Generiranje pseudoslučajnih brojeva

Marinčić, Dino

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike

Dino Marinčić

Generiranje pseudoslučajnih brojeva

Diplomski rad

Osijek, 2019.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni diplomski studij matematike

Dino Marinčić

Generiranje pseudoslučajnih brojeva

Diplomski rad

Voditelj: doc. dr. sc. Danijel Grahovac

Osijek, 2019.

Sadržaj

1. Uvod

Primjena slučajnih brojeva u svakodnevnom životu je veoma široka. Svoju primjenu nalaze u raznim područjima kao što su znanost, programiranje, kriptografija, izrada računalnih igrica, igre na sreću i još mnogo toga. Kako su tradicionalne metode za generiranje slučajnih brojeva poput bacanja novčića ili igraće kocke spora i nepraktična, mijenjaju ih računala. Problem koji se nameće je taj što su računala deterministički strojevi i kao takvi ne mogu generirati zaista slučajne brojeve.

Na početku rada razradit ćemo razlike između koncepta slučajnih i pseudoslučajnih brojeva, objasniti principe generatora slučajnih i pseudoslučajnih brojeva i pozabaviti se jednima od najstarijih i primitivnijih algoritama za generiranje pseudoslučajnih brojeva. U središnjem dijelu rada predstavit ćemo Mersenne Twister algoritam te pokazati njegovu implementaciju u R jeziku. Zatim ćemo izgenerirati testne uzorke iz svakog od predstavljenih algoritama te se pozabaviti njihovim svojstvima. Predstavit ćemo i razne statističke testove kojima testiramo kvalitetu generatora pseudoslučajnih brojeva. Završni dio rada rezerviran je za generiranje pseudoslučajnih brojeva iz proizvoljnih distribucija.

Sve algoritme i statističke testove koje ćemo predstaviti u ovom radu isprogramirat ćemo u R jeziku. Izvorni kod ćemo navesti gdje god to bude korisno.

2. Generatori slučajnih brojeva

Slučajnost možemo opisati kao odsustvo uzorka, pravila ili predvidivosti nekog događaja. Kao primjere stvarno slučajnih događaja većinom navodimo prirodne fenomene poput atmosferskog šuma ili količine radijacije uzrokovane svemirskim zračenjem. Vjerojatno najpoznatiji primjer onoga što je generalno prihvaćeno kao zaista slučajno jest koncept radioaktivnog raspada. Prema kvantnoj teoriji smatra se da je nemoguće predvidjeti kada će doći do raspada atoma, bez obzira koliko dugo je prije toga postojao. Proces generiranja slučajnih brojeva zasnovan na tom konceptu mogli bi konstruirati na sljedeći način. Zamislimo da imamo proizvoljno mnogo atoma na jednom mjestu koje promatramo. Sa S_1 označimo vrijeme proteklo od početka mjerenja t_0 do raspada prvog atoma t_1, sa S_2 vrijeme proteklo od raspada prvog atoma t_1 do raspada drugog atoma t_2 i općenito neka je $S_i = t_i - t_{i-1}$ vrijeme proteklo od raspada $(i-1)$-vog atoma do raspada i-tog atoma. Definiramo varijablu

$$X_i = \begin{cases} 1, & \text{za } S_{2i-1} < S_i \\ 0, & \text{inače.} \end{cases}$$

Kako je vremenski trenutak raspada atoma nepredvidiv, duljina vremenskog intervala između raspada dvaju atoma je također nepredvidiva. Ta činjenica, pod pretpostavkom da su vremenski trenutci u kojima bilježimo raspad $t_i \in T \subseteq \mathbb{R}$, implicira da je vjerojatnost $P(S_{2i-1} \leq S_i) = P(S_{2i-1} > S_i)$. Kako su to jedina dva moguća ishoda zaključujemo da slučajna varijabla X_i ima Bernoullijevu distribuciju s parametrom $p = 0.5$ [4]. Ova slučajna varijabla može se transformirati u bilo koju drugu slučajnu varijablu. Neka je Y slučajna varijabla sa diskretnom uniformnom distribucijom na skupu svih brojeva koji imaju konačni d-bitni binarni zapis između 0 i 1. Y se tada može dobiti generirajući niz uzastopnih varijabli X_i, gdje će vrijednost svakog bita biti jedna realizacija varijable X_i.

Često i fizički procesi na samom računalu mogu dovesti do generiranja slučajnih brojeva. Postoji mnogo načina na koji se to može napraviti, a jedan od njih su postigli 1994. godine Davis, Ihaka i Fenstermacher [2] koristeći se nasumičnim kretanjem zraka u hard diskovima kao izvorom slučajnosti. Jedni od poznatijih generatora zaista slučajnih brojeva (eng. *True Random Number Generator, TRNG*) su generator na stranici random.org koja se služi atmosferskim šumovima za generiranje slučajnih brojeva kao i John Walkerov generator koji se temelji na raspadu kriptona 85. Povijesnim podacima o vremenima raspada kao i praćenju uživo može se pristupiti na web stranici http://www.fourmilab.ch/hotbits/.

Problemi generatora ovakvih slučajnih brojeva zasnovanih na mjerenju prirodnih fenomena za koje mislimo da su slučajni su višestruki. Prvi problem je problem samog mjerenja i potrebne opreme. Potrebno je primjerice prispojiti Geigerov brojač na računalo kako bi se utvrdila količina radijacije u prostoriji ili imati dovoljno precizne uređaje koje mogu pratiti raspadanje atoma. Drugi problem koji imaju je nedovoljna brzina, tj. potrebno je dosta vremena da se prikupe svi ti vanjski podaci. Također nizovi brojeva generiranih na ovaj način nisu reproducibilni što može stvoriti problem primjerice prilikom testiranja aplikacija koje koriste ovakve generatore.

Upravo zbog ovih problema razvija se koncept generatora pseudoslučajnih brojeva. Nizovi pseudoslučajnih brojeva izgledaju kao nizovi slučajnih brojeva, zadovoljavaju većinu statističkih svojstava kao nizovi slučajnih brojeva, ali su dobiveni deterministčkim procesom. U idućem poglavlju donosimo dva primitivna, ali i ilustrativna algoritma za generiranje pseudoslučajnih brojeva na kojima ćemo objasniti njihova osnova obilježja te analizirati njihove dobre i loše strane.

3. Primitivni generatori pseudoslučajnih brojeva

Generatori pseudoslučajnih brojeva (engl. *Pseudorandom number generator, PRNG*) su algortimi za generiranje niza brojeva čija svojstva dobro aproksimiraju svojstva niza slučajnih brojeva, ali nisu zaista slučajni već su dobiveni nekim determinističkim postupkom. Deterministički algoritam, f, generira rekurzivno niz brojeva, tako da je vrijednost svakog sljedećeg broja unaprijed određena prethodnim vrijednostima brojeva u nizu (najčešće jednom vrijednošću):

$$x_i = f(x_{i-1}, \ldots, x_{i-k}).$$

Broj prethodno generiranih pseudoslučajnih brojeva koji determiniraju vrijednost sljedećeg pseudoslučajnog broja k naziva se red generatora. Skup prethodno generiranih vrijednosti, najčešće jedna vrijednost, koje determiniraju sljedeći broj u nizu nazivaju se **sjeme** ili **ključ**.

Važno je primijetiti da ukoliko pokrenemo algoritam s istim sjemenom, tada ćemo dobiti i isti rezultat. Što drugim riječima znači da ukoliko se prilikom generiranja niza pseudoslučajnih brojeva ponovi vrijednost sjemena tada će se ponoviti i cijeli dio niza nastao nakon toga sjemena. Duljina niza prije nego što se niz počne ponavljati zove se **period**.

Upravo ovu gore navedenu činjenicu, uz naravno činjenicu da brojevi nastali ovakvim procesom nisu zaista slučajni, označit ćemo kao glavni problem generatora pseudoslučajnih brojeva. Kao prednosti generatora pseudoslučajnih brojeva možemo navesti to što su reproduciblni, efikasni, portabilni te za njihovu upotrebu ne treba koristiti nikakve vanjske uređaje.

3.1. Middle-Square algoritam

Middle-Square algoritam prethodno je predstavio, zatim i objavio u svom radu John von Neumann 1951. [11]. Neka je $n \in \mathbb{N}$ paran. Algoritam je sljedeći:

1. Postavimo najviše n-znamenkasti broj kao sjeme $s_0 = c_0$.

2. Kvadriramo s_0 i provjerimo ima li s_0^2 $2n$ znamenki. Ukoliko nema dopunimo nulama s lijeve strane sve dok se ne dobije $2n$-znamenkasti broj. Taj broj označimo sa s_1.

3. Uzmemo srednjih n znamkenaka broja s_1 (izbacimo prvih $\frac{n}{2}$ znamenki, zadržimo slijedećih n te izbacimo posljednjih $\frac{n}{2}$ znamenki). Taj broj je naš prvi generirani pseudoslučajni broj u nizu. Označimo ga s x_1.

4. Postavimo x_1 kao novo sjeme i ponovimo korake od 1.-3. onoliko puta koliko želimo generirati pseudoslučajnih brojeva. Rezultat i-te, $i \in \{1, \ldots, k\}, k \in \mathbb{N}$, iteracije označimo s x_i.

Nakon provođenja algoritma dobijamo k-dimenzionalan vektor $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \ldots & x_k \end{bmatrix}^T$ čije komponente sačinjavaju naši generirani pseudoslučajni brojevi.

Idući ulomak donosi Middle-square algoritam napisan u R jeziku koji nam je poslužio da testiramo njegove performanse za različite vrijednosti sjemena i različit broj znamenki n.

```
install.packages("TeachingDemos")
library("TeachingDemos")

MidSquareAlgorithm <- function(seed, dig, itr) {
  result <- NULL
  period <- 0
  for(i in 1:itr) {
    value <- seed * seed
    value <- paste(digits(value, 2 * dig), collapse = "")
    value <- substr(value, dig / 2 + 1, 3 * dig / 2)
    value <- as.numeric(value)
    seed <- value
    if (!(value %in% result)) {
      period <- period + 1
    }
    result <- c(result, value)

  }
  return(list(result, period))
}
```

Za pokretanje algoritma osim osnovnih paketa R jezika potrebno je još instalirati paket *"TeachingDemos"* koji sadrži funkciju *digits*. Funkcija *digits* prima dva parametra. Prvi je integer ili vektor integera koji treba rastaviti na znamenke, drugi je broj znamenki koji treba vratiti. Ukoliko broj ima manje znamenki nego što funkcija treba vratiti tada doda onoliko 0 prije prve znamenke broja koliko je potrebno da se dostigne zadana vrijednost parametra.

Sam algoritam je zamišljen kao funkcija koja prima tri integer parametra: *seed* je sjeme, *dig* je broj znamenki, ono što smo u definiciji algoritma označili s n, dok je *itr* broj iteracija, u definiciji označen s k. Funkcija provede zadani algoritam kako je to navedeno u definiciji algoritma te vrati k-dimenzionalni vektor pseudoslučajnih brojeva i period. Podsjetimo se, ukoliko je period manji od k, tada je zasigurno došlo do ponavljanja dijelova niza pseudoslučajnih brojeva što je rezultat koji želimo izbjeći. Primjetimo i to da je maksimalna vrijednost perioda 10^n.

Iduća tablica donosi vrijednosti perioda za razne vrijednosti sjemena. Vrijednosti sjemena u tablici bit će zapisana u terminima broja znamenaka koje smo poslali algoritmu kao parametar. Primjerice ukoliko algoritam pokrećemo sa parametrom $dig = 8$, a vrijednost sjemena je 1234, tada će vrijednost sjemena u tablici biti napisana 00001234.

Iz tablice je jasno vidljivo da se periodi drastično razlikuju po pojedinim vrijednostima sjemena. Odabirom sjemena za koji algoritam ima nizak period algoritam postaje beskoristan. Upravo je to glavni razlog zašto smo ovaj algoritam iskoristili samo u ilustrativne svrhe te se on ne koristi u praksi za generiranje pseudoslučajnih brojeva.

Sjeme	Period
1234	56
001234	67
00001234	3
2500	1
12345678	10451
48205429	1638

Tablica 1: Iznos perioda za različite vrijednosti sjemena

3.2. Linearni kongruentni generator

Linearni kongruentni generator (engl. *Linear congruential generator* ili kraće LCG) predstavio je američki matematičar Derrick Henry Lehmer 1948. godine. Ideja je da generator osim sjemenom bude određen s još tri unaprijed definirana parametra, multiplikatorom $a \in \mathbb{N}$, inkrementom $c \in N_0$ i operatorom cjelobrojnog ostatka dijeljenja - modulom $m \in \mathbb{N}$.

Pretpostavimo da je $a < m$, $c < m$ te neka je x_0, $x_0 \in \mathbb{N}_0$, $x_0 < m$ početna vrijednost. Tada se LCG algoritam može zapisati kao:

$$x_i \equiv (ax_{i-1} + c) \; mod \; m, \quad \text{gdje je } 0 \leq x_i < m. \tag{1}$$

Često u jednadžbi (1) postavimo $c = 0$. Takva klasa LCG generatora koji imaju $c = 0$ zovu se multiplikativni kongruentni generatori. Kada je $c \neq 0$ često govorimo o kombiniranim kongruentnim generatorima. Niz nastao iz rekurzije (1) zovemo Lehmerovim nizom.

Kao i kod Middle-square algoritma i ovdje donosimo kod napisan u R jeziku kojim smo konstruirali linearni kongruentni generator.

```
LCGAlgorithm <- function (seed, a, c, m, itr) {
  result <- NULL
  period <- 0
  for (i in 1:itr) {
    value <- (a * seed + c) %% m
    seed <- value
    if (!(value %in% result)) {
      period <- period + 1
    }
    result <- c(result, value)

  }
  return(list(result, period))
}
```

Iz koda, ali i same rekurzije vidljivo je da period ovisi o tome kako izaberemo parametre a, c i m. Kako je x_i definiran preko x_{i-1}, a postoji ograničen broj vrijednosti koji x_i može

poprimiti $\{0, 1, \ldots, m-1\}$, maksimalni period LCG-a je m. Također, kako je $x_{i-1} \neq 0$ u algoritmu multiplikativnog kongruentnog generatora, njegov maksimalni period je $m-1$.

U doba razvoja računala kada se nije mogao računati tako veliki broj računskih operacija algoritmi koji su za m imali potenciju broja 2 brže su se izvršavali. Maksimalni period takvog algoritma je $m/4$ i može se dobiti sa svakim multiplikatorom koji pri dijeljenju s 8 daje ostatak 3 ili 5 (vidjeti [4] str. 12).

Period multiplikativnog kongurentnog generatora s parametrima a i m ovisi o najmanjem prirodnom broju k za koji vrijedi:

$$a^k \equiv 1 \ mod \ m. \tag{2}$$

Razlog u tome leži što kada je relacija (3) zadovoljena niz se počne ponavljati. Točnije ukoliko je $a^k \equiv 1 \ mod \ m$, tada je idući član niza $1 * a = a \ mod \ m$ što upravo odgovara prvom članu niza. Period stoga ne može biti veći od k.

Po Euler-Fermatovom teoremu koji kaže da ukoliko su dva broja a i m relativno prosta tada vrijedi

$$a^{\phi(m)} \equiv 1 \ mod \ m, \tag{3}$$

gdje je $\phi(m)$ Eulerova funkcija koja broji prirodne brojeve do broja m koji su relativno prosti s m. Podsjetimo se, brojevi a i m su relativno prosti ako je najveći prirodni broj s kojim su djeljivi i a i m jednak 1. Sukladno tome za dani m tražimo a takav da je k u jednadžbi (3) jednak $\phi(m)$. Takav broj a se naziva primitivan korijen modulo m. Ukoliko je m prost broj tada je broj primitivnih korijena modulo m jednak $\phi(m-1)$.

Pogledajmo to na primjeru. Neka je $m = 31$ i $a = 7$, tj. imamo relaciju $x_i \equiv 7x_{i-1} \ mod \ 31$. Za početnu vrijednost smo izabrali $x_0 = 19$. Sljedeći brojevi u nizu su:

$$9, 1, 7, 18, 2, 14, 5, 4, 28, 10, 8, 25, 20, 16, 19$$

nakon čega se niz počinje ponavljati, što znači da je period 15. Zaključujemo kako imamo $7^{15} \equiv 1 \ mod \ 31$, tako da 7 nije primitivan korijen modulo 31.

U drugom primjeru postavit ćemo $m = 31$, $a = 3$ i ponovno počinjemo s $x_0 = 19$. Kada smo definirali početne parametre i vrijednost sjemena možemo pokrenuti algoritam. Dobijemo niz od 30 brojeva prije nego se vratimo na 19 i niz se počne ponavljati. Takav rezultat je očekivan jer vrijedi da je $3^{30} \equiv 1 \ mod \ 31$, a kako je $\phi(31) = 30$ jer je 31 prost broj, zaključujemo da je 3 primitivan korijen modulo 31. Postoji $\phi(30) = 8$ primitivnih korijena modulo 31.

Kroz povijest mnogi matematičari pokušali su naći optimalne parametre kako bi period modela bio što veći, a koreliranost između članova niza što manja. Često se za m uzima Mersennov prost broj $2^{31} - 1$. Podsjetimo se, prost broj je Mersennov prost broj ukoliko je za jedan manji od neke potencije broja 2. Kao česti izbor multiplikatora a za ovaj m uzima se 7^5 (vidjeti [4] str. 13).

Wu (1997.) [19] u svojoj knjizi nudi optimalne multiplikatore za ova dva gore navedena modula. Za $m = 2^{31} - 1$ predlaže $a = 2^{15} - 2^{10}$, dok za $m = 2^{61} - 1$, predlaže $a = 2^{30} - 2^{19}$ te $a = 2^{42} - 2^{31}$.

LCG algoritam koristio se kroz povijest u mnogim programskim jezicima za generiranje pseudoslučajnih brojeva. Tako je multiplikativni kongruentni generator s parametrima $m = 2^{31} - 1$ i $a = 16807$ bio algoritam ugrađene funkcije *minstd_rand0* u C++ programskom jeziku, točnije u verziji C++11. Također upotrebljavan je i u jeziku *Carbon* koji je pandan C jeziku za Macintosh operacijske sustave. Park i Miller 1998. godine sumirali su probleme linearnih kongruentih generatora i predložili minimalni standard *(engl. minimal standard)* za LCG generatore [12]. Minimalnim standardom za LCG generatore smatra se da LGC generator generira pseudoslučajne brojeve "najmanje dobro" kao i gore navedeni algoritam.

Dodatna testiranja bit će napravljena u pretposljednjem dijelu rada kada ćemo se više baviti statističkim testovima koji ocjenjuju kvalitetu generatora pseudoslučajnih brojeva.

4. Mersenne Twister

Mersenne Twister je jedan od najpoznatijih i najkorištenijih generatora pseudoslučajnih brojeva. Implementiran je u mnoga softverska rješenja te je i bazni generator pseudoslučajnih brojeva u mnogim programskim jezicima uključujući i R. Najčešće korišten Mersenne Twister algoritam je onaj kojemu je period jednak Mersennovom prostom broju $2^{19937}-1$. Navedena verzija algoritma, MT19937, koristi 32-bitne računalne riječi. Postoji i 64-bitna verzija, MT19937-64, koja koristi 64-bitne računalne riječi. Algoritam su 1998. godine prvi put predstavili Makoto Matsumoto i Takuji Nishimura [9]. Pod w-bitnom računalnom riječi smatramo vektor od w binarinih brojeva.

Ovaj generator također prolazi mnoge statističke testove namijenjene posebno za testiranje kvalitete generatora pseudoslučajnih brojeva poput *Diehard* testova i većine testova iz *TestU01* paketa [5].

4.1. MT19937 Algoritam

U ovom potpoglavlju uvest ćemo masna slova poput $\mathbf{x}$ za označavanje računalnih riječi, tj. w-dimenzionalnih vektora nad poljem $\mathbb{F}_2 = \{0,1\}$, pri čemu w označava veličinu računalne riječi, obično 32 ili 64. MT algoritam generira niz w-dimenzionalnih vektora koji su ustvari pseudoslučajni brojevi između 0 i 2^w-1. Dijeljenjem s 2^w-1 to postaju realni brojevi $[0,1]$.

Označimo s $\mathbf{x}_k^u$ prvih (gornjih) $w-r$, $r \in \mathbb{N}_0$, $0 \leq r \leq w$, bitova riječi $\mathbf{x}_k$, s $\mathbf{x}_{k+1}^l$ posljednjih (donjih) r bitova riječi $\mathbf{x}_{k+1}^l$, dok s $(\cdot|\cdot)$ označimo običnu konkatenaciju riječi. Prema tome, ukoliko je $\mathbf{x} = (x_w, x_{w-1}, \ldots, x_1)$, tada je $\mathbf{x}^u = (x_w, \ldots, x_{r+1})$ $(w-r)$-dimenzionalni binarni vektor, dok je $\mathbf{x}^l = (x_r, \ldots, x_1)$ r-dimenzionalni binarni vektor. Operator $\oplus$ označava zbrajanje po bitovima modulo 2, odnosno $\oplus$ predstavlja logičku operaciju isključivo ili. Za $P, Q \in \mathbb{F}_2$, definiramo $P \oplus Q$ kao u tablici 2.

P	Q	$P \oplus Q$
0	0	0
0	1	1
1	0	1
1	1	0

Tablica 2: Logička operacija isključivo ili

Algoritam tada zadajemo preko sljedeće rekurzije:

$$\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l) A \tag{4}$$

pri čemu $n \in \mathbb{N}$ označava stupanj rekurzije, r označava mjesto razdvajanja riječi, za $m \in \mathbb{N}$ vrijedi $1 \leq m \leq n$, dok je A $w \times w$ dimenzionalna matrica s elementima iz polja $\mathbb{F}_2$.

Zadavanjem sjemena $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ za $k = 1$ algoritam generira $\mathbf{x}_{n+1}$. Povećavanjem k na $2, 3, \ldots$ dobivamo $\mathbf{x}_{n+2}, \mathbf{x}_{n+3}, \ldots$. Matricu A odabiremo tako da se množenje s njom može

učiniti vrlo brzo u računalnom smislu. Prikladan kandidat za to je matrica

$$A = \begin{bmatrix} & 1 & & \\ & & 1 & \\ & & & 1 \\ a_w & a_{w-1} & \dots & a_1 \end{bmatrix}$$

jer se tada $\mathbf{x}A$ može izračunati koristeći samo bitovne operacije. Slijedi:

$$\mathbf{x}A = \begin{cases} shiftright(\mathbf{x}), & \text{za } LSB(\mathbf{x}) = 0 \\ shiftright(\mathbf{x}) \oplus \mathbf{a}, & \text{za } LSB(\mathbf{x}) = 1 \end{cases}$$

gdje je $LSB(\mathbf{x})$ najmanje signifikantni bit od $\mathbf{x}$, $\mathbf{a} = (a_w, \dots, a_1)$ **bitovna maska**, $shiftright(\mathbf{x})$ operator pomaka udesno. Točnije ukoliko je $\mathbf{x} = (x_w, x_{w-1}, \dots, x_1)$, tada je $shiftright(\mathbf{x}) = (0, x_w, \dots, x_2)$. Analogno se definira i operator pomaka ulijevo $shiftleft(\mathbf{x}) = (x_{w-1}, \dots, x_1, 0)$.

Radi postizanja boljih svojstava generatora, svaku generiranu riječ množimo s $w \times w$ invertibilnom matricom T, koji autori algoritma nazivaju *"Tempering"* matrica [9]. Transformacije koje koristimo da bismo dobili $\mathbf{z} = \mathbf{x}T$ su sljedeće:

$$\begin{aligned} \mathbf{y} &:= \mathbf{x} \oplus (\mathbf{x} \gg u) \\ \mathbf{y} &:= \mathbf{y} \oplus ((\mathbf{y} \ll s) \wedge \mathbf{b}) \\ \mathbf{y} &:= \mathbf{y} \oplus ((\mathbf{y} \ll t) \wedge \mathbf{c}) \\ \mathbf{z} &:= \mathbf{y} \oplus (\mathbf{y} \gg l) \end{aligned}$$

gdje su $u, s, t, l \in \mathbb{N}_0$, $\mathbf{b}, \mathbf{c}$ bitovne maske duljine w, dok $\mathbf{x} \gg u$ označava pomak udesno u bitova. Pomak udesno u bitova definira se kao $shiftright^u(\mathbf{x})$ pri čemu je $shiftright^u(\mathbf{x}) = shiftright(shiftright^{u-1}(\mathbf{x}))$. Analogno se definira i pomak t koraka u lijevo $\mathbf{x} \ll t$ kao $shiftleft^t = shiftleft(shiftleft^{t-1}(\mathbf{x}))$.

Za postizanje rekurzije (4) dovoljno je uzeti vektor od n riječi. Neka su $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ nenegativni cijeli brojevi veličine računalne riječi, obično 32 ili 64, i cjelobrojna varijabla, te $\mathbf{u}, \mathbf{l}$ i $\mathbf{a}$ nenegativne cjelobrojne konstatne također veličine računalne riječi. Algoritam je sljedeći:

1. Definiramo $\mathbf{u} = 1 \dots 10 \dots 0$ kao bitovnu masku za gornjih $w-r$ bitova, tj. jest vodećih $w - r$ bitova je jednako jedan, dok ostalih r bitova je jednako nuli.
   Slično, definiramo $\mathbf{l} = 0 \dots 01 \dots 1$ kao bitovnu masku za donjih r bitova ($\mathbf{l} = \neg\mathbf{u}$). Prvih $w - r$ bitova jednako je nula, dok je posljednjih r bitova jednako jedan .
   Također definiramo i $\mathbf{a} = (a_w, \dots, a_1)$.

2. Postavimo $i = 1$ te $\mathbf{x}_1, \mathbf{x}_2, \dots \mathbf{x}_n$ bilo koje nenul početne vrijednosti.

3. Računamo konkatenaciju $\mathbf{y} = (\mathbf{x}_i^u | \mathbf{x}_{i+1}^l)$ kao: $\mathbf{y} = (\mathbf{x}_i \wedge \mathbf{u}) \vee (\mathbf{x}_{i+1} \wedge \mathbf{l})$.

4. Računamo $\mathbf{x}_{n+i} = \mathbf{x}_{i+m} \oplus (\mathbf{x}_i^u | \mathbf{x}_{i+1}^l)A$ kao:
   $$\mathbf{x}_{n+i} - \begin{cases} \mathbf{x}_{i+m} \oplus (y \gg 1), & \text{gdje je zadnji signifikantni bit od } y = 0 \\ \mathbf{x}_{i+m} \oplus (y \gg 1) \oplus \mathbf{a}, & \text{gdje je zadnji signifikantni bit od } y = 1. \end{cases}$$

5. Računamo $\mathbf{z} = \mathbf{x}_{n+i}T$ kao:
   $\mathbf{z} \leftarrow \mathbf{x}_{n+i}$
   $\mathbf{z} \leftarrow \mathbf{z} \oplus (\mathbf{z} \gg u)$
   $\mathbf{z} \leftarrow \mathbf{z} \oplus ((\mathbf{z} \ll s) \wedge \mathbf{b})$
   $\mathbf{z} \leftarrow \mathbf{z} \oplus ((\mathbf{z} \ll t) \wedge \mathbf{c})$
   $\mathbf{z} \leftarrow \mathbf{z} \oplus (\mathbf{z} \gg l)$
   rezultat je $\mathbf{z}$.

6. Postavljamo $i = i + 1$.

7. Vrati se na korak 3.

Izbor parametara koji daje MT19937 algoritam je sljedeći:

- $(w, n, m, r) = (32, 624, 397, 31)$
- $a = 9908B0DF_{16}$
- $u = 11$
- $s = 7, b = 9D2C5680_{16}$
- $t = 15, c = EFC60000_{16}$
- $l = 18$.

Prije nego što pokažemo kako izgleda MT19937 algoritam u R jeziku, pozabavimo se pitanjem sjemena, tj. inicijalizacije algoritma. Iz rekurzije (4) vidljivo je da algoritam zahtjeva niz od n w-bitnih računalnih riječi kao sjeme. Ideja je da se niz od n riječi ne generira na način da zadamo svaku riječ, već samo prvi element u nizu, a potom se na osnovu te prve vrijednosti, izgeneriraju i ostalih $n - 1$ vrijednosti sjemena.

Inicijalizacija se odvija na sljedeći način. Zadamo w-bitnu računalnu riječ $\mathbf{x}_0$. Sljedeće vrijednosti u nizu se generiraju formulom:

$$\mathbf{x}_i = f \cdot (\mathbf{x}_{i-1} \oplus (\mathbf{x}_{i-1} \gg (w-2)) + i.$$

Ukoliko je $\mathbf{x}_i > 2^{32} - 1$, za $i \in 1, 2, \ldots, n$ tj. ne može se zapisati u 32-bitnom binarnom zapisu, na njega djelujemo operacijom *mod* 2^{32} koja daje cjelobrojni ostatak pri dijeljenju broja $\mathbf{x}_i$ brojem 2^{32}. Za algoritam MT19937 vrijednost konstante f je 1812433253.

4.2. Implementacija MT19937 u R jeziku

```
install.packages("binaryLogic")
install.packages("bitops")

library("binaryLogic")
library("bitops")
```

```
w <- 32
n <- 624
m <- 397
r <- 31
a <- 2567483615  # 9908b0df -> 2567483615
b <- 2636928640  # 9d2c5680 -> 2636928640
c <- 4022730752  # efc60000 -> 4022730752
u <- 11
s <- 7
t <- 15
l <- 18
f <- 1812433253

uu <- c(rep(TRUE, w-r), rep(FALSE, r))
ll <- c(rep(FALSE, w-r), rep(TRUE, r))
unum <- as.numeric(as.binary(uu, logic = TRUE))
lnum <- as.numeric(as.binary(ll, logic = TRUE))

SetSeed <- function(seed) {
  x <- c()
  x[1] <- seed
  for (i in 2:n) {
    x[i] <- (f * bitXor(x[i - 1], bitShiftR(x[i - 1], w - 2)) + (i - 1)) %% 2^32
    i <- i + 1
  }
  return(x)
}

MersenneTwister19937 <- function(x,len) {
  x <- SetSeed(x)
  result <- NULL
  period <- 0
  for (i in 1:len) {
    y <- bitOr(bitAnd(x[i], unum), bitAnd(x[i + 1], lnum))
    if (y %% 2 == 0) {
      z <- bitXor(x[i + m], bitShiftR(y, 1))
    } else {
      z <- bitXor(bitXor(x[i + m], bitShiftR(y, 1)), a)
    }
```

```
    z <- bitXor(z, bitShiftR(z, u))
    z <- bitXor(z, bitAnd(bitShiftL(z, s), b))
    z <- bitXor(z, bitAnd(bitShiftL(z, t), c))
    z <- bitXor(z, bitShiftR(z, l))

    if (!(z %in% x[(n + 1):length(x)])) {
      period <- period + 1
    }

    x[n + i] <- z
    i <- i + 1
  }
  result <- x[(n + 1):length(x)]
  return(list(result, period))
}
```

Kako radimo s w-bitnim računalnim riječima koji su zapravo zapis cijelih brojeva iz intervala $[0, 2^{32} - 1]$ u binarnom sustavu, što znači da prvi element u binarnom zapisu ne označava predznak broja potrebno je bilo instalirati i učitati pakete *binaryLogic* i *bitops* koji služe za lakše konvertiranje upravo takvih tipova binarnih podataka u cjelobrojni tip podatka i obratno. Nakon učitavanja navedenih paketa inicijalizirali smo parametre modela zadane u prethodnom poglavlju. Sve vrijednosti smo konvertirali u cjelobrojni tip podatka.

Funkcija *SetSeed* je pomoćna funkcija koja služi za generiranje sjemena iz početne vrijednosti sjemena. Kao parametre prima jedan cjelobrojni broj iz intervala $[0, 2^{32} - 1]$ te generira vektor od 624 cijela broja iz tog intervala. Svaki broj kada se pretvori u binarni zapis tada se može zapisati kao 32-bitna računalna riječ.

MersenneTwister19937 je funkcija koja prima dva parametra, x je početna vrijednost sjemena, dok je *len* broj koliko pseudoslučajnih brojeva želimo izgenerirati. Funkcija vraća *len*-dimenzionalni vektor kojemu je svaka komponenta jedan pseudoslučajni broj.

Na taj način izgenerirali smo pseudoslučajne cijele brojeve iz intervala $[0, 2^{32} - 1]$.

5. Testiranje kvalitete generatora pseudoslučajnih brojeva

Potreba za testiranjem kvalitete generatora pseudoslučajnih brojeva postoji otkad postoje generatori. Kao glavni kriterij kvalitete generatora slučajnih brojeva uzimamo taj da izlaz generatora (niz pseudoslučajnih brojeva) dobro imitira niz nezavisnih i uniformno distribuiranih slučajnih varijabli, tj. svojstva tog niza. Kao što je već ranije navedeno u radu, svi generatori pseudoslučajnih brojeva su deterministički i kao takvi jasno je da ne mogu proći sve testove vezane uz uniformnost i nezavisnost.

Na trenutak napravimo malu digresiju i podsjetimo se da interval $(0, 1)$ u računalnom zapisu nije neprekidan, već diskretan. Razlog tome je vrlo jednostavan. Svako računalo ima određenu preciznost do koje može bilježiti vrijednosti brojeva. Pretpostavimo da možemo bilježiti vrijednosti brojeva sa w-bitnom preciznošću. Sada uzmimo dva broja u binarnom zapisu koji imaju vrijednost svih bitova do posljednjeg (w-tog) jednaku. S a označimo broj kojemu je vrijednost posljednjeg bita 0, dok s b označimo broj kojemu su sve vrijednosti do w-tog bita jednake broju a, međutim vrijednost posljednjeg bita je 1. Tada je jasno da između a i b ne postoji niti jedan broj u računalnom smislu. Razliku između a i b ćemo označiti s δ.

Ta činjenica povlači idući zaključak. Iako je u matematičkom smislu $U[0, 1]$ odgovara $U(0, 1)$ jer je interval neprekidan pa je vjerojatnost da ćemo odabrati točno određeni broj iz nekog intervala jednaka 0, ovdje to nije slučaj jer radimo na diskretnom skupu.

Pretpostavimo da imamo generator pseudoslučajnih brojeva koji generira cijele nenegativne pseudoslučajne x_i brojeve na skupu $\{0, 1, \ldots, n\}$. Tada su $u_i = x_i/n$ pseudoslučajni brojevi iz intervala $[0, 1]$. Problem granica, tj. brojeva 0 i 1 koji ovakvim pristupom nikad ne bi trebali biti generirani je karakterističan za svaki algoritam ili klasu algoritama te ga ovdje nećemo posebno obrađivati.

Pojam **granularnosti** se odnosi na to kolika je minimalna razlika između dva pseudoslučajna broja koja se mogu generirati. Pretpostavimo da imamo generator koji može genenirati brojeve iz skupa $\{0, 1, \ldots, n\}$. Ako podijelimo svaki generirani pseudoslučajni broj s maksimalnim, u ovom slučaju n, onda dobivamo minimalnu razliku među dva generirana pseudoslučajna broja, $1/n$. Jasno je da tu minimalnu razliku želimo držati što manjom i približiti je što više δ.

5.1. Statističko testiranje

Pri testiranju generatora pseudoslučajnih brojeva nul-hipoteza je da je izlaz generatora, tj. niz uzastopnih izlaznih varijabli generatora $x_1, x_2, \ldots, x_n$ realizacija niza nezavisnih slučajnih varijabli s distribucijom $U(0, 1)$ ako se radi o brojevima (u računalnom slučaju diskretna uniformna $U(0, 1)$) odnosno Bernoullijevom distribucijom s parametrom $p = 0.5$ ukoliko se radi o bitovima. Alternativna hipoteza je da niz nema tu distribuciju.

Inače kod statističkih testova hipoteze postavimo tako da ono što želimo opovrgnuti stavimo u nul-hipotezu, tada nakon provođenja testa na određenoj razini značajnosti za-

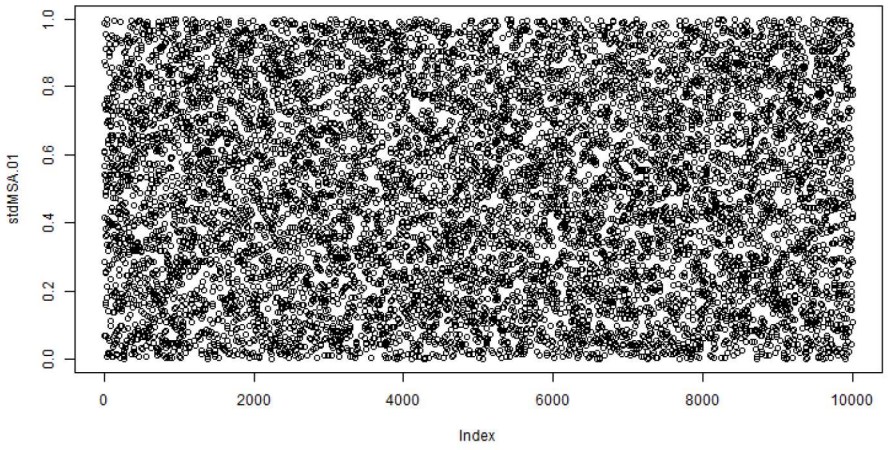
ključujemo odbacujemo li nul-hipotezu ili ne. Pozitivan rezultat, u smislu donošenja zaključka, je tada odbaciti nul-hipotezu. Odluku donosimo tako da p-vrijednost testa usporedimo s unaprijed fiksiranom razinom značajnosti α (najčešće 0.05). Problem koji ovdje imamo s takvim testiranjem kvalitete generatora pseudoslučajnih brojeva jest taj što ovdje priželjkujemo negativan rezultat, tj. ne želimo odbaciti nul-hipotezu. Svejedno, višestruko provođenje takvih testova može biti vrlo informativno jer će lošiji generatori pasti na testovima puno češće od onih dobrih.

Za početak ćemo izraditi više nizova pseudoslučajnih brojeva generiranih algoritmima predstavljenim u ovom radu kao i standardnim R funkcijama. Generirali smo dva niza pseudoslučajnih brojeva *Middle Square* algoritmom kako bismo vidjeli ovisnost tog algoritma o vrijednosti sjemena. Upotrijebili smo i dva LCG algoritma za generiranje pseudoslučajnih brojeva kako bismo vidjeli koliko se razlikuju dva algoritma iste klase za drugačije vrijednosti parametara, dok nam pseudoslučajni brojevi generirani funkcijama *runif()* služe kao kontrolni algoritmi za usporedbu s algoritmima koje smo programirali. Funkcija *set.seed()* fiksira rezultate *runif()* funkcije kako bismo mogli ponovno reporoducirati iste rezultate. Parametar *kind* specificira koji ćemo algoritam koristiti za generiranje pseudoslučajnih brojava. Ukoliko se parametar ne definira tada se koristi *Mersenne Twister* algoritam. Za generiranje niza *runif.01* nismo specificirali koji ćemo algoritam algoritam iskoristiti pa se u tom slučaju koristi *Mersenne Twister* algoritam. Naredbom *set.seed(320, kind = "Marsaglia-Multicarry")* smo definirali da će korišteni pozadinski algoritam za generiranje niza *runif.02* biti *Marsaglia-Multicarry (multiply with carry)* algoritam [6]. U slučaju da nije drugačije naznačeno funkcijom *runif()* generirat ćemo brojeve *Mersenne Twister* algoritmom. Treba napomenuti da se MT algoritam koji koristi *runif()* funkcija te MT algoritam kojeg smo u ovom radu programirali ne razlikuju u samom algoritmu već u načinu na koji se inicijalizira sjeme. Zbog toga će se pseudoslučajni brojevi koje ti algoritmi izgeneriraju razlikovati. Inicijalizaciju sjemena koji se koristi u integriranoj R funkciji definirao je Brian D. Ripley 2002. godine [13].

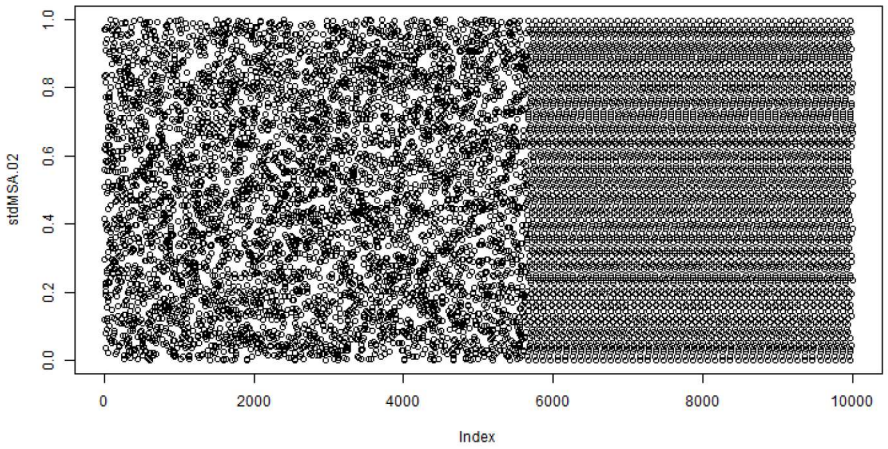
```
MSA.01 <- MidSquareAlgorithm(12345678, 8, 10000)[[1]]
stdMSA.01 <- MSA.01 / 10^8

MSA.02 <- MidSquareAlgorithm(54065874, 8, 10000)[[1]]
stdMSA.02 <- MSA.02 / 10^8

LCG.01 <- LCGAlgorithm(1234, 16807, 0, 2^31-1, 100000)[[1]]
stdLCG.01 <- LCG.01 / 2^31-1

LCG.02 <- LCGAlgorithm(134, 151, 1, 2^15, 10000)[[1]]
stdLCG.02 <- LCG.02 / 2^15

MT19937.01 <- MersenneTwister19937(12345, 10000)[[1]]
```

```
stdMT19937.01 <- MT19937.01 / 2^32

set.seed(15)
runif.01 <- runif(10000)

set.seed(320, kind = "Marsaglia-Multicarry")
runif.02 <- runif(10000)
```

Prisjetimo se da algoritmi prikazani u ovom radu generiraju nenegativne pozitivne cijele brojeve. Kako bismo ih mogli prikazati na istoj skali podijelili smo svaki član niza pseudoslučajnih brojeva s najvećim brojem koji algoritam može generirati kako je i prikazano u kodu. Svakim od algoritama izgenerirano je deset tisuća pseudoslučajnih brojeva. Prije nego pređemo na statističke testove naredbom *plot()* nizove ćemo prikazati grafički. Na x-osi je redni broj svakog od elemenata niza, dok je na y-osi prikazana njegova vrijednost.



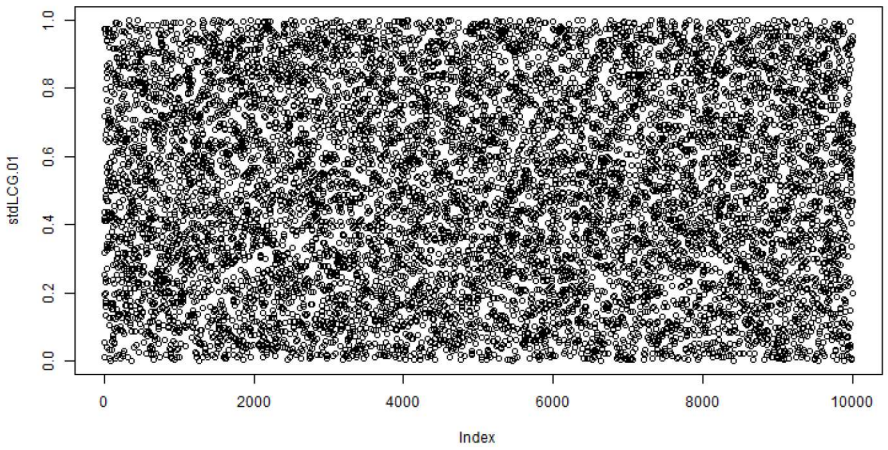
Slika 1: Grafički prikaz niza pseudoslučajnih brojeva *stdMSA.01*



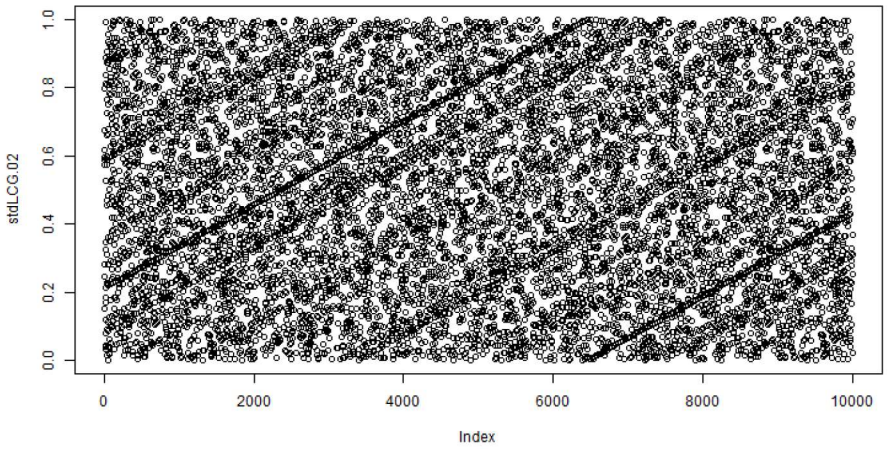
Slika 2: Grafički prikaz niza pseudoslučajnih brojeva *stdMSA.02*

Usporedivši grafove vidimo da na slici 1 uzorak naočigled izgleda slučajno, dok na slici 2 izgleda slučajno do jedne granice zatim se uzorak počinje ponavljati. Od dobrih generatora očekujemo da generiraju brojeve koje dobro oponašaju realizacije jednostavnog slučajnog uzorka. Niz *stdMSA.02* ima period 5735 i to je granica do koje grafički prikaz niza izgleda slučajno. Poslije toga člana generator ulazi u petlju gdje se uzastopce ponavlja sljedećih 100 članova niza.

Na sljedeća dva prikaza vidjet ćemo kako uzorak može postojati i prije ponavljanja elemenata niza.



Slika 3: Grafički prikaz niza pseudoslučajnih brojeva *stdLCG.01*



Slika 4: Grafički prikaz niza pseudoslučajnih brojeva *stdLCG.02*

Niz *srdLCG.02* ima period 8192 te se nakon toga člana niz počinje vrtjeti u krug što implicira da će se grafički prikaz poslije tog člana krenuti ponavljati. No primjetimo da na slici 4 i prije ponavljanja vidimo da postoje područja koja su gušća te područja koja su rijeđa kao i to da gušća područja izgledaju kao da prate nekakve zamišljene pravce. Intuitivno nam je jasno da tako nešto ne može biti slučajno te generatore (algoritme) koji su izgenerirali nizove *stdLCG.02* i *stdMSA.02* nećemo smatrati relevantima već ih koristimo u ilustrativne svrhe. Kako odabir vrijednosti sjemena ne ovisi o kreatoru generatora već korisniku, dobar generator mora generirati odgovarajuće nizove za bilo koju dozvoljenu vrijednost sjemena.

5.2. Testovi uniformnosti

5.2.1. KS test

Nakon što smo proveli tzv. *eye test*, test pogledom, prvi statistički test koji ćemo provesti jest Kolmogorov-Smirnov, odnosno KS test kojim ćemo testirati pripadnost uniformnoj $U(0,1)$ distribuciji. U pravilu samo najlošiji generatori padaju na ovakvim testovima globalne uniformnosti. Iako se radi o deterministčkim algoritmima u ovom poglavlju prema nizovima pseudoslučajnih brojeva $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ odnosit ćemo se kao prema realizacijama slučajnih varijabli. Sve testove provodit ćemo na razini značajnosti 0.05.

Hipoteze testa su:

$$H_0 : \mathbf{x} \text{ dolazi iz U(0,1) distribucije}$$
$$H_1 : \mathbf{x} \text{ ne dolazi iz U(0,1) distribucije.}$$

Test statistika KS testa je

$$D_n = \sup_x |F_n(x) - F(x)|,$$

gdje je

$$F_n(x) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{(-\infty, x]}(x_i)$$

emiprijska funkcija distribucije niza $\mathbf{x} = x_1, x_2, \ldots, x_n$. U uvjetima istinitosti nul-hipoteze test statistika ima distribuciju

$$F(x) = \frac{\sqrt{2\pi}}{x} \sum_{k=1}^{\infty} \exp^{-(2k-1)^2\pi^2/(8x^2)},$$

što zapravo odgovara vjerojatnosti $P(K \leq x)$, gdje je $K = \sup_{t\in[0,1]} |B(t)|$, a $B(t)$ Brownov most [8]. Distribucija slučajne varijable K često se naziva i Kolmogorovljevom distibucijom. P-vrijednost možemo definirati kao vjerojatnost da test statistika bude veća ili jednaka od dobivene realizacije test statistike u uvjetima istinitosti nulhipoteze.
Iduća tablica nam donosi prikaz nekih numeričkih karakteristika i p-vrijednosti provedenih KS testova. Testove smo proveli u programskom jeziku R funkcijom *ks.test* [14].

Niz	$\bar{x}_n$	s^2	p-vrijednost
stdMSA.01	0.5000878	0.08374917	0.9794
stdMSA.02	0.4954894	0.08390785	0.1294
LCG.01	0.5050986	0.08235768	0.09461
LCG.02	0.4988756	0.08308755	0.9993
stdMT19937.01	0.4994664	0.08230072	0.6229
runif.01	0.4998352	0.08377019	0.8556
runif.02	0.5023268	0.08366743	0.8495

Tablica 3: Vrijednosti prosjeka, korigirane uzoračke varijance i p-vrijednosti KS testova za generirane nizove pseudoslučajnih brojeva

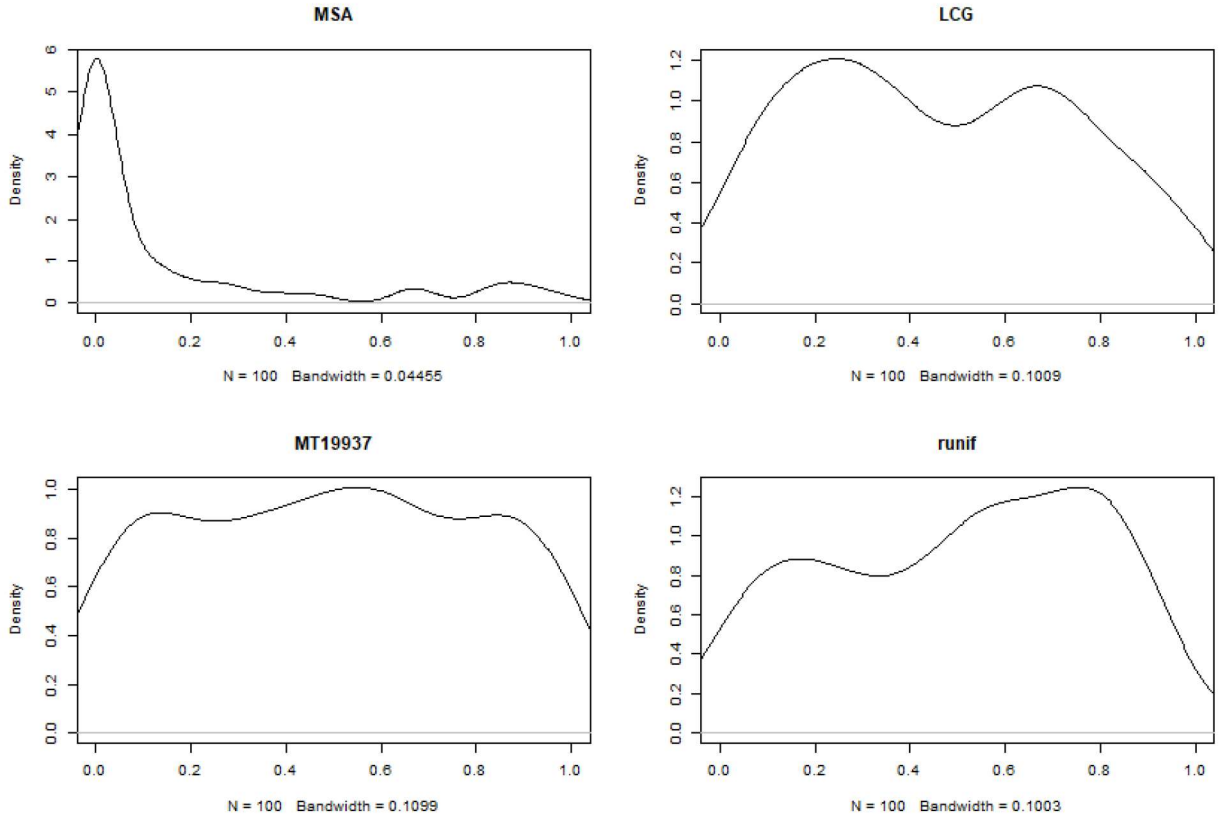
Sve vrijednost su veće od 0.05 prema tome ne možemo na razini značajnosti 0.05 odbaciti nulhipotezu i zaključiti kako uzorak ne pripada $U(0, 1)$ distribuciji. To naravno ne znači ni da prihvaćamo nulhipotezu već jednostavno nemamo dovoljno dokaza da je odbacimo.

Ovdje su testovi, izuzev Middle-square algoritma, provedeni na samo jednom uzorku svakog od algoritama. Važno je napraviti razliku između klase algoritama i algoritama. Primjerice svi LCG generatori bi bili klasa algoritama, dok bi LCG generator sa specificiranim parametrima a, c, m bio točno određen algoritam.

Kako smo rekli da svaki generator mora dobro raditi za svaku dopuštenu vrijednost sjemena ima smisla testirati više uzoraka dobivenih pojedinim algoritmom. Također, ima smisla promatrati dobivene p-vrijednosti takvih testova. Ako p-vrijednosti zamislimo kao slučajne varijable tada bi u uvjetima istinitosti nulhipoteze, p-vrijednosti bile distribuirane uniformno s distribucijom $U(0, 1)$, dok bi u slučaju da je alternativna hipoteza istinita distribucija bila pomaknuta prema nuli [10]. Gentle [4] to u svojoj knjizi navodi kao test drugog reda. Testirali smo algoritme za 100 pseudoslučajno izabranih sjemena iz dopuštenih vrijednosti za svaki algoritam. Sjemena smo izgenerirali funkcijom *sample.int()*. Testirajući Middle-square algoritam od 100 dobivenih p-vrijednosti samo su njih 37 veće od 0.05. Što možemo tumačiti na način da bismo u 63 od 100 provedenih testova zaključili kako na razini značajnosti 0.05 odbacujemo nul-hipotezu te zaključujemo kako $\mathbf{x}$ ne dolazi iz uniformne distribucije. Ponavljajući ovaj postupak za LCG algoritam koji smatramo minimalnim standardom za

generatore pseudoslučajnih brojeva dobivamo 95 p-vrijednosti koje su veće od 0.05, te njih 5 kojih su manje. KS testom smo testirali dolaze li slučajne varijable čijim realizacijama smatramo p-vrijednosti iz $U(0,1)$ distribucije. Kod MSA algoritma p-vrijednost takvog testa je manja od $2.2 \cdot 10^{-16}$, dok je kod LCG algoritma p-vrijednost 0.396. To tumačimo na način da na razini značajnosti 0.05 odbacujemo nulhipotezu te tvrdimo da p-vrijednosti u prvom slučaju (ukoliko ih zamislimo kao varijable) ne dolaze iz $U(0,1)$ distribucije. U drugom slučaju nemamo dovoljno dokaza da donesemo takav zaključak. Analogno, proveli smo isti postupak za MT19937 algoritam, kao i za funkciju *runif()* prethodno definiranu naredbom *set.seed(320, kind = "Marsaglia-Multicarry")*. P-vrijednosti provedenih testova drugog su 0.6219 za Mersenne Twister te 0.5872 za *runif()* algoritam.

Za kraj ovog potpoglavlja donosimo grafički prikaz procijenjene funkcije gustoće p-vrijednosti nastalih testiranjem gore navedenih algoritama.



Slika 5: Procjenjene funkcije gustoće p-vrijednosti KS testova

5.2.2. χ^2 test prilagodbe

χ^2 test prilagodbe (eng. *Chi-square goodness of fit*) je statistički test koji ispituje pripadnost nekoj distribuciji. Test se provodi na način da se za svaki događaj usporedi stvaran i očekivan broj realizacija i na temelju odstupanja u stvarnom i očekivanom broju realizacija donese zaključak o pripadnosti toj distribuciji.

U našem slučaju promatrat ćemo pripadaju li izlazne varijable generatora uniformnoj

$U(0,1)$ distribuciji. Kako je iz opisa testa jasno da se ovdje može testirati samo pripadnost nekoj diskretnoj distribuciji, diskretizirat ćemo interval $(0,1)$ na k disjunktih intervala jednake duljine $(0, \frac{1}{k}) \cup [\frac{1}{k}, \frac{2}{k}) \cup \ldots \cup [\frac{k-1}{k}, 1)$. Sada nam je jasno da na velikom uzorku brojeva očekujemo da u svakom od tih intervala nalazi otprilike $\frac{1}{k}$ od ukupnog broja generiranih brojeva. Test statistiku u ovom slučaju definiramo kao

$$H = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i},$$

gdje o_i označava stvarni broj generiranih brojeva u i-tom intervalu, dok e_i označava očekivani broj elemenata u i-tom intervalu što bi ovdje bilo $\frac{1}{k}$.

U uvjetima istinitosti nul-hipoteze test statistika ima χ^2 distribuciju s $k-1$ stupnja slobode.

Ovdje je također provedeno testiranje na 100 uzoraka četiri algoritma. Middle-square algoritam s parametrom $dig = 8$, multiplikativni linearni kongruentni generator s parametrima $a = 16807$ i $m = 2^{32} - 1$, MT19937 algoritam te već integrirana R funkcija *runif()*, prije koje je pokrenuta naredba *set.seed(420, kind = "Marsaglia-Multicarry")* radi reproducibilnosti. Za broj intervala k uzeli smo 100. Rezultate testiranja prikazat ćemo u idućoj tablici.

Algoritam	$p < 0.05$
MSA	87
LCG	7
MT19937	5
runif	3

Tablica 4: Rezultati testiranja χ^2 testova prilagodbe

Vidimo da je od 100 provedenih testova nad MSA algoritmom 87 puta odbačena nul-hipoteza i prihvaćena alternativna. Po definiciji p-vrijednosti je jasno da će u uvjetima istinitosti nul-hipoteze p-vrijednost biti manja od 0.05 u 5% slučajeva. Kako je očekivani broj odbacivanja 5, a ovdje imamo 87 dodatno potvrđujemo zaključak o lošoj kvaliteti MSA algoritma.

5.3. Problem nezavisnosti

Kako smo već spomenuli u ovom radu od generiranih pseudoslučajnih brojeva očekujemo da se ponašaju kao realizacija niza nezavisnih i jednakodistribuiranih slučajnih varijabli. U protekla dva potpoglavlja bavili smo se pitanjem distribucije, dok ćemo u ovom poglavlju staviti naglasak na problem nezavisnosti. Naravno, jasno je da je svaka sljedeća realizacija jedinstveno određena prethodnom realizacijom i samim algoritmom, prema tome ne možemo zaista očekivati da su varijable čije su realizacije upravo naši pseudoslučajni brojevi nezavisne.

Problem nezavisnosti promatrat ćemo kroz problem korelacije. Znamo da nekoreliranost općenito ne povlači nezavisnost, no nekoreliranost je nužan uvjet za nezavisnost. Prema

tome koreliranost između dviju varijabli će nam implicirati da postoji zavisnost. Ukoliko generirane pseudoslučajne brojeve promotrimo kao realizaciju vremenskog niza tada možemo govoriti o autokorelaciji.

Prije nego što pređemo na testne primjere definirat ćemo pojmove koje ćemo koristiti u ovom poglavlju.

Definicija 5.1. *Neka su X i Y slučajne varijable s konačnim drugim momentima. Kovarijanca $Cov(X,Y)$ definira se kao:*

$$Cov(X,Y) = E[(X - EX)(Y - EY)].$$

Definicija 5.2. *Neka su X i Y slučajne varijable s konačnim drugim momentima. Korelacija ρ_{XY} definira se kao:*

$$\rho_{XY} = \frac{Cov(X,Y)}{Var(X)Var(Y)}.$$

Broj ρ_{XY} još se naziva i koeficijent korelacije.

Definicija 5.3. *Neka je $Y_t,\ t \in 0, 1, \ldots, N$ slučajni proces. Funkcija autokovarijanci procesa definira se kao:*

$$\gamma_Y(t,s) = Cov(Y_t, Y_s).$$

Ukoliko je proces stacionaran tada je $\gamma_Y(t,s)$ funkcija jedne varijable i za $k = t - s$ pišemo $\gamma_Y(k) = \gamma_Y(k,0)$.

Definicija 5.4. *Neka je $Y_t,\ t \in 0, 1, \ldots, N$ slučajni proces. Funkcija autokorelacija procesa definira se kao:*

$$\rho_Y(t,s) = \frac{Cov(Y_t, Y_s)}{Var(Y_t)Var(Y_s)}.$$

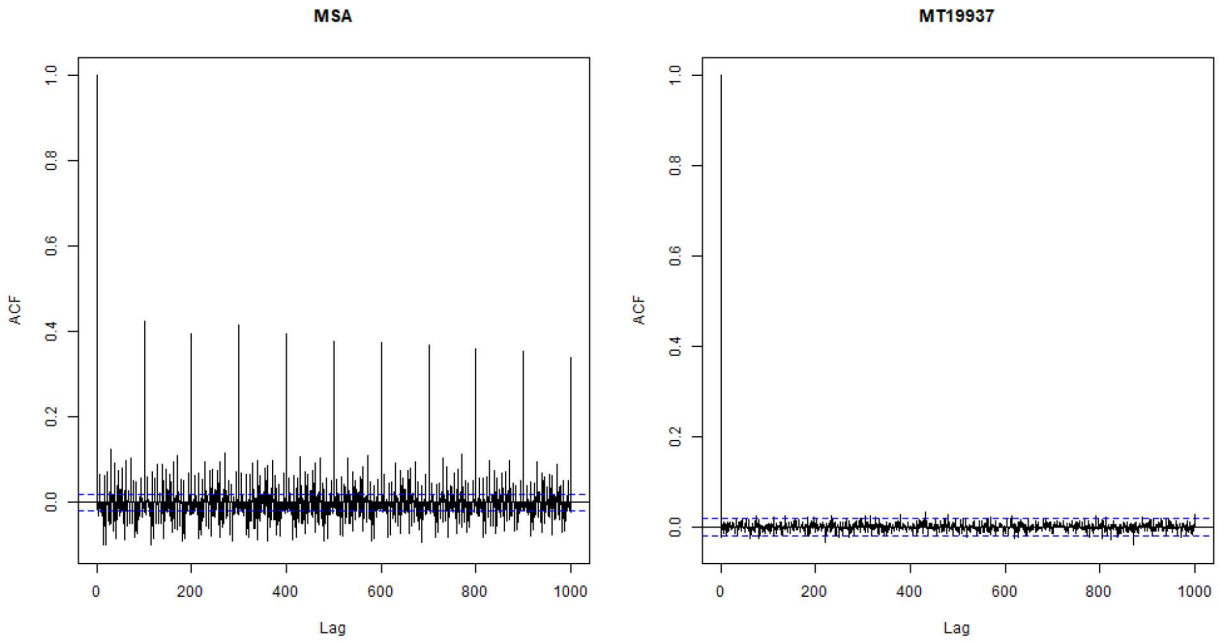
Definicija 5.5. *Za niz podataka $y_1, \ldots, y_n$ funkciju autokovarijance možemo procijeniti izrazom*

$$\hat{\gamma}(k) = \frac{1}{N}\sum_{i=1}^{N-k}(y_i - \overline{y})(y_{i+k} - \overline{y}),$$

gdje je $\overline{y}$ prosjek vrijednosti realizacija y_t. Analogno funkciju autokorelacije možemo procijeniti izrazom

$$\hat{\rho} = \frac{\hat{\gamma(k)}}{\hat{\gamma(0)}}.$$

U pravilu, za većinu generatora autokorelacija opada prema nuli, na malom broju koraka te ponovno poraste nakon velikog broja koraka. Također, ako je broj koraka jednak periodu tada korelacija iznosi 1. Promotrit ćemo procjenu funkcije autokorelacije na dva uzorka dobivena iz algoritama navedenih u prethodnom poglavlju.



Slika 6: Uzoračke funkcije autokorelacije nizova dobivenih MSA i MT19937 algoritmima

Sudeći po prikazu vidimo da MSA algoritam s parametrom $dig = 8$ i sjemenom $seed = 54065874$ ima vrijednost autokorelacije na koracima 100, 200, 300, ... približno 0.4 što je poprilično visoka korelacija. Također korelacija je statistički značajna i na velikoj većini ostalih koraka. S druge strane desni prikaz nam donosi uzoračku funkciju autokorelacije za MT19937 algoritam s pokrenut sjemenom $seed = 12345$. Primjećujemo da su vrijednosti autokorelacije u najvećoj mjeri unutar granica u kojima ih ne smatramo statistički značajnima.

Uz grafičke prikaze autokorelaciju smo testirali i Ljung-Box testom [14]. Ljung-Box test možemo definirati kao:

$$H_0 : \text{Sve korelacije do koraka } h \text{ jednake su nuli.}$$
$$H_1 : \text{Barem jedna korelacija do koraka } h \text{ nije jednaka nuli.}$$

Test statistiku definiramo kao $Q = n(n+2)\sum_{k=1}^{h} \frac{\hat{r}_k^2}{n-k}$, gdje n odgovara broju realizacija, h broj koraka za koji testiramo, a $\hat{r}_k$ vrijednost autokorelacije na koraku k [1]. U uvjetima istinitosti nul-hipoteze test statistika asimptotski ima χ^2 distribuciju s h stupnjeva slobode. Testove smo proveli na četiri uzorka za različite vrijednosti koraka. Uz u ovom poglavlju već predstavljene pseudoslučajne brojeve dobivene algortmima MSA i MT19937, dodajemo i uzorke iz LCG *minimal standard* algoritma pokrenutog sa sjemenom $seed = 1234$, te integrirane R-ove runif pokrenute nakon naredbe *set.seed(420, kind = "Marsaglia-Multicarry")* radi reproducibilnosti. Sljedeći tablični prikaz nam donosi rezultate provedenih Ljung-Box testova.

Korak	MSA	LCG	MT19937	runif
1	0.0919223	0.69571660	0.33345453	0.86839344
2	0.2403156	0.92629789	0.04280389	0.98517648
3	0.0053333	0.96926465	0.08818870	0.99025198
4	0.0101869	0.99269060	0.13850544	0.99595776
5	≈ 0	0.98533673	0.15066703	0.99053064
10	≈ 0	0.10808174	0.28025968	0.99800247
20	≈ 0	0.46922761	0.39685168	0.92623505
50	≈ 0	0.61928008	0.76566814	0.65739619
100	≈ 0	0.47265792	0.42184636	0.07576078
200	≈ 0	0.01964292	0.55825272	0.24319586
500	≈ 0	0.15775913	0.45150014	0.05128787
1000	≈ 0	0.02386635	0.60637805	0.05677297

Tablica 5: P-vrijednosti provedenih Ljung-Box testova za različite vrijednosti koraka

Rezultati testova su u skladu s onim što smo dosad zaključili o algoritmima predstavljenim u radu. Gotovo za svaku vrijednost koraka za MSA odbacujemo nulhipotezu i zaključujemo da postoji autokorelacija u danom nizu. Za ostala tri algoritma većinom nismo uspjeli odbaciti nulhipotezu na razini značajnosti 0.05. Sukladno tome nismo mogli donijeti isti zaključak.

U nastavku rada predstavit ćemo još neke standardne statističke testove koji se koriste pri testiranju generatora (pseudo)slučajnih brojeva kao i njihovu implementaciju u R jeziku.

5.4. Primjeri statističkih testova i njihova implementacija u R jeziku

Statističke testove za nizove pseudoslučajnih ili slučajnih brojeva možemo podijeliti u dvije kategorije. Testove namijenjene za nizove brojeva iz $[0, 1]$, kao i za nizove bitova. U ovom radu fokusirat ćemo se samo na testove namijenjene za testiranje nizova brojeva iz $[0, 1]$.

5.4.1. Run Testovi

Prvi po redu testovi koje ćemo predstaviti su *Run* testovi. Pseudoslučajne brojeve generirane algoritmom i dalje promatramo kao realizacije niza nezavisnih i jednakodistribuiranih slučajnih varijabli. Standardni run testovi, poznat i kao *Wald-Wolfowitz Runs Test*, radi na način da se odabere *žarište*, najčešće medijan, zatim se ispituje je li generirani broj manji ili jednak od medijana, ukoliko jest pridružimo mu -1, ukoliko nije pridružimo mu 1. Tako naš niz pseudoslučajnih brojeva transformiramo u niz s elementima iz skupa $\{-1, 1\}$. Broj elemenata koji su jednaki -1 u takvom nizu označimo s n_1, dok broj elemenata koji su jednaki 1 označimo s n_2. Sada je jasno da $n = n_1 + n_2$ odgovara ukupnom broju izgeneriranih pseudoslučajnih brojeva. *Runom* ili serijom nazivamo najduži podniz susjednih elemenata

koji se sastoji isključivo od -1 ili isključivo od 1. Primjerice, pogledajmo niz

$$-1, -1, 1, 1, 1, -1, -1 - 1, 1.$$

Na tom primjeru vidimo četiri serije, dvije negativne te dvije pozitivne. S R_1 označimo broj negativnih serija, dok s R_2 označimo broj pozitivnih serija. Tada je $R = R_1 + R_2$ ukupni broj serija.

Za velike n_1 i n_2 u uvjetima istinitosti nul-hipoteze R asimptotski ima normalnu distribuciju [18] s parametrima

$$\mu_1 = \frac{2n_1n_2}{n} + 1,\ \sigma_1^2 = \frac{2n_1n_2(2n_1n_2 - n)}{n^2(n-1)}.$$

Samo testiranje u R-u proveli smo funkcijom *runs.test()* iz paketa *'randtests'* sljedećom naredbom:

```
runs.test(stdMT19937.01, threshold = 0.5, alternative = "two.sided").
```

Idući tablični prikaz donosi nam p-vrijednosti takvih testova za generirane nizove pseudoslučajnih brojeva definiranih u prethodnom potpoglavlju.

Algoritam	p-vrijednost
MSA	0.01864
LCG	0.8833
MT19937	0.4112
runif	0.7654

Tablica 6: P-vrijednosti Wald-Wolfowitz run testova

Primjetimo da smo samo kod MSA algoritma odbacili nulhipotezu i zaključili da na razini značajnosti 0.05 možemo tvrditi da uzorak ne dolazi iz niza nezavisnih jednakodistribuiranih slučajnih varijabli. Dodatnom analizom utvrđeno je da imamo manje serija nego što je očekivani broj te zaključujemo da je ovdje problem podmiješanosti, tj. *under-mixinga* [15]. To tumačimo kao da algoritam "ne skače" dovoljno s jedne na drugu stranu medijana, već se u pravilu predugo zadržava na jednoj od strana prije nego prijeđe na drugu.

Drugi oblik *run* testova su tzv. *Runs up and down* testovi. Kod njih gledamo razliku dva susjedna člana niza te ukoliko je razlika negativna pridružimo -1, ukoliko je razlika pozitivna pridružimo 1. Ovakav tip testa, za razliku od Wald-Wolfowitz testa, može se primjeniti samo na numeričke varijable. Kao i u Wald-Wolfowitz testu i ovdje nam R označava broj serija, dok je m broj nenul razlika dvaju susjednih članova. U uvjetima istinitosti nulhipoteze za veliki n R asimptotski ima normalnu distribuciju s parametrima

$$\mu_2 = \frac{2m-1}{3},\ \sigma_2^2 = \frac{16m - 29}{90}.$$

Također postoji mogućnost da uzmemo unaprijed definiranu duljinu serije, primjerice 7, te usporedimo stvaran broj takvih serija s očekivanim brojem takvih serija.

5.4.2. Gap Testovi

Gap test ili test raskoraka promatra broj koraka između dvaju uzastopnih posjeta generatora nekom podskupu skupa vrijednosti koji generator može generirati. Kako radimo s generatorima koji generiraju brojeve iz intervala $(0, 1)$, definiramo skup $A = [a, b] \subset (0, 1)$. Na osnovi duljine intervala definiramo $p = b - a$ kao vjerojatnost da generirani pseudoslučajni broj bude iz $[a, b]$. Promatramo vektor $\mathbf{x}$ koji je realizacija niza nezavisnih i jednakodistribuiranih varijabli $X_1, \ldots, X_n$. Tada definiramo varijablu

$$G_i = \begin{cases} 1, & \text{za } a \leq X_i \leq b \\ 0, & \text{inače.} \end{cases}$$

S n_j ćemo označiti broj podnizova uzastopnih članova niza duljine j gdje se varijabla G_i realizirala s 1. Kako smo s p označili vjerojatnost da se generira broj iz segmenta $[a, b]$ vjerojatnost da se izgenerira podniz uzastopnih elemenata duljine j čiji bi svi elementi bili također iz tog segmenta možemo izračunati kao:

$$p_j = (1 - p)^2 p^j.$$

S p^j možemo izračunati vjerojatnost da j uzastopnih generiranih brojeva dolazi iz $[a, b]$, ali kako nam je uvjet da imamo točno j takvih brojeva u nizu, jasno je da dva člana niza koja omeđuju takav podniz ne smiju biti iz tog segmenta. Vjerojatnost da se to ostvari može se izračunati kao $(1 - p)^2$. Uz pretpostavku da su realizacije nezavisne dolazimo do gore navedenog izraza.

Test statistika je tada dana s:

$$S = \sum_{j=1}^{m} \frac{(n_j - np_j)^2}{np_j},$$

gdje je m maksimalni broj duljina. U uvjetima istinosti nul-hipoteze asimptotski ima χ^2 distribuciju s $m - 1$ stupnjeva slobode.

Testiranje smo proveli funkcijom *gap.test()* iz paketa *randtoolbox* [16] na istim testnim primjerima kao i u prethodnom potpoglavlju. Parametre smo zadali kao $a = 0$, $b = 0.1$.

Algoritam	p-vrijednost
MSA	$4 \cdot 10^{-19}$
LCG	0.097
MT19937	0.63
runif	0.78

Tablica 7: P-vrijednosti provedenih *gap* testova

Zaključujemo da možemo odbaciti nul-hipotezu samo u slučaju MSA algoritma te tvrdimo na razini značajnosti 0.05 da uzorak nije realizacija niza nezavisnih jednakodistribuiranih slučajnih varijabli.

5.4.3. Birthday spacing

Još jedan standardni test koji se primjenjuje pri testiranju generatora pseudoslučajnih brojeva je tzv. *Birthday spacing* test. Za ovaj test slučajno generiramo m, $m < n$ rođendana u n dana u godini te rođendane sortiramo u rastući poredak. Zatim napravimo razliku susjednih članova sortiranog niza kako bismo dobili razliku između dva susjedna rođendana. Ukoliko se vrijednost te razlike ponovi više od jednom tada govorimo o koliziji. Varijabla Y koja broji kolizije asimptotski ima Poissonovu distribuciju s parametrom $\lambda = m^3/(4n)$ (vidi [4], str. 81). U našem slučaju n možemo shvatiti kao maksimalni cijeli broj koji algoritam može izgenerirati.

Marsaglia i Tsang u svome radu [7] sugeriraju kako generator (pseudo)slučajnih brojeva koji prođe *Birthday spacing* test za parametre $n = 2^{32}$, $m = 2^{12}$ i $\lambda = 4$ će vjerojatno proći i za ostale vrijednosti parametra pa ćemo se ovdje koncentrirati na te parametre ukoliko to sam algoritam dopusti.

Za početak ćemo algoritmima generirati m rođendana i definirati broj dana n za svaki od algoritama koji ćemo testirati. Potom ćemo izračunati broj kolizija i taj postupak ponoviti 1000 puta, a zatim testirati dobivene rezultate s očekivanim χ^2 testom prilagodbe. Pogledajmo kako to izgleda na primjeru Mersenne Twister algoritma.

```
set.seed(15)
seed32 <- sample.int(2^32 - 1, size = 1000, replace = F)
coll <- c()
for (i in 1:1000) {
  x <- c()
  x <- MersenneTwister19937(seed32[i], 4096)[[1]]
  x <- sort(x)
  dx <- diff(x)
  dsx <- sort(dx)
  df <- as.data.frame(table(dsx))
  coll[i] <- nrow(df[df$Freq > 1,])
  i = i + 1
}

for (i in 1:1000){
  coll[i] <- ifelse(coll[i] >= 10, 10, coll[i])
}

for (i in 1:10) {
th.values[i] <- dpois(i - 1, 4)
}
th.values[11] <- 1 - sum(th.values[1:10])
```

```
chisq.test(x = table(coll), p = th.values)
```

Gore navedenim kodom prvo smo izgenerirali funkcijom *sample.int* 1000 pseudoslučajnih brojeva koji će nam služiti kao sjemena za pokretanje 1000 iteracija algoritma. Zatim smo unutar prve *for* petlje proveli postupak računanja kolizija u pojedinoj iteraciji na način kako smo naveli u radu. Drugom *for* petljom grupirali smo interacije gdje se dogodilo 10 ili više kolizija u jednu kategoriju. U vektor *th.values* spremili smo teorijske vrijednosti Poissonove distribucije s parametrom $\lambda = 4$ te smo konačno funkcijom *chisq.test* testirali pripadnost Poissonovoj distribuciji s tim parametrom.

Sljedeća tablica donosi prikaz ostavarenog broja kolizija u odnosu na očekivani broj u uvjetima istinitosti nulhipoteze.

Broj kolizija	0	1	2	3	4	5	6	7	8	9	10+
Očekivani broj	18.3	73.3	146.5	195.4	195.4	156.3	104.2	59.5	29.8	13.2	8.1
Stvarni broj	18	71	140	195	198	152	104	66	37	14	5

Tablica 8: Očekivane i realizirane vrijednosti kolizija za MT19937 algoritam na 1000 iteracija

P-vrijednost provedenog testa je 0.9365 što znači da na razini značajnosti 0.05 ne možemo odbaciti nul-hipotezu te tvrditi da slučajna varijabla Y koja broji kolizije ne dolazi iz Poissonove distribucije s parametrom $\lambda = 4$.

Sličan postupak proveli smo i za LCG *minimal standard* algoritam. Kako je najveći broj koji taj algoritam može generirati $2^{31} - 1$, postavit ćemo $n = 2^{31} - 1$ dok ćemo m ostaviti kao i dosad $m = 2^{12}$. To znači da je parametar $\lambda \approx 8$. P-vrijednost koju smo dobili provodeći χ^2 test prilagodbe ovdje manja je od $2 \cdot 10^{-16}$ što znači da na razini značajnosti 0.05 možemo zaključiti da slučajna varijabla Y koja broji kolizije ne dolazi iz Poissonove distribucije s parametrom $\lambda = 8$.

Nema potrebe da testiramo MSA algoritam jer je taj algoritam padao na puno jednostavnijim testovima od *Birthsday spacing* testa pa je iluzorno očekivati da će ovdje proći. Ovim testom smo konačno napravili i distinkciju u kvaliteti LCG *minimal standard* algoritma i MT19937 algoritma što nam dosad nije uspijevalo poći za rukom.

6. Generiranje pseudoslučajnih brojeva iz vjerojatnosnih distribucija

U dosadašnjem radu vidjeli smo generiranje pseudoslučajnih brojeva iz uniformne distibucije na intervalu $(0,1)$. Ponekad nam to nije dosta, već želimo generirati pseudoslučajne brojeve koji prate neku unaprijed zadanu distribuciju. Teoremom o inverznoj tranformaciji možemo generirati pseudoslučajne brojeve iz bilo koje proizvoljne distribucije. Prije nego ga iskažemo i pokažemo, imamo sljedeći teorem.

Teorem 6.1. *Neka je X neprekidna slučajna varijabla s funkcijom distribucije F_X. Tada slučajna varijabla $Y = F_X(X)$ ima uniformnu distribuciju na intervalu $(0,1)$, tj. $Y \sim U(0,1)$.*

Dokaz. Za danu slučajnu varijablu X, definiramo $Y = F_X(X)$. Također definiramo i $F_X^{-1}(y)$, $y \in [0,1]$ kao inverz funkcije distribucije F_X ukoliko postoji. Ukoliko ne postoji tada se definira kao

$$F_X^{-1}(y) = \inf\{x \in \mathbb{R} : F_X(x) \geq y\}. \tag{5}$$

Sada imamo:

$$\begin{aligned} F_Y(y) &= P(Y \leq y) = P(F_X(X) \leq y) \\ &= \begin{cases} 0, & \text{za } y < 0 \\ P(X \leq F_X^{-1}(y)), & \text{za } y \in [0,1) \\ 1, & \text{za } y \geq 1 \end{cases} \\ &= \begin{cases} 0, & \text{za } y < 0 \\ F_X(F_X^{-1}(y)), & \text{za } y \in [0,1) \\ 1, & \text{za } y \geq 1 \end{cases} \\ &= \begin{cases} 0, & \text{za } y < 0 \\ y, & \text{za } y \in [0,1) \\ 1, & \text{za } y \geq 1 \end{cases} \end{aligned}$$

Primjetimo da je F_Y funkcija distribucije uniformne slučajne varijable na intervalu $(0,1)$. Prema tome $Y \sim U(0,1)$. □

Teorem 6.2. (Teorem o inverznoj transformaciji) *Neka je $Y \sim U(0,1)$ te neka je F_X funkcija distribucije slučajne varijable X. Označimo s $F_X^{-1}(y)$, $y \in [0,1]$ inverz definiran kao u (5). Tada slučajna varijabla $F_X^{-1}(Y)$ ima funkciju distribucije F_X.*

Dokaz. Moramo pokazati da vrijedi: $P(F_X^{-1}(Y) \leq x) = F_X(x)$. Pretpostavimo prvo da je F_X neprekidna, što povlači da je invertibilna jer je neprekidna i strogo rastuća [3]. Pokazat ćemo jednakost događaja $\{F_X^{-1}(Y) \leq x\} = \{Y \leq F_X(x)\}$. Neka je $Y \sim U(0,1)$. Tada vrijedi $P(Y \leq y) = y, \text{za } y \in (0,1)$. Iz

$$P(F_X^{-1}(Y) \leq x) = P(Y \leq F_X(x)) = F_X(x)$$

vidimo da varijabla $F_X^{-1}(Y)$ ima funkciju distribucije F_X.

Promotrimo sada slučaj kada F nije neprekidna. Kako je $F_X(F_X^{-1}(y)) = y$ i F_X monotono rastuća funkcija vrijedi da ako je $F_X^{-1}(Y) \leq x$, tada je $Y = F_X(F_X^{-1}(Y) \leq F_X(x)$. Slično vrijedi $F_X^{-1}(F_X(x)) = x$ pa ako je $Y \leq F_X(x)$ tada je $F_X^{-1}(Y) \leq x$. Zaključujemo da smo pokazali jedakost događaja kako smo i najavili. Općenito, lako se pokaže da vrijedi

$$\{Y < F_X(x)\} \subseteq \{F_X^{-1}(Y) \leq x\} \subseteq \{Y \leq F_X(x)\},$$

te da je vjerojatnost ovih događaja ista zbog $P(Y = F_X(x)) = 0$, jer je Y neprekidna slučajna varijabla. □

U prethodnim poglavljima vidjeli smo da generiranjem niza pseudoslučajnih brojeva iz $\{0, 1, \ldots, n\}$, možemo dobiti niz pseudoslučajnih brojeva iz uniformne distribucije na intervalu $(0, 1)$ u računalnom smislu koji zatim primjenom teorema 6.2. možemo transformirati u niz pseudoslučajnih brojeva iz proizvoljne distribucije. U nastavku poglavlja pogledat ćemo kako to izgleda za neke od ostalih vjerojatnosnih distribucija.

6.1. Generiranje pseudoslučajnih brojeva iz neprekidnih distribucija

Generiranje podataka iz neprekidnih distribucija metodom inverzne transformacije prikazat ćemo na primjeru eksponencijalne distribucije.

6.1.1. Eksponencijalna distribucija s parametrom λ

Neka je X eksponencijalna slučajna varijabla s očekivanjem λ. Funkcija distribucije od X, $F_X(x)$ zadana s:

$$F(x) = \begin{cases} 1 - e^{-\lambda x}, & \text{za } x \geq 0 \\ 0, & \text{za } x < 0. \end{cases}$$

Postavimo $x = F_X^{-1}(y)$. Tada imamo $y = F_X(x) = 1 - e^{-\lambda x}$, odnosno $e^{-\lambda x} = 1 - y$. Kada logaritmiramo dobivamo $-\lambda x = \ln(1 - y)$ što znači da je $x = \frac{-\ln(1-y)}{\lambda}$.

Sada je jasno da ukoliko imamo generator pseudoslučajnih brojeva koji generira brojeve y iz uniformne distribucije na intervalu $(0, 1)$ uvrštavanjem u izraz

$$x = \frac{-\ln(1 - y)}{\lambda}$$

možemo dobiti pseudoslučajne brojeve koji prate eksponencijalnu distribuciju s parametrom λ.

6.2. Generiranje pseudoslučajnih brojeva iz diskretnih distribucija

Pretpostavimo da želimo generirati pseudoslučajne brojeve iz neke diskretne distribucije s gustoćom

$$P\{X = x_j\} = p_j, \quad j = 0, 1, \ldots, \quad \sum_j p_j = 1.$$

Da bismo postigli to prvo generiramo pseudoslučajni broj y koji je realizacija varijable $Y \sim U(0,1)$ i postavimo:

$$X = \begin{cases} x_0, & \text{ako je } y < p_0 \\ x_1, & \text{ako je } p_0 \leq y < p_0 + p_1 \\ \vdots & \\ x_j, & \text{ako je } \sum_{i=0}^{j-1} p_i \leq y < \sum_{i=0}^{j} p_i \\ \vdots & \end{cases} \tag{6}$$

Kako je $0 < a < b < 1$, $P\{a \leq Y < b\} = b - a$, imamo:

$$P\{X = x_j\} = P\left\{ \sum_{i=0}^{j-1} p_i \leq Y < \sum_{i=0}^{j} p_i \right\} = p_j$$

prema tome varijabla X ima traženu diskretnu distibuciju (vidjeti [17]).

6.2.1. Generiranje iz Bernoullijeve distribucije

Neka je X slučajna varijabla iz Bernoullijeve distribucije s parametrom p. Tada su vjerojatnosti realizacija od X dane izrazom:

$$P(X = k) = p^k (1-p)^{1-k}, \quad k \in \{0, 1\}.$$

Koristeći zaključke iz prethodnog potpoglavlja vidimo da možemo generirati pseudoslučajne brojeve iz Bernoullijeve distribucije s parametrom p, tako da zadamo:

$$X = \begin{cases} 0, & \text{ako je } 0 < y < p \\ 1, & \text{ako je } p \leq y < 1 \end{cases},$$

gdje je y realizacija slučajne varijable $Y \sim U(0,1)$.

```
x <- stdMT19937.01
for (i in 1:length(x)) {
y[i] <- ifelse(x[i] < 0.5, 0, 1)
}
chisq.test(x = table(y), p = c(0.5, 0.5))
```

Gore navedenim kodom simulirali smo od niza pseudoslučajnih brojeva *stdMT19937.01* niz pseudoslučajnih brojeva koji su realizacija Bernoullijeve slučajne varijable s parametrom $p = 0.5$. Provodeći χ^2 test prilagodbe dobili smo p-vrijednost 0.6745 te zaključujemo da na razini značajnosti 0.05 ne možemo odbaciti nul-hipotezu, tj. ne možemo tvrditi da uzorak ne dolazi iz Bernoullijeve distribucije.

6.2.2. Generiranje iz geometrijske distribucije s parametrom p

Kažemo da slučajna varijabla X ima geometrijsku distribuciju s parametrom p ako vrijedi:

$$P\{X = i\} = pq^{i-1}, \quad i \geq 1, \quad q = 1 - p.$$

Tada X možemo shvatiti kao model za broj pokušaja do prvog uspjeha nezavisnih pokusa, gdje je u svakom pokusu vjerojatnost uspjeha p.

Kako je:

$$\begin{aligned}\sum_{i=1}^{j-1} P\{X = i\} &= 1 - P\{X > j - 1\} \\ &= 1 - P\{\text{prvih } j-1 \text{ pokušaja su neuspjesi}\} \\ &= 1 - q^{j-1}\end{aligned}$$

možemo generirati pseudoslučajne brojeve iz geometrijske distribucije s parametrom p tako da zadamo $X = j$ za j za koji vrijedi:

$$1 - q^{j-1} \leq y < 1 - q^j,$$

odnosno

$$q^j < 1 - y \leq q^{j-1},$$

gdje je y realizacija slučajne varijable $Y \sim U(0, 1)$. Drugim riječima X možemo definirati izrazom:

$$X = \min\{j : q^j < 1 - Y\}.$$

Kako je logaritam monotona funkcija pa za $a < b$ vrijedi da je $\log(a) < \log(b)$ zaključujemo da se X može izraziti i kao:

$$\begin{aligned}X &= \min\{j : j \cdot \log(q) < \log(1 - Y)\} \\ &= \min\{j : j > \frac{\log(1 - Y)}{\log(q)}\} \\ &= \left\lfloor \frac{\log(1 - Y)}{\log(q)} \right\rfloor + 1,\end{aligned}$$

gdje je $f(x) = \lfloor x \rfloor$ funkcija koja decimalni broj zaokruži na najveći cijeli broj manji od tog broja.

```
p = 0.2
q = 1 - p
x <- stdMT19937.01
for (i in 1:length(x)) {
y[i] <- floor(log(1-x[i]) / log(q)) + 1
}
```

Ovdje smo također iz niza pseudoslučajnih brojeva *stdMT19937.01* navedenim algoritmom simulirali generiranje pseudoslučajnih brojeva iz geometrijske distribucije s parametrom $p = 0.2$. P-vrijednost provedenog χ^2 testa je 0.2619 što znači da na razini značajnosti 0.05 ne možemo odbaciti nul-hipotezu odnosno tvrditi da uzorak ne dolazi iz geometrijske distribucije s parametrom $p = 0.2$. Kada umjesto niza *stdMT19937.01* upotrijebimo niz pseudoslučajnih brojeva *stdLCG.01* koji je dobiven *minimal standard* algoritmom p-vrijednost testa je 0.0003131 što znači da u tom slučaju na razini značajnosti 0.05 odbacujemo nul-hipotezu te tvrdimo da tako dobiveni niz nije realizacija niza nezavisnih i jednakodistribuiranih slučajnih varijabli iz geometrijske distribucije s parametrom $p = 0.2$.

6.2.3. Generiranje iz Poissonove distribucije s parametrom λ

Za slučajnu varijablu X kažemo da ima Poissonovu distribuciju s parametrom λ ako vrijedi:

$$p_i = P\{X = i\} = e^{\lambda}\frac{\lambda^i}{i!}.$$

Uvrštavanjem u gore navedeni izraz lako se izvede sljedeća relacija:

$$p_{i+1} = \frac{\lambda}{i+1}p_i$$

koja će nam poslužiti da transformiramo niz pseudoslučajnih brojeva. Kada smo uspješno odredili $p_0, p_1, \ldots$ tada za svaki pseudoslučajni broj koji želimo generirati primjenimo postupak naveden u (6). Kako smo taj postupak već pokazali, nećemo ga ponavljati u ovom potpoglavlju.

Literatura

[1] P. J. Brockwell, R. A. Davis, *Introduction to Time Series and Forecasting, 2nd edition*, Springer-Verlag New York, New York, 2002.

[2] D. Davis, R. Ihaka, P. Fenstermacher, *Cryptographic randomness from air turbulence in disk drives*, Annual International Cryptology Conference: Advances in Cryptology — CRYPTO '94(1994), 114-120.

[3] L. Devroye, *Non-Uniform Random Variate Generation*, Springer Science Business Media, Inc., New York, 1986.

[4] J. E. Gentle, *Random number generation and Monte Carlo methods*, Springer Science Business Media, Inc., New York, 2006.

[5] P. L'Ecuyer, R. Simard, *TestU01 - A Software Library in ANSI C for Empirical Testing of Random Number Generators*, ACM Transactions on Mathematical Software 33(2007), čl. 22.

[6] George Marsaglia, *Random number generators*, Journal of Modern Applied Statistical Methods 2(2003), 2-13.

[7] George Marsaglia, Wai Wan Tsang, *Some difficult-to-pass tests of randomness*, Journal of statistical software 7(2002), 1-9.

[8] George Marsaglia, Wai Wan Tsang, Jingbo Wang, *Evaluating Kolmogorov's Distribution*, Journal of statistical software 8(2003), 1-15.

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

[10] Duncan J. Murdoch, Yu-Ling Tsai, James Adcock, *P-Values are Random Variables*, The American Statistician 62(2008), 242-245.

[11] John Von Neumann, *Various Techniques Used in Connection with Random*, NBS Applied Mathematics Series 12(1951), 36–38.

[12] Stephen K. Park, Keith W. Miller, *Random number generators: good ones are hard to find*, Communications of the ACM 12(1998), 1192-1201.

[13] R dokumentacija, paket *base*, *https://stat.ethz.ch/R-manual/R-devel/RHOME/library/base/html/Random.html*

[14] R dokumentacija, paket *stats* verzija 3.7.0, *https://stat.ethz.ch/R-manual/R-devel/library/stats/html/00Index.html*

[15] R DOKUMENTACIJA, PAKET *randtests* VERZIJA 1.0, *https://cran.r-project.org/web/packages/randtests/randtests.pdf*, 2005.

[16] R DOKUMENTACIJA, PAKET *randtoolbox* VERZIJA 1.17, *https://www.rdocumentation.org/packages/randtoolbox/versions/1.17/topics/gap.test*

[17] SHELDON M. ROSS, *Simulation, 4th edition*, Elsevier Inc., San Diego, 2006.

[18] Y. WANG, *Nonparametric Tests for Randomness*, Chicago, 2003.

[19] P. WU, *Multiplicative, congruential random-number generators with multiplier* $\pm 2^{k_1} \pm 2^{k_2}$ *and modulus* $2^p - 1$, ACM Transactions on Mathematical Software 23(1997), 255–265.

Sažetak

Sažetak. Tema ovog rada jest generiranje pseudoslučajnih brojeva. Na početku rada predstavljamo slučajne i pseudoslučajne brojeve te generatore slučajnih i pseudoslučajnih brojeva kao i razlike među njima. Prikaz algoritama počinjemo s Middle-square algoritmom te linearnim kongruentnim generatorom, a završavamo Mersenne Twister algoritmom. U nastavku objašnjavamo statističke testove koje koristimo za testiranje generatora pseudoslučajnih brojeva te provodimo testiranja. Zadnji dio rada bavi se problemom generiranja pseudoslučajnih brojeva iz proizvoljne distribucije. Sva programska podrška korištena u ovom radu isprogramirana je u R jeziku.

Ključne riječi: Generator slučajnih brojeva, generator pseudoslučajnih brojeva, Middle-square algoritam, Linearni kongruentni generatori, Mersenne Twister, Kolmogorov-Smirov test, Hi-kvadrat test prilagodbe, Run and gap testovi, Birthday spacing test, metoda inverzne transformacije

Title and summary

Summary. Main subject of this paper is generating pseudorandom numbers. At the beginning we present random and pseudorandom numbers, as well as random number generators and pseudorandom number generators. First pseudorandom number generator we meet is Middle-square algorithm, followed by Linear congruential generator and finally Mersenne Twister algorithm. Later we present statistical tests for testing randomness of pseudorandom number generators while conducting the tests themselves. The final chapter of this paper describes inverse transform method for generating pseudorandom numbers from arbitrary distributions. Algorithms and statistical tests are programmed in R language.

Keywords: Random number generator, pesudorandom number generator, Middle-square algorithm, Linear congruential generators, Mersenne Twister, Kolmogorov-Smirnov test, Chi-square goodness of fit, Run and gap tests, Birthday spacing test, inverse transform method.

Životopis

Rođen sam u Osijeku 19. siječnja 1993. godine. Osnovnu školu Ivana Filipovića sam završio 2007. godine te iste godine upisao III. gimnaziju u Osijeku. Srednju školu završavam 2011. godine kada i upisujem sveučilišni preddiplomski studij matematike na Odjelu za matematiku sveučilišta J.J. Strossmayera u Osijeku. Isti studij završavam 2015. godine završnim radom na temu Euklidskih prostora te upisujem sveučilišni diplomski studij matematike smjer Financijska matematika i statistika. U trenutku pisanja rada zaposlen sam u SPAN d.o.o.