Wartość E_i obliczamy ze wzoru (3.3). Przy hipotezie zerowej statystyka

$$T = \sum_{i=0}^{s+1} \frac{(O_i - E_i)^2}{E_i} \tag{3.12}$$

ma w przybliżeniu rozkład $\chi^2(s+1)$. Duże wartości tej statystyki są dowodem wadliwości GLP.

3.1.2 Test iterowanego logarytmu

Przyjrzyjmy się teraz metodzie zastosowanej w [10]. Jest ona podobna do metody opisanej w poprzednim paragrafie. Liczymy jedynie inną charakterystykę ciągów i dostosowujemy partycję prostej. Przypomnijmy oznaczenie

$$S^{lil}_n = \frac{S_n}{\sqrt{2n\log\log n}}. \tag{3.13}$$

Można łatwo znaleźć teoretyczny rozkład tej charakterystyki (tzn. rozkład dla idealnego ciągu losowych bitów). Istotnie, korzystając z centralnego twierdzenia granicznego dostajemy

$$\begin{aligned}\mathbb{P}\left(S_n^{lil} \in (a,b)\right) &= \mathbb{P}\left(\frac{S_n}{\sqrt{n}} \in \left(a\sqrt{2\log\log n}, b\sqrt{2\log\log n}\right)\right) \\ &\approx \Phi(b\sqrt{2\log\log n}) - \Phi(a\sqrt{2\log\log n})\end{aligned} \tag{3.14}$$

Jako, że S_n^{lil} przyjmuje swoje wartości w szerszym przedziale niż S_n^{asin}, dlatego korzystamy z innej partycji prostej, mianowicie $\mathcal{P}_s^{lil} = \{P_0^{lil}, P_1^{lil}, \ldots, P_{s+1}^{lil}\}$, gdzie

$$\begin{aligned}P_0^{lil} &= (-\infty, -1), \\ P_i^{lil} &= \left[-1 + \frac{2(i-1)}{s}, -1 + \frac{2i}{s}\right), \quad 1 \leqslant i \leqslant s, \\ P_{s+1}^{lil} &= [1, \infty).\end{aligned}$$

Teoretyczny i empiryczny rozkład określamy w tym przypadku następująco:

$$\mu_n^{lil}\left(P_i^{lil}\right) = \mathbb{P}\left(S_n^{lil} \in P_i^{lil}\right), \quad 0 \leqslant i \leqslant s+1, \tag{3.15}$$

$$\nu_n^{lil}\left(P_i^{lil}\right) = \frac{|\{j :\ S_{n,j}^{lil} \in P_i^{lil}, 1 \leqslant j \leqslant m\}|}{m}, \quad 0 \leqslant i \leqslant s+1, \tag{3.16}$$

przy czym wartość (3.15) znamy dzięki (3.14), zaś $S_{n,j}^{lil}$ oznacza oczywiście replikacje S_n^{lil}, obliczone dla kolejnych ciągów.

3.2 Analiza błędu

Omawiając test arcusa sinusa oraz test iterowanego logarytmu dokonywaliśmy w obliczeniach pewnych przybliżeń. Wystarcza to do przekazania idei opisanych metod testowania, jednak dla porządku należy dowieść, że wprowadzona niedokładność nie ma istotnego znaczenia. W [10] uzasadniono, że dla $n \geqslant 26$ błąd przybliżenia w (3.14) jest pomijalny. Tutaj oszacujemy błąd popełniany w (3.3).

W przypadku testu arcusa sinusa korzystaliśmy z aproksymacji dwukrotnie: najpierw przybliżając $p_{2k,2n}$ używając wzoru Stirlinga, a później przybliżając sumę całką. Dla analizy pierwszego z tych przybliżeń przyda nam się następujący fakt, pochodzący z [8].

Lemat 3.1. *Dla każdej liczby naturalnej n istnieje liczba θ_n, $0 < \theta_n \leqslant 1$, taka że*

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \exp\left\{\frac{\theta_n}{12n}\right\}$$

.

Uzupełniając obliczenia, które wykonaliśmy, aby otrzymać (1.7) o czynnik $\exp\left\{\frac{\theta_n}{12n}\right\}$ otrzymujemy

$$p_{2k,2n} = \frac{1}{\pi\sqrt{k(n-k)}} \exp\left\{\frac{\theta_{2k} - 4\theta_k}{24k} + \frac{\theta_{2(n-k)} - 4\theta_{n-k}}{24(n-k)}\right\} \tag{3.17}$$

Oznaczmy

$$d_{k,n} = \frac{1}{\pi\sqrt{k(n-k)}}$$

Naszym celem jest pokazanie, że $|p_{2k,2n} - d_{k,n}|$ jest małe. Z (3.17) dostajemy

$$\frac{p_{2k,2n}}{d_{k,n}} \leqslant \exp\left\{\frac{1}{24k} + \frac{1}{24(n-k)}\right\} = \exp\left\{\frac{n}{24k(n-k)}\right\}$$

oraz

$$\frac{p_{2k,2n}}{d_{k,n}} \geqslant \exp\left\{\frac{-4}{24k} + \frac{-4}{24(n-k)}\right\} = \exp\left\{-\frac{n}{6k(n-k)}\right\}.$$

Korzystając z powyższych i z nierówności $e^x - 1 \leqslant 2x$ (dla $x > 0$ i dostatecznie małych) oraz $1 - e^{-x} \leqslant x$ uzyskujemy

$$p_{2k,2n} - d_{k,n} \leqslant d_{k,n}\left(\exp\left\{\frac{n}{24k(n-k)}\right\} - 1\right) \leqslant d_{k,n}\frac{n}{12k(n-k)}$$

oraz

$$d_{k,n} - p_{2k,2n} \leqslant d_{k,n}\left(1 - \exp\left\{-\frac{n}{6k(n-k)}\right\}\right) \leqslant d_{k,n}\frac{n}{6k(n-k)}$$

co razem daje

$$|p_{2k,2n} - d_{k,n}| \leqslant d_{k,n}\frac{n}{6k(n-k)} = \frac{n}{6\pi\left(k(n-k)\right)^{\frac{3}{2}}}.$$

Ustalmy $\delta > 0$ i założymy dodatkowo, że $\delta \leqslant \frac{k}{n} \leqslant 1 - \delta$. Funkcja $k \mapsto (k(n-k))^{3/2}$ przyjmuje minimalną wartość na brzegu przedziału, w którym się ją rozpatruje, dlatego

$$|p_{2k,2n} - d_{k,n}| \leqslant \frac{n}{6\pi\left(\delta n(n-\delta n)\right)^{\frac{3}{2}}} = \frac{1}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}}.$$

Wielkość błędu aproksymacji w (3.3) oszacujemy w dwóch etapach. Na początek weźmy takie liczby a, b, że $\delta \leqslant a < b \leqslant 1 - \delta$. Wtedy

$$\begin{aligned}\left|\sum_{a\leqslant\frac{k}{n}\leqslant b} p_{2k,2n} - \sum_{a\leqslant\frac{k}{n}\leqslant b} d_{k,n}\right| &\leqslant \sum_{a\leqslant\frac{k}{n}\leqslant b} |p_{2k,2n} - d_{k,n}| \leqslant \sum_{a\leqslant\frac{k}{n}\leqslant b} \frac{1}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}} \\ &= \frac{\lceil bn - an\rceil}{6\pi n^2\left(\delta(1-\delta)\right)^{\frac{3}{2}}} \leqslant \frac{b-a}{3\pi n\left(\delta(1-\delta)\right)^{\frac{3}{2}}} \\ &\leqslant \frac{1}{3\pi n\left(\delta(1-\delta)\right)^{\frac{3}{2}}} = (\blacklozenge)\end{aligned}$$

Drugim źródłem niedokładności jest zastąpienie sumy przez całkę. Rozpatrzmy dowolną funkcję f, różniczkowalną w przedziale (a, b). Podzielmy (a, b) na odcinki długości $\frac{1}{n}$ i niech x_k będzie

dowolnym punktem w odcinku zawierającym $\frac{k}{n}$, a M_k i m_k odpowiednio maksymalną i minimalną wartością funkcji na tym odcinku. Korzystając z twierdzenia Lagrange'a dostajemy

$$\begin{aligned}\left|\int_a^b f(x)dx - \sum_{a\leqslant\frac{k}{n}\leqslant b}\frac{1}{n}f(x_k)\right| &\leqslant \sum_{a\leqslant\frac{k}{n}\leqslant b}\frac{1}{n}(M_i - m_i) = \sum_{a\leqslant\frac{k}{n}\leqslant b}\frac{1}{n^2}|f'(\xi_i)| \\ &\leqslant \sum_{a\leqslant\frac{k}{n}\leqslant b}\frac{1}{n^2}\sup_{a\leqslant x\leqslant b}|f'(x)| = \frac{\lceil bn-an\rceil}{n^2}\sup_{a\leqslant x\leqslant b}|f'(x)| \\ &\leqslant \frac{2(b-a)}{n}\sup_{a\leqslant x\leqslant b}|f'(x)|.\end{aligned} \tag{3.18}$$

W szczególności dla

$$f(x) = \frac{1}{\pi\sqrt{x(1-x)}}$$

mamy

$$f'(x) = \frac{2x-1}{2\pi(x(1-x))^{\frac{3}{2}}},$$

$$\frac{1}{n}f\left(\frac{k}{n}\right) = \frac{1}{n\pi\sqrt{\frac{k}{n}(1-\frac{k}{n})}} = \frac{1}{\pi\sqrt{k(n-k)}} = d_{k,n}$$

a stąd w interesującym nas przedziale

$$\left|\int_a^b f(x)dx - \sum_{a\leqslant\frac{k}{n}\leqslant b} d_{k,n}\right| \leqslant \frac{2}{n}\sup_{\delta<x<1-\delta}|f'(x)| = \frac{1-2\delta}{\pi n(\delta(1-\delta))^{\frac{3}{2}}} = (\bigstar)$$

W testach wykorzystamy partycję prostej $\mathcal{P}_{40}^{asin}$, dlatego u nas $\delta = \frac{1}{80} > 0.01$. Będzie ponadto $n \geqslant 2^{26}$. Zatem

$$\begin{aligned}(\blacklozenge) &\leqslant \frac{107.72}{n} \leqslant 1.6\cdot 10^{-6} \\ (\bigstar) &\leqslant \frac{316.69}{n} \leqslant 4.7\cdot 10^{-6}\end{aligned}$$

Ostatecznie

$$\begin{aligned}\left|\int_a^b f(x)dx - \sum_{a\leqslant\frac{k}{n}\leqslant b} p_{2k,2n}\right| &\leqslant \left|\int_a^b f(x)dx - \sum_{a\leqslant\frac{k}{n}\leqslant b} d_{k,n}\right| + \left|\sum_{a\leqslant\frac{k}{n}\leqslant b} d_{k,n} - \sum_{a\leqslant\frac{k}{n}\leqslant b} p_{2k,2n}\right| \\ &= (\blacklozenge) + (\bigstar) \leqslant 6.3\cdot 10^{-6}\end{aligned}$$

co uzasadnia wzór (3.3) w przypadku $\delta \leqslant a < b \leqslant 1-\delta$.

Musimy jeszcze zbadać niedokładność „na brzegu". Mamy

$$\begin{aligned}\left|\int_0^{\delta} f(x)dx - \sum_{0\leqslant\frac{k}{n}<\delta} p_{2k,2n}\right| &= \left|\int_0^{\frac{1}{2}} f(x)dx - \int_{\delta}^{\frac{1}{2}} f(x)dx - \sum_{0\leqslant\frac{k}{n}\leqslant\frac{1}{2}} p_{2k,2n} + \sum_{\delta\leqslant\frac{k}{n}\leqslant\frac{1}{2}} p_{2k,2n}\right| \\ &= \left|\frac{1}{2} - \int_{\delta}^{\frac{1}{2}} f(x)dx - \frac{1}{2} + \sum_{\delta\leqslant\frac{k}{n}\leqslant\frac{1}{2}} p_{2k,2n}\right| \\ &= \left|\int_{\delta}^{\frac{1}{2}} f(x)dx - \sum_{\delta\leqslant\frac{k}{n}\leqslant\frac{1}{2}} p_{2k,2n}\right| \leqslant 6.3\cdot 10^{-6},\end{aligned}$$

gdzie ostatnie przejście wynika z wcześniej przeprowadzonych rachunków. Powyższa analiza pokazuje, że błąd przybliżenia jest pomijalny.

CZĘŚĆ IV

Testy

Przebrnąwszy przez długie teorie i rozważania, przechodzimy do najciekawszej części pracy, czyli do implementacji opisanej metody testowania i sprawdzenia jej na powszechnie wykorzystywanych generatorach.

4.1 Implementacja

Testy oparte o prawa arcusa sinusa i iterowanego logarytmu przedstawione w części 3 zaimplementowałem w języku Julia. Napisany program stanowi równie ważną część tej pracy.

Julia jest to nowoczesny (prace nad nim rozpoczęto w roku 2009) i szybki język programowania przeznaczony do obliczeń naukowych. Jest to więc narzędzie bardzo dobrze nadające się do naszych eksperymentów.

Powstały skrypt można wykorzystać do przetestowania dowolnego generatora. Trzeba zadbać jedynie o to, aby wyjście z GLP było przekazywane do programu testującego w odpowiednim formacie. Poniżej opisujemy dokładniej jak to zrobić. Jednak najpierw omówmy kilka prozaicznych kwestii, które wpłynęły na końcową architekturę programu.

4.1.1 Uwagi praktyczne

Najwygodniejszym sposobem testowania GLP byłoby prawdopodobnie podzielenia zadania na dwa etapy. W pierwszej kolejności użylibyśmy GLP do wygenerowania m sekwencji bitów długości n, które zapisalibyśmy do pliku. Następnie program testujący wczytałby ten plik i wyliczył odpowiednie statystki, rozstrzygnął czy dane są losowe, itd. Niestety nie da się zaprojektować systemu w taki sposób, aby opisany tryb pracy był możliwy.

Powodem jest olbrzymia ilość używanych danych. W celu przetestowania większości GLP wykorzystujemy je do wygenerowania $m = 10000$ ciągów bitów długości $n = 2^{34}$. Rodzi to niebagatelne problemy implementacyjne. Pojedynczy ciąg ma rozmiar $2GB$, więc w pamięci przeciętnego komputera nie zmieści się ich nawet kilka. Wymusza to przetwarzanie ciągów jednego po drugim – po obliczeniu statystyk dla jednego ciągu należy natychmiast zwolnić pamięć dla kolejnego. Większym problemem jest jednak fakt, że dane zajmują łącznie $20TB$, więc zapisanie

ich do pliku jest możliwe na mało której maszynie – co uzasadnia dlaczego pomysł przedstawiony w poprzednim akapicie jest niewykonalny.

Jedyną możliwością jest postępowanie w taki sposób, by GLP i skrypt funkcjonowały naprzemiennie. GLP generuje sekwencję bitów, program testujący ją analizuje, po czym pamięć zostaje zwolniona i możemy powtórzyć procedurę. Na szczęście istnieje prosty środek pozwalający zorganizować obliczenia w ten sposób. Mowa tu o uniksowym mechanizmie potoku. Wystarczy wyniki z GLP przekazywać na standardowe wyjście, które będzie połączone ze standardowym wejściem programu testującego. System operacyjny sam zadba o to, żeby oba procesy działały na zmianę.

Kolejną istotną kwestią jest tryb zapisu danych do strumienia wejścia-wyjścia. Zwróćmy uwagę, że gdybyśmy przekazywali dane w trybie tekstowym, to przekazywany łańcuch znaków zajmowałby 8 razy więcej miejsca niż jest to potrzebne – każdy bit byłby reprezentowany jako jednobajtowy znak `'0'` lub `'1'`. Byłoby to fatalne podejście, gdyż, po pierwsze, prawdopodobnie nie starczyłoby pamięci do zapisania całego ciągu. Po drugie, nawet gdyby pamięci było wystarczająco, to zapisywanie i wczytywanie tych danych do i ze strumienia kilkukrotnie wydłużyłoby (i tak bardzo długi) czas pracy programu. Dlatego dane przekazujemy w trybie binarnym.

Ostatnią rzeczą, na którą warto zwrócić uwagę, jest opłacalność oddzielenia obliczania charakterystyk ciągów (S_n^{asin} lub S_n^{lil}) od ich porównywania z teoretycznym rozkładem. Lepiej jest zapisywać wartości charakterystyk do pliku, a następnie oddzielnym skryptem badać zgodność z oczekiwanym rozkładem. Dzięki takiemu podejściu można uruchomić instancje generatora na wielu maszynach (zadbawszy o to by korzystały one z innych ziaren). Po zakończeniu obliczeń łatwo jest scalić wyniki i wyznaczyć sumaryczne statystyki.

4.1.2 Użycie programu

Uruchomienie. Załóżmy, że dysponujemy programem `gen.bin` generującym sekwencje pseudolosowych bitów. Powiedzmy, że przyjmuje on z linii poleceń dwa argumenty oznaczające liczbę i logarytm długości generowanych ciągów. Opiszemy jak przetestować ten generator.

Z perspektywy użytkownika najważniejsze jest, że punkt startowy programu testującego jest w pliku `Tester.jl`. Program wczytuje ze standardowego wejścia strumień bitów. Do pliku podanego w linii poleceń zapisuje wyniki swoich obliczeń, tj. wartości S_n^{asin} lub S_n^{lil}. Po zakończeniu działania używamy skryptu `ResultReader.jl` do wyznaczenia rozkładów empirycznych ze wzorów (3.5) i (3.16). Skrypt następnie wyliczy odległości d^{tv} i d^{sep} oraz p-wartości testu zgodności χ^2.

Skrypt `Tester.jl` przyjmuje z linii poleceń następujące argumenty:

- `testType` – słowo `asin` lub `lil` oznaczające którą z charakterystyk S_n^{asin} i S_n^{lil} obliczamy,
- `nrOfCheckPoints` – Dla ciągów długości n do pliku wynikowego zapisujemy nie tylko $S_n^\bullet$, ale również $S_{n/2}^\bullet$, $S_{n/4}^\bullet$, itd. `nrOfCheckPoints` to liczba tych wartości.
- `pathToFile` – nazwa pliku do którego zapisujemy wyniki.

Podobne argumenty ma skrypt `ResultReader.jl`:

- `testType` – słowo `asin` lub `lil` mówiące, z którą teoretyczną miarą należy porównywać wyniki,
- `logLength` – liczba naturalna oznaczająca logarytm długości ciągów wykorzystanych do otrzymania podanych wyników,
- `pathToFile` – nazwa pliku z którego odczytujemy wyniki.

Wróćmy do generatora `gen.bin`. Możemy go przetestować wywołując z konsoli przykładowo

```
$ ./gen.bin 1000 25 | julia Main.jl asin 5 wyniki.csv
```

W ten sposób testujemy generator `gen.bin` na podstawie 1000 ciągów zawierających 2^{25} bitów przy użyciu charakterystyki S_n^{asin}. Wyniki znajdą się w pliku `wyniki.csv`. Po zakończeniu obliczeń możemy je zinterpretować poleceniem

```
$ julia ResultReader.jl asin 25 wyniki.csv
```

Podobnie można przetestować dowolny inny generator, pamiętając, że jego wyjście musi być zapisane zgodnie z formatem opisanym poniżej.

Testowanie własnych GLP. Z punktu widzenia osoby, która chce wykorzystać program do sprawdzenia swojego GLP ważne jest co dokładnie ma się znaleźć w strumieniu wejściowym programu testującego. Format jest prosty:

- Pierwsze 8 bajtów strumienia zawiera 64-bitową wartość typu `integer` oznaczającą liczbę ciągów bitów.
- Kolejne 8 bajtów zawiera 64-bitową wartość typu `integer` oznaczającą długość pojedynczego ciągu.
- Dalej następuje $m \cdot n$ bitów danych, przy czym każde kolejne n bitów traktowane jest jako jeden ciąg używany w testach.

Uwaga. Nie ma żadnych „specjalnych" bitów oznaczających przerwy między ciągami, ani niczego podobnego. Po wczytaniu n bitów jednego ciągu, kolejny bit jest uważany za pierwszy bit następnego ciągu.

4.2 Wyniki

Przyjrzyjmy się wynikom testowania kilku znanych GLP. Każdy z nich testowany był przy użyciu $m = 10000$ ciągów. Długością używanej sekwencji było w większości przypadków $n = 2^{34}$.

Wartości $S_\bullet^{asin}$ lub $S_\bullet^{lil}$ obliczone zostały nie tylko dla całych ciągów, ale także dla podciągów długości $n/2$, $n/4$, itd. Pozwala to obserwować zgodność z pożądaną miarą na różnych etapach. Do badania zgodności użyto partycji $\mathcal{P}_{40}^{lil}$ dla testów korzystających z charakterystyki S_n^{lil} i partycji $\mathcal{P}_{40}^{asin}$ dla testów korzystających z charakterystyki S_n^{asin}.

Do testów wykorzystano własne implementacje rozpatrywanych GLP (za wyjątkiem MT19937). Program wywołujący GLP korzystał losowych ziaren pobranych ze strony www.random.org. Przed wygenerowaniem każdej kolejnej sekwencji bitów ustawiano nowe ziarno generatora.

Podkreślmy, że otrzymanie poniższych wyników nie było błahostką. Przetestowanie jednego GLP na 10 komputerach w pracowni Instytutu Informatyki zajmowało około półtorej doby.

4.2.1 RANDU

Na rozgrzewkę rozpoczynamy od generatora, który od dawna nie jest w użyciu. RANDU powstał na początku lat 60. Wspominaliśmy już o nim w paragrafie 2.3, dając go jako przykład generatora fatalnie oblewającego test spektralny.

RANDU jest to po prostu $MCG(2^{31}, 65539)$. Wybór liczby 65539 wydawał się dobry, gdyż $65539 = 2^{16} + 3$, co umożliwiało szybkie, sprzętowe wykonanie mnożenia.

Rezultaty testowania RANDU przedstawione są w Tabelach 4.1 i 4.2. Pierwszy wiersz w tabelach oznacza długości podciągów. Dla podciągu długości n_k została obliczona teoretyczna miara μ_{n_k} według wzoru (3.4) lub (3.15) oraz empiryczna miara ν_k według wzoru (3.5) lub (3.16). Każda kolumna zawiera wartości $d^{tv}(\mu_{n_k}, \nu_{n_k})$, $d^{sep}(\mu_{n_k}, \nu_{n_k})$, $d^{sep}(\nu_{n_k}, \mu_{n_k})$ oraz p-wartość statystyki (3.12). **Dla wszystkich GLP tabele z wynikami sporządzono w analogiczny sposób.**

Tabela 4.1: Wyniki testu arcusa sinusa dla generatora RANDU.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.4604	0.4616	0.4637	0.4670	0.4695	0.4697
sep1	0.6017	0.6646	0.6229	0.6138	0.5934	0.6453
sep2	0.8659	0.8662	0.8667	0.8675	0.8681	0.8682
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Tabela 4.2: Wyniki testu iterowanego logarytmu dla generatora RANDU.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.4955	0.496	0.4965	0.4969	0.4973	0.4977
sep1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Jak widać nie trzeba bardzo długich ciągów, aby przekonać się, że wyjścia RANDU nie można uznać za losowe. Nie bez przyczyny został on szybko wyparty przez lepsze generatory.

4.2.2 Biblioteczny rand w Microsoft Visual C++

Funkcja `rand` w Microsoft Visual C++ opiera się o $LCG(2^{32}, 214013, 2531011)$. Od zwykłego LCG różni się jednak tym, że zwracane są jedynie bity na pozycjach 30..16.

Jest to jeden z generatorów testowanych w [10]. Podobnie jak autorzy tej pracy odrzucamy najmniej istotne 7 bitów liczb zwracanych przez funkcję `rand` (czyli używamy tylko bitów 30..23 liczby otrzymanej z LCG). Wyniki testów zestawiono w Tabelach 4.3 i 4.4.

Pamiętajmy, że okresem tego LCG jest 2^{31}, a z jednego wywołania otrzymujemy $8 = 2^3$ bitów. Oznacza to, że aby dostać ciąg bitów długości 2^{34} przechodzimy przez pełen cykl generatora. Musieliśmy więc każdy układ 8 bitów wygenerować tyle samo razy, skąd wniosek, że po 2^{34} krokach błądzenie losowe zawsze wracało do zera. Widać to wyraźnie w wynikach testowania

Tabela 4.3: Wyniki testu arcusa sinusa dla generatora w Visual C++.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0289	0.0387	0.0615	0.0768	0.0848	0.0928	0.0965	0.1870	0.2093
sep1	0.2030	0.1548	0.1952	0.2605	0.3664	0.4210	0.5528	0.6533	0.8163
sep2	0.1780	0.2260	0.2524	0.2364	0.2841	0.3160	0.4336	0.6112	0.4328
p-val	0.0128	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Tabela 4.4: Wyniki testu iterowanego logarytmu dla generatora w Visual C++.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0350	0.0512	0.0938	0.1234	0.1672	0.2423	0.3100	0.4991	0.9500
sep1	0.2831	0.7419	0.9731	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.2670	0.1416	0.2635	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

charakterystyką S_n^{lil}. Dla $n = 2^{34}$ wszystkie obserwacje wpadają do przedziału $[0, 0.05)$. Jego teoretyczna miara wynosi ~ 0.05, a stąd $d^{tv}(\mu_n^{lil}, \nu_n^{lil}) \approx 0.95$.

Ktoś złośliwy mógłby powiedzieć, że w bardzo złożony sposób udowodniliśmy oczywisty fakt, że generatory o krótkich okresach nie nadają się do generowania dużych ilości liczb pseudolosowych. Zauważmy jednak, że analizowany generator ma problemy już przy $n = 2^{26}$, a do wygenerowania błądzenia tej długości potrzeba tylko $2^{23}/2^{31} = 1/2^8 \approx 0.4\%$ całego okresu. Trudno jest więc ocenić ten generator dobrze, nawet w kategorii generatorów o krótkich okresach.

Co zaskakujące, NIST Test Suite uznaje ten generator za poprawny, jak zauważyli autorzy [10].

4.2.3 Biblioteczny rand w Borland C/C++

Funkcja `rand` w środowisku Borland jest implementacją $LCG(2^{32}, 22695477, 1)$, która, podobnie jak `rand` w Visual C++, zwraca jedynie bity 30..16.

Postępując tak jak w poprzednim przykładzie bierzemy do testów tylko 8 najistotniejszych bitów zwróconej liczby.

Tabela 4.5: Wyniki testu arcusa sinusa dla generatora w Borland C/C++.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0384	0.0502	0.0697	0.0861	0.1394	0.1562	0.1231	0.1466	0.2148
sep1	0.1293	0.1840	0.2513	0.3733	0.5485	0.5149	0.5696	0.6873	0.8219
sep2	0.2024	0.2120	0.2096	0.3137	0.3702	0.3730	0.3799	0.5167	0.4009
p-val	0.0001	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Wyniki przedstawione są w Tabelach 4.5 i 4.6. Są one bardzo podobne do wyników generatora z Visual C++ i tyczą się ich te same uwagi.

Tabela 4.6: Wyniki testu iterowanego logarytmu dla generatora w Borland C/C++.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0684	0.1002	0.1395	0.1940	0.3187	0.4136	0.4685	0.5911	0.9500
sep1	0.6752	0.7357	0.9193	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.1610	0.2565	0.2978	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

4.2.4 Biblioteczny rand w BSD libc

Funkcja `rand` z biblioteki systemu BSD używała kiedyś implementacji $LCG(2^{31}, 1103515245, 12345)$ i w przeciwieństwie do dwóch poprzednich generatorów zwraca wszystkie bity generowanych liczb – i do tego testu użyliśmy ich wszystkich. Tabele 4.7 i 4.8 obrazują dlaczego generator zmieniono.

Tabela 4.7: Wyniki testu arcusa sinusa dla starego generatora z BSD libc.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.0883	0.0980	0.0862	0.0951	0.0936	0.1081
sep1	0.3402	0.3979	0.4111	0.4980	0.4346	0.4751
sep2	0.3634	0.4198	0.3053	0.2805	0.3424	0.4070
p-val	0.1829	0.0326	0.1874	0.1767	0.1181	0.0051

Tabela 4.8: Wyniki testu iterowanego logarytmu dla starego generatora z BSD libc.

n	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}
tv	0.1956	0.2191	0.2472	0.2539	0.2464	0.2707
sep1	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Wystarczają stosunkowo krótkie ciągi, by zobaczyć, że wyjście funkcji `rand` z BSD nie jest losowe. Przyczyną tak złych wyników jest fakt, że to LCG wykonuje obliczenia modulo 2^{31}. Fakt 2.2 mówi, że w takiej sytuacji okres d najmniej znaczących bitów wynosi 2^d. W efekcie liczba zer i jedynek jest zbyt wyrównana co łatwo wyłapują testy oparte o własności błądzenia przypadkowego. Trzeba jednak przyznać, że charakterystyka S_n^{lil} pokazuje to zdecydowanie wyraźniej.

4.2.5 Biblioteczny rand w GLIBC

Funkcja `rand` z GNU C Library korzysta z bardziej skomplikowanego generatora od testowanych do tej pory. Jego stan opisany jest przez 34 liczby $x_i, x_{i+1}, \ldots, x_{i+33}$. Generator inicjowany jest

ziarnem s, $0 \leqslant s \leqslant 2^{31}$, zaś początkowym stanem jest

$$\begin{aligned} x_0 &= s \\ x_i &= 16807 x_{i-1} \mod (2^{31}-1), && \text{gdy } 0 < i < 31 \\ x_i &= x_{31-i}, && \text{gdy } i \in \{31, 32, 33\}. \end{aligned}$$

Kolejne wartości x_i wyznaczane są ze wzoru

$$x_i = (x_{i-3} + x_{i-31}) \mod 2^{32}.$$

Przy k-tym wywołaniu generator zwraca $x_{k+343} \gg 1$.

Tabela 4.9: Wyniki testu arcusa sinusa dla standardowego generatora w GCC.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0287	0.0217	0.0264	0.0228	0.0202	0.0289	0.0237	0.0254	0.0230
sep1	0.1157	0.1249	0.1629	0.1186	0.1661	0.1717	0.1306	0.1316	0.1405
sep2	0.1754	0.1909	0.1436	0.1869	0.1110	0.1344	0.1786	0.1262	0.1879
p-val	0.0649	0.5511	0.2565	0.4887	0.7967	0.1115	0.5880	0.2599	0.3930

Tabela 4.10: Wyniki testu iterowanego logarytmu dla standardowego generatora w GCC.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0185	0.0243	0.0242	0.0244	0.0210	0.0220	0.0330	0.0237	0.0221
sep1	0.2593	0.1640	0.1844	0.2573	0.2010	0.1853	0.2882	0.4129	0.2563
sep2	0.0807	0.2133	0.2299	0.2060	0.1839	0.2361	0.3456	0.2078	0.1405
p-val	0.9627	0.4878	0.5225	0.3022	0.6978	0.5382	0.0009	0.4903	0.7901

Do testów brane były wszystkie 31 bitów zwracanych przez GLP. Na podstawie wyników zebranych w Tabelach 4.9 i 4.10 można stwierdzić, że test arcusa sinusa nie daje podstaw do odrzucania hipotezy o losowości sekwencji generowanych przez analizowany GLP. W przypadku testu iterowanego logarytmu, rezultaty dla $n = 2^{34}$ i $n = 2^{33}$ również sugerowałyby, że GLP jest dobry. Jednakże dla $n = 2^{32}$ obserwujemy coś dziwnego, mamy bardzo niską p-wartość wynoszącą około 1/1000. Może to być kwestia przypadku – z prawdopodobieństwem 1/1000 zdarzyłoby się to nawet generatorowi liczb prawdziwie losowych. Popatrzmy jednak na p-wartości dla dziesięciu 1000-elementowych podzbiorów danych. Wynoszą one: 0.9313, 0.0949, 0.8859, 0.1739, 0.3675, 0.0334, 0.0321, 0.1824, 0.0017, 0.6205. Zauważmy, że aż 4 p-wartości są mniejsze niż 0.1. W przypadku ciągów prawdziwie losowych zdarzenie, że 4 lub więcej otrzymanych p-wartości znajdzie się w tym przedziale wynosi

$$1 - \sum_{i=0}^{3} \binom{10}{i} \left(\frac{1}{10}\right)^i \left(1 - \frac{1}{10}\right)^{10-i} \approx 0.0016$$

To przemawia za tym, by generator z GLIBC również uznać za podejrzany. Jednak mimo tego jest to wyraźnie najlepsza implementacja funkcji `rand`.

4.2.6 Minstd

Minstd (skrót od ang. *minimal standard generator*) jest to MCG z parametrami zasugerowanymi przez Parka i Millera. Ich celem było zaprojektowanie prostego generatora, który wprawdzie nie byłby doskonały, ale za to szybki, prosty w implementacji i nadający się do większości niespecjalistycznych zastosowań. W efekcie zaproponowali $MCG(2^{31}-1, 16807)$. Do generowania ciągów zerojedynkowych użyto ośmiu najbardziej znaczących bitów zwracanych przez GLP. Rezultaty testów pokazują Tabele 4.11 i 4.12.

Tabela 4.11: Wyniki testu arcusa sinusa dla generatora Minstd z mnożnikiem 16807.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0321	0.0321	0.0402	0.0725	0.0917	0.1116	0.1648	0.2003	0.2109
sep1	0.1172	0.1686	0.1407	0.2985	0.2880	0.6670	0.5203	0.7345	0.8486
sep2	0.1302	0.1490	0.2420	0.3986	0.4051	0.3735	0.3822	0.5352	0.3894
p-val	0.0360	0.0203	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Tabela 4.12: Wyniki testu iterowanego logarytmu dla generatora Minstd z mnożnikiem 16807.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0370	0.0622	0.0823	0.1120	0.1877	0.2002	0.3538	0.3194	0.9500
sep1	0.5035	0.5465	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.1444	0.2366	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Kilka lat później później Park i Miller zasugerowali, że lepiej używać mnożnika 48271. Czy zmiana mnożnika poprawiła wyniki testów widać w Tabelach 4.13 i 4.14.

Tabela 4.13: Wyniki testu arcusa sinusa dla generatora Minstd z mnożnikiem 48271.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0318	0.0430	0.0477	0.0982	0.1303	0.0918	0.0848	0.1131	0.2089
sep1	0.1607	0.1467	0.1901	0.3803	0.4280	0.3845	0.4583	0.3975	0.8079
sep2	0.1828	0.1719	0.2120	0.2548	0.3449	0.3179	0.3192	0.3186	0.4579
p-val	0.0115	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Tabela 4.14: Wyniki testu iterowanego logarytmu dla generatora Minstd z mnożnikiem 48271.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0372	0.0563	0.0590	0.2573	0.2879	0.3408	0.3167	0.3434	0.9500
sep1	0.3298	0.4839	0.9328	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
sep2	0.3843	0.1850	0.2207	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
p-val	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

W przypadku test arcusa sinusa odległość total variation poprawiła się po zmianie mnożnika. Jednak w teście iterowanego logarytmu wyszło dokładnie odwrotnie, trudno jest więc powiedzieć czy zmiana mnożnika wniosła istotną poprawę. W każdym razie test χ^2 w obu przypadkach pokazuje, że Minstd to nie jest dobry generator.

Mimo swoich niedoskonałości Minstd wszedł w zakres biblioteki standardowej C++11. Implementacjami są klasy `std::minstd_rand0` (z mnożnikiem 16807) oraz `std::minstd_rand` (z mnożnikiem 48271).

4.2.7 CMRG

CMRG (od ang. *combined multiple recursive generator*) jest jednym z generatorów mieszanych, omawianych w paragrafie 2.1. Zwraca on liczby Z_n wyznaczone według wzoru

$$\begin{aligned} Z_n &= X_n - Y_n \mod 2^{31} - 1 \\ X_n &= 63308X_{n-2} - 183326X_{n-3} \mod 2^{31} - 1 \\ Y_n &= 86098Y_{n-1} - 539608Y_{n-3} \mod 2^{31} - 2000169 \end{aligned} \tag{4.1}$$

Do testów braliśmy bity 15..8 zmiennej Z_n. Tabele 4.15 i 4.16 przedstawiają ich wyniki.

Tabela 4.15: Wyniki testu arcusa sinusa dla generatora CMRG.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0191	0.0234	0.0235	0.0239	0.0232	0.0228	0.0218	0.0268	0.0272
sep1	0.1135	0.2090	0.1187	0.1450	0.1607	0.1191	0.1232	0.1592	0.2084
sep2	0.1402	0.0900	0.1084	0.1477	0.1934	0.1666	0.0931	0.1353	0.1158
p-val	0.9752	0.4101	0.8062	0.7402	0.4043	0.6841	0.9821	0.2394	0.1343

Tabela 4.16: Wyniki testu iterowanego logarytmu dla CMRG.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0218	0.0241	0.0272	0.0203	0.0231	0.0251	0.0330	0.0233	0.0234
sep1	0.2336	0.3273	0.1803	0.2551	0.3085	0.1690	0.3146	0.3310	0.2829
sep2	0.3450	0.2105	0.1234	0.1032	0.1901	0.1658	0.2302	0.2486	0.1761
p-val	0.6046	0.2689	0.2030	0.8653	0.4581	0.3966	0.0128	0.4106	0.6803

Na pierwszy rzut oka widać, że mamy tu do czynienia z generatorem o dłuższym okresie. Wyniki nie dają żadnych podstaw do odrzucenia hipotezy o losowości ciągów bitów. Trzeba tu jednak odnotować, że w [5] przedstawiono silne dowody, że wyjście tego generatora nie jest dobre.

4.2.8 Mersenne Twister

Generator Mersenne Twister został przetestowany przy użyciu implementacji dostępnej w języku C++11. Wykorzystana została wersja 64-bitowa (czyli klasa `std::mt19937_64`). Jest ona

nieznaczną modyfikacją 32-bitowej wersji opisanej w paragrafie 2.1, dostosowaną do maszyn operujących na 8-bajtowych słowach. Do testów użyto wszystkie 64-bity zwracana przy jednokrotnym wywołaniu generatora.

Tabela 4.17: Wyniki testu arcusa sinusa dla generatora MT19967-64.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0271	0.0245	0.0226	0.0281	0.0276	0.0228	0.0271	0.0230	0.0255
sep1	0.1421	0.1606	0.1167	0.1824	0.1938	0.1571	0.1760	0.1397	0.1573
sep2	0.1133	0.1195	0.1056	0.1399	0.1256	0.1287	0.1391	0.0994	0.1444
p-val	0.2755	0.6823	0.8804	0.0801	0.1267	0.7343	0.1094	0.7596	0.2029

Tabela 4.18: Wyniki testu iterowanego logarytmu dla generatora MT19967-64.

n	2^{26}	2^{27}	2^{28}	2^{29}	2^{30}	2^{31}	2^{32}	2^{33}	2^{34}
tv	0.0254	0.0264	0.0238	0.0234	0.0266	0.0265	0.0290	0.0299	0.0215
sep1	0.1853	0.2147	0.1689	0.3093	0.1887	0.2015	0.2877	0.2318	0.1953
sep2	0.2841	0.1696	0.1917	0.1282	0.2017	0.2492	0.1842	0.1541	0.1716
p-val	0.2483	0.3690	0.5965	0.5066	0.2196	0.1495	0.0499	0.0189	0.8170

Wyniki zgromadzone w Tabelach 4.17 i 4.18 pokazują, że nie bez powodu Mersenne Twister jest najpopularniejszym generatorem. Wyniki nie dają powodów do podejrzeń, że wyjście generatora nie jest losowe.

4.2.9 Hipotetyczny, wadliwy generator

Przedstawiając NIST Test Suite w paragrafie 2.3 wspomnieliśmy o pewnej jego wadzie, tkwiącej nieodłącznie w zastosowanym podejściu. Ten zestaw testowy skupia się jedynie na jakości poszczególnych sekwencji bitów otrzymanych z GLP, nie ocenia natomiast jakości zbioru wszystkich ciągów „jako całości". Moglibyśmy mieć do czynienia z generatorem, który zazwyczaj generuje sekwencje doskonale imitujące idealny ciąg losowych bitów, ale z jakiegoś powodu, np. dla niektórych wartości ziaren, produkuje wyjście ewidentnie nielosowe. Ten wadliwy ciąg zapewne został by rozpoznany przez NIST Test Suite jako nielosowy, jednak nie przeważyłoby to do uznania całego generatora za zły.

Żeby sprawdzić jak z taką sytuacją mogą radzić sobie testy iterowanego logarytmu i arcusa sinusa, przyjrzyjmy się następującemu eksperymentowi. Wykorzystamy generator, który

- dla co setnego ziarna będzie produkował ciąg opisany wyrażeniem regularnym 10(0110)*01,
- w pozostałych przypadkach zwraca wyjście generatora MT19997-64.

Rezultaty zebrane są w Tabelach 4.19 i 4.20.

Od razu widać, że test arcusa sinusa jednoznacznie nakazuje odrzucić hipotezę o losowości bitów generowanych przez GLP.

Dlaczego w tym wypadku test iterowanego logarytmu się nie sprawdził? Błądzenie (zdecydowanie nieprzypadkowe) opisane wyrażeniem 10(0110)*01 oscyluje wokół osi OX jak sinusoida.

Tabela 4.19: Wyniki testu arcusa sinusa dla opisanego generatora.

n	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
tv	0.0326	0.0241	0.0281	0.0285	0.0298	0.0339	0.0298	0.0285	0.0276
sep1	0.1466	0.1754	0.1529	0.1124	0.1249	0.1752	0.1594	0.1666	0.1257
sep2	0.3255	0.334	0.345	0.4212	0.3608	0.3582	0.3556	0.3925	0.3901
p-val	0.0000	0.0031	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000

Tabela 4.20: Wyniki testu iterowanego logarytmu dla opisanego generatora.

n	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
tv	0.0239	0.0303	0.0255	0.0269	0.0241	0.0280	0.0248	0.0294	0.0229
sep1	0.1929	0.2834	0.1541	0.2018	0.2327	0.1586	0.1631	0.2654	0.1689
sep2	0.1625	0.1517	0.1663	0.1469	0.1381	0.1648	0.2106	0.1667	0.1815
p-val	0.4073	0.0199	0.2584	0.1250	0.5742	0.1250	0.2703	0.0436	0.5789

Oznacza to, że frakcja czasu przewagi liczby jedynek nad liczbą zer jest równa dokładnie $\frac{1}{2}$. Dzięki temu wartość charakterystyki S_n^{asin} wpada do najmniej prawdopodobnego przedziału w partycji $\mathcal{P}^{asin}$ w prawie $0.01m$ przypadkach częściej niż dla dobrego generatora. Wystarcza to by wartość statystyki (3.12) bardzo urosła. Tymczasem charakterystyka S_n^{lil} przyjmuje dla ciągu $10(0110)^*01$ wartość zero, a tym samym wpada do najbardziej prawdopodobnego przedziału. Dlatego testowi χ^2 trudniej jest to wykryć.

Gdybyśmy jednak w powyższym eksperymencie wykorzystali $m = 100000$ ciągów, to również test iterowanego logarytmu zauważyłby zbyt duże odstępstwo. Wypływa stąd wniosek, że w celu wykrywania tego rodzaju wad generatorów lepiej postawić na liczbę ciągów, a nie na ich długość.

4.3 Podsumowanie

Przedstawiliśmy stosunkowo nowatorską metodę testowania GLP. Od standardowych metod różni się tym, że wyjście GLP traktuje jak ciąg bitów, a nie jako ciąg liczb. Daje to szerokie pole do wymyślania testów opartych o błądzenie przypadkowe.

Wyniki z poprzedniego paragrafu pokazują, że testy arcusa sinusa oraz iterowanego logarytmu dobrze wykrywają regularności w generatorach opartych na kongruencjach liniowych. Są jednak generatory, o których wiemy, że są wadliwe, a zaproponowana metoda tego nie wyłapuje, przykładem jest CMRG. Przedstawione testy, tak jak wszystkie inne testy statystyczne, biorą pod lupę tylko pewien aspekt analizowanego zagadnienia. Znalezienie odchyleń w tym aspekcie jest oczywiście dowodem wadliwości całego generatora, jednak pomyślny wynik testu mówi tak naprawdę tylko tyle, że „pod rozpatrywanym względem nie dopatrzono się nieprawidłowości".

Przedstawioną metodę trudno byłoby uznać za przełomową – z pewnością nie jest ona uniwersalna, gdyż, jak zauważyliśmy powyżej, nie wykryjemy nią wszystkich możliwych usterek generatorów. Zdaje się jednak, że testy oparte o własności błądzenia przypadkowego byłyby dobrym uzupełnieniem standardowych pakietów testowych takich jak np. NIST Test Suite, zwłaszcza, że mają potencjał do wyłapywania problemów nieco innej natury, tak jak to wyjaśniliśmy w paragrafie 4.2.9.

Z pewnością można wymyślić więcej testów działających na podobnej zasadzie jak test arcusa sinusa czy test iterowanego logarytmu. Warto pracować nad nowymi testami wyłapującymi międzybitowe zależności, które są niezauważane przez dotychczasowe metody. Przysłużyłoby się to utworzeniu szczelniejszej paczki testów. Jest to ciekawe zagadnienie, nad którym z pewnością opłaca się prowadzić dalsze badania.

Bibliografia

[1] S. Asmussen, P. Glynn, *Stochastic Simulation: Algorithms and Analysis*, Springer, New York, 2007.

[2] W. Feller, *Wstęp do rachunku prawdopodobieństwa*, Wydanie piąte, PWN, Warszawa, 1987.

[3] T. Hull, A. Dobell, Random number generators, *SIAM Review* **4**, 1962, s. 230-254.

[4] J. Jakubowski, R. Sztencel, *Wstęp do teorii prawdopodobieństwa*, Wydanie IV, Script, Warszawa, 2010.

[5] C. Kim, G.H. Choe, D.H. Kim, Tests of randomness by the gambler's ruin algorithm, *Applied Mathematics and Computation* **199**, 2008, s. 195-210.

[6] D. Knuth, *The Art of Computer Programming volume 2: Seminumerical algorithms (2nd ed.)*, Addison-Wesley, 1981.

[7] P. L'Ecuyer, Efficient and Portable Combined Random Number Generator, *Communications of the ACM* **31**, 1988, s. 742–749, 774.

[8] F. Leja, *Rachunek różniczkowy i całkowy ze wstępem do równań różniczkowych*, Wydanie dziesiąte, PWN, Warszawa, 1969.

[9] M. Matsumoto, T. Nishimura, Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation* **8** (1), 1998, s. 3–30.

[10] Y. Wang, T. Nicol, On Statistical Distance Based Testing of Pseudo Random Sequences and Experiments with PHP and Debian OpenSSL, *Computers & Security* **53**, 2015, s. 44–64.