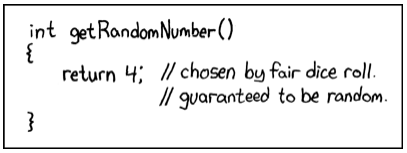
L02 – Zufall

19. April 2021

Contents

Was ist Zufall? Wikipedia (verkürzt): Zufall ist die Abwesenheit kausaler Zusammenhänge.

Was ist eine Zufallszahl?



Was ist ein Zufallszahlengenerator?

Eine Zahl ist nicht "zufällig" oder "nicht zufällig", sondern der Prozess, der sie erzeugt.

Ein Zufallszahlengenerator (*random number generator*, RNG) ist ein Verfahren zur Erzeugung einer Folge von Zahlen, die einer bestimmten Wahrscheinlichkeitsverteilung folgen.

1 Näherung von Wahrscheinlichkeiten

Definition: Sei $(P, \mathbb{R}, \mathcal{B})$ ein Wahrscheinlichkeitsraum. Wir bezeichnen eine Folge $(z_i)_{i\in\mathbb{N}} \subset \mathbb{R}$ als **Zufallszahlengenerator (RNG)** der Verteilung P, falls für alle Ereignisse $B \in \mathcal{B}$ gilt

$$\left| \frac{\#\{1 \le i \le n \colon z_i \in B\}}{n} - P(B) \right| \xrightarrow{n\to\infty} 0\,.$$

Beispiel: Sei $X \sim \mathcal{N}(0,1) =: P$ standardnormalverteilt. Wir interessieren uns für die Wahrscheinlichkeit $p := \mathbb{P}(X \in [-1,3])$, also $p = P(B)$ mit $B := [-1,3]$, sind aber zu faul das entsprechende Integral auszurechnen. Stattdessen haben wir einen Zufallszahlengenerator $(z_i)_{i\in\mathbb{N}}$ der Verteilung $\mathcal{N}(0,1)$ zur Verfügung. Damit nähern wir p durch p_{n_0} an. Hierbei ist $p_n := \frac{k_n}{n}$, wobei k_n die Anzahl der z_i mit $i \le n$ ist, die in $[-1,3]$ liegen. Wegen $p_n \xrightarrow{n\to\infty} p$, können wir hoffen, dass p_{n_0} nahe bei p liegt, wenn n_0 genügend groß gewählt wurde.

Aufgabe: Wir vollziehen diese Näherung in R nach. Vervollständige folgenden Code, sodass `approx_p()` die Wahrscheinlichkeit $\mathbb{P}(X \in [a,b])$ wie oben beschrieben mit `n0` Zahlen aus dem RNG `rnorm()` (siehe `?rnorm`) annähert.

```r
approx_p <- function(n0, a, b) {
  # TODO
}
# berechne wahres p durch Integration
p <- integrate(dnorm, lower=-1, upper=3)$value
p
# berechne Differenz zwischen Näherung und wahrem Wert für verschiedene n0
abs(p - approx_p(1e1, -1, 3))
abs(p - approx_p(1e3, -1, 3))
abs(p - approx_p(1e5, -1, 3))
```

2 Linearer Kongruenzgenerator

Selbst RNGs, die nur Näherungen realisieren, sind schwierig am PC umzusetzen. Außerdem ist es für den praktischen Einsatz eher wichtig, dass auch die ersten Zahlen der Folge genügend zufällig aussehen, anstatt der asymptotischen Eigenschaft.

Zum praktischen Einsatz kommen PRNGs (Pseudo-RNGs), welche das Verhalten von RNGs imitieren ohne deren mathematische Eigenschaften exakt zu erfüllen.

Ein simples Beispiel für ein PRNG ist der Lineare Kongruenzgenerator:

Seien

- $m \in \{2, 3, 4, \dots\}$
- $a, b \in \{1, \dots, m-1\}$
- $y_1 \in \{1, \dots, m-1\}$

Definiere die Folge $(y_i)_{i \in \mathbb{N}}$ durch $y_{i+1} = (ay_i + b) \mod m$. Dabei entspricht $u \mod v$ hier `u %% v`.

Aufgabe: Schreibe eine Funktion `lcg_prng()`, die $(y_1, \dots, y_n)$ ausgibt.

```r
# linear congruential generator (lcg)
lcg_prng <- function(n, y1, m, a, b) {
  # TODO
}
```

Die Parameter a, b, m bestimmen die Qualität des PRNG.

```r
lcg_prng(18, y1=1L, m=9L, a=4L, b=7L)
##  [1] 1 2 6 4 5 0 7 8 3 1 2 6 4 5 0 7 8 3
lcg_prng(18, y1=1L, m=9L, a=3L, b=3L)
##  [1] 1 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

Satz von Knuth

Lineare Kongruenzgeneratoren erreichen genau dann ihre maximal mögliche Periodenlänge m, wenn folgende 3 Bedingungen erfüllt sind:

- b und m sind teilerfremd.
- Jeder Primfaktor von m teilt $a - 1$.
- Wenn m durch 4 teilbar ist, dann auch $a - 1$.

In diesem Fall erzeugt der Generator jede Zahl von 0 bis m - 1 genau einmal und beginnt danach von vorn.

Nachteile:

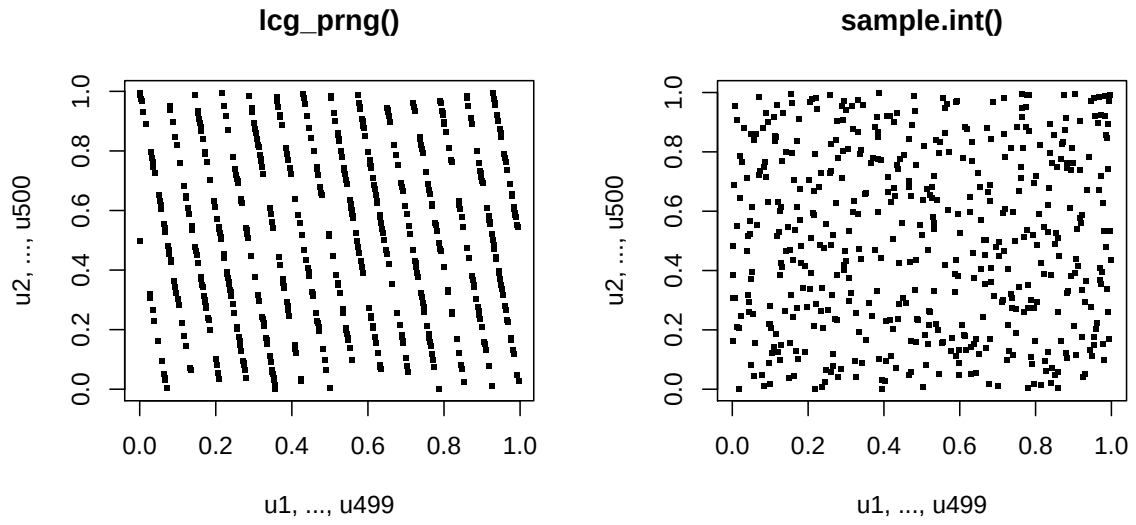
1) Angenommen d teilt m und Bedingungen aus Satz von Knuth sind erfüllt, dann $y_{i+d} \equiv y_i \mod d$.

```r
y <- lcg_prng(18, y1=1L, m=9L, a=4L, b=7L)
y[1:(18-3)] %% 3
##  [1] 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0
y[(1+3):18] %% 3
##  [1] 1 2 0 1 2 0 1 2 0 1 2 0 1 2 0
```

2) **Satz von Marsaglia:**

Sei $u_i = \frac{y_i}{m}$. Dann liegen die Vektoren $(u_1, \ldots, u_k), (u_2, \ldots, u_{k+1}), \cdots \in \mathbb{R}^k$ auf maximal $(m \cdot k!)^{\frac{1}{k}}$ parallelen Hyperebenen.

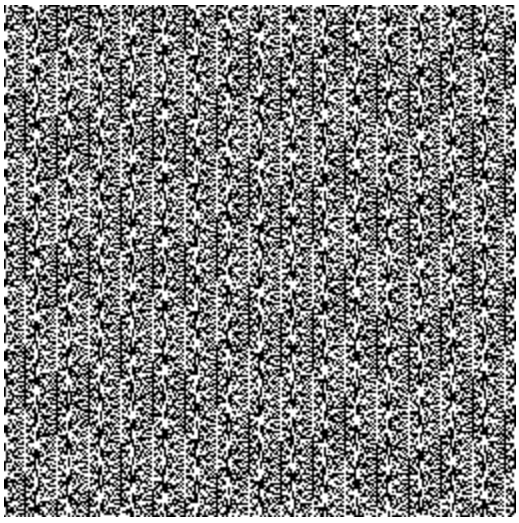
```r
par(mfrow=c(1,2))
m <- 2^11
x <- lcg_prng(500, y1=1, m=m, a=1017, b=1)
plot(x[1:499]/m, x[2:500]/m, pch=".", cex=4,
     xlab="u1, ..., u499", ylab="u2, ..., u500", main="lcg_prng()")
# Vergleich mit R-Funktion sample.int(), siehe ?sample
x <- sample.int(m, 500)
plot(x[1:499]/m, x[2:500]/m, pch=".", cex=4,
     xlab="u1, ..., u499", ylab="u2, ..., u500", main="sample.int()")
```



Hinweis: R nutzt anstatt eines linearen Kongruenzgenerators den Mersenn-Twister https://en.wikipedia.org/wiki/Mersenne_Twister.

3 Probleme mit schlechten PRNGs

Bei schlechten PRNGs können vorhersehbare Muster entstehen. Ein Beispiel ist die Funktion `rand()` einer älteren Version der Programmiersprache PHP. Werden mit ihr "zufällig" schwarze und weiße Pixel erzeugt, entsteht zB folgendes Bild.



Es lässt klare Strukturen erkennen. Gilt dies auch für unseren `lcq_prng()`? Wie verhält es sich mit den entsprechenden R-Funktionen?

Aufgabe: Erzeuge `w` × `h` Matrizen mit Einträgen 0 oder 1, die zufällig gewählt sind. Nutze für `mat_sample` die Funktion `sample()` (siehe `?sample`) und für `mat_lcq` die Funktion `lcq_prng()` wie im Code unten angegeben. Führe dann `image()` aus, um das entsprechende Bild zu erhalten.

```
w <- 256L
h <- 256L
set.seed(0)

# TODO: img_sample <-

m <- 2^11
y <- lcg_prng(w*h, y1=1, m=m, a=1017, b=1)
# TODO: img_lcg <-

par(mfrow=c(1,2), mar=c(1,1,1,1))
image(
  img_sample,
  col = c("black", "white"),
  axes = FALSE,
  useRaster = TRUE,
  asp=1, # fixes aspect ratio
  main="sample()")
image(
  img_lcg,
  col = c("black", "white"),
  axes = FALSE,
  useRaster = TRUE,
  asp=1,
  main="lcg_prng()")
```

4 Inversionsmethode

Durch entsprechende Skalierung erhält man aus einer Gleichverteilung über die Zahlen 1 bis 2^{32} eine uniforme Verteilung auf $[0, 1]$, wobei Werte zu den (maschinenrepräsentierbaren) `double`-Zahlen in $[0, 1]$ gerundet

werden.

Um Zufallszahlen einer anderen Verteilung zu generieren, kann die **Inversionsmethode** genutzt werden.

Seien $U \sim \mathsf{Unif}([0,1])$ und $F_\mu \colon \mathbb{R} \to [0,1]$ die Verteilungsfunktion einer Verteilung μ auf $\mathbb{R}$. Definiere $X := F_\mu^{-1}(U)$, wobei $F_\mu^{-1} \colon [0,1] \to \mathbb{R}$, $F_\mu^{-1}(p) = \inf\{x \in \mathbb{R} \colon p \le F_\mu(x)\}$ die Quantilfunktion von μ ist. Dann gilt für $t \in \mathbb{R}$,

$$F_X(t) = \mathbb{P}(X \le t) = \mathbb{P}(F_\mu^{-1}(U) \le t) = \mathbb{P}(U \le F_\mu(t)) = F_\mu(t)\,.$$

Daher gilt $F_X = F_\mu$, also $X \sim \mu$.

Aufgabe: Benutze `runif()` und `qnorm()` (siehe `?Distributions`), um unabhängige uniformverteilte Zufallszahlen `u` zu erzeugen und diese zu standardnormalverteilten Zufallszahlen `x` zu transformieren. Vergleiche die Verteilung der `u` und der `x` mit der gleichen Anzahl an durch `rnorm()` erzeugten standardnormalverteilten Zufallszahlen `y` mittels eines QQ-Plots.

Ein Quantil-Quantil-Diagramm (QQ-Plot) ist ein exploratives, grafisches Werkzeug, in dem die Quantile zweier statistischer Variablen gegeneinander abgetragen werden, um ihre Verteilungen zu vergleichen.

`qqplot(x, y)` entspricht `plot(sort(x), sort(y))`. Bei identischer Verteilung der Einträge von `x` und `y` nähert sich der Plot der Geraden $x \mapsto x$ an (für große Anzahl an Beobachtungen). Bei genügend unterschiedlicher Verteilung weicht der Plot von der Geraden ab.

Bemerkung: Offensichtlich liefert der Plot an sich keine mathematisch belastbaren Aussagen. Ein statistischer Test auf Gleichheit von Verteilungen ist zB der Kolmogorov–Smirnov-Test https://de.wikipedia.org/wiki/Kolmogorow-Smirnow-Test.

```r
n <- 300
# TODO: u <-
# TODO: x <-
# TODO: y <-

par(mfrow=c(1,2), mar=c(2,2,1,1))
qqplot(u, y, main="uniform vs normal")
abline(0, 1, col="gray")
qqplot(x, y, main="normal (inversion) vs normal")
abline(0, 1, col="gray")
```

5 Bonus: Statistical Tests for Randomness

Wie kann eine endliche Folge von Zahlen auf Zufälligkeit geprüft werden?

Bei 10-maligem fairen Münzwurf ist die Folge `0000000000` genauso wahrscheinlich wie `0101111001`.

Wir beobachten eine Realisierung von $X = (X_1, \dots, X_n)$ und möchten Testen, ob sie einer Produktverteilung μ^n entstammt, dh ob $X_1, \dots, X_n$ unabhängig, identisch verteilt (**uiv**) sind mit $X_i \sim \mu$.

Beispiel: Entstammte die Folge $X = 0101010101$ ($X_1 = 0, X_2 = 1, \dots$) einem fairen Münzwurf $\mu = \mathsf{Unif}(\{0,1\})$?

Dazu konstruieren wir einen statistischen Test zum Niveau $\alpha \in [0,1]$ für die Nullhypothese $H_0 \colon \mathbb{P}^X = \mu^n$ gegen die Alternative $H_1 \colon \mathbb{P}^X \neq \mu^n$.

Bei einem Test auf "Zufälligkeit" eines PRNG sind wir meist vor allem an der Unabhängigkeits-Annahme interessiert.

Mit Statistiken wie `mean()` oder `var()` können wir nicht testen, ob in der Abfolge der Elemente von X eine Struktur erkennbar ist, da `mean()` und `var()` **permutationsinvariant** sind.

Für den Münzwurf sind Runs-Tests hilfreich. Ein **Run** ist eine Teilfolge gleicher Beobachtungen. ZB $x = 0010001100$ enthält 5 Runs, nämlich `00`, `1`, `000`, `11` und `00`.

Wir berechnen die Statistiken $s_{\mathsf{len}}(x) =$ Anzahl der Runs und $s_{\mathsf{max}}(x) =$ Länge des längsten Runs.

Dann vergleichen wir $s_{\mathsf{len}}(x)$, $s_{\mathsf{max}}(x)$ für unsere Beobachtung x mit der Verteilung von $s_{\mathsf{len}}(X)$, $s_{\mathsf{max}}(X)$ unter Annahme der Nullhypothese.

Die Verteilungen von $s_{\mathsf{len}}(X)$ und $s_{\mathsf{max}}(X)$ können explizit berechnet werden. Alternativ können die Verteilungen durch eine Simulation empirisch angenähert werden: Wir berechnen $s_{\mathsf{len}}(y)$ und $s_{\mathsf{max}}(y)$ für sehr oft wiederholte Ziehungen von y aus $\mathsf{Unif}(\{0,1\})^n$.

```
# Welche der beiden Sequenzen entstammt einem fairen Münzwurf (mit unabhängigen Würfe)?
x <- c(1,1,1,0,1,0,0,1,1,1,0,1,0,0,1,1,0,1,1,1,
       0,0,0,0,0,0,1,1,1,1,1,1,0,0,1,0,0,1,0,0,
       1,1,1,1,1,0,1,0,1,1,1,0,1,1,1,0,1,1,0,0,
       1,0,1,0,0,1,1,0,1,1,0,1,0,0,0,0,0,0,0,0,
       1,0,1,0,1,0,0,0,1,0,1,0,1,1,1,1,0,0,1,1)
y <- c(1,0,1,0,1,1,0,1,0,0,1,1,0,0,0,1,0,1,1,1,
       0,0,1,1,0,0,1,0,1,1,0,1,0,1,0,1,1,0,1,0,
       0,1,0,1,0,0,1,0,0,1,1,1,0,1,0,1,0,1,0,1,
       0,1,0,1,1,0,1,0,1,0,0,1,0,0,0,1,0,0,1,1,
       0,1,0,0,0,1,1,0,0,1,0,1,0,0,1,0,1,0,1,1)

c(length(x), length(y)) # gleiche Länge
## [1] 100 100
table(x) # zähle Anzahl der 1en und 0en
## x
##  0  1
## 47 53
table(y)
## y
##  0  1
## 51 49
```

Aufgabe: Nutze die Funktion `rle()` (siehe `?rle`) um die Funktion `get_runs_statistic()` zu vervollständigen. Der Input `v` ist ein atomarer Vektor mit Elementen `0` und `1`. Wie angegeben ist der Rückgabewert ein atomarer Vektor der Länge 3 mit der Anzahl der Runs, der Länge des längsten Runs und der Anzahl der 1en in `v`.

```
get_runs_statistic <- function(v) {
  # TODO
  # TODO: return(c(
  #  number_of_runs = ???,
  #  longest_run = ???,
  #  number_of_ones = ???))
}
```

Der restliche Code generiert einen hübschen Plot. Gib nach Betrachtung des Plots einen Tipp ab, welche der beiden Sequenzen `x` und `y` von einem fairen Münzwurf stammt und für welche ein Mensch gebeten wurde, sich eine "zufällige" Sequenz auszudenken.

```
get_confidence_region <- function(v, alpha) {
  stopifnot(length(alpha)==1)
  q <- 1 - alpha
  stopifnot(q > 0 && q <= 1)
  i <- 0
  m <- mean(v)
  while(
    sum(m-i <= v & v <= m+i) / length(v) < q
```

```
  ) i <- i+0.5
  return(list(left=ceiling(m-i)-0.5, right=floor(m+i)+0.5))
}

n <- length(x)

rx <- get_runs_statistic(x)
ry <- get_runs_statistic(y)


# simulate fair coin tosses

library(tibble)
set.seed(0)

simu <- replicate(1e5, # execute following function call 1e5 times
  get_runs_statistic(sample(0:1, n, replace=TRUE)))
data <- as_tibble(t(simu))


# Plot results

layout(
  rbind(
    c(1, 2),
    c(3, 3)
  ),
  heights=c(1, 1),
)
clrs <- list(x = "red", y = "blue", conf = "#00FF0040")
mark_lwd <- 3
invisible(lapply(names(data), function (nm) {
  v <- data[[nm]]
  hist(v, breaks=(min(v)-0.5):(max(v)+0.5), main=nm, xlab=NULL)
  y_axis_max <- par("yaxp")[2]
  segments(rx[nm], 0, y1=y_axis_max, col=clrs$x, lwd=mark_lwd)
  segments(ry[nm], 0, y1=y_axis_max, col=clrs$y, lwd=mark_lwd)
  conf <- get_confidence_region(v, 0.05)
  rect(conf$left, 0, conf$right, y_axis_max, col=clrs$conf, border=NA)
}))
legend(
  "topright",
  c("x", "y", "95% confidence"),
  col=c(clrs$x, clrs$y, clrs$conf),
  lwd=mark_lwd)
```