YOUNG-GEUN statistics

Statistical Computing

R Lab

O RLY?

Young-geun Kim

R Lab for Statistical Computing

Young-geun Kim
Department of Statistics, SKKU
dudrms33@g.skku.edu

2019-04-01

Contents

Welcome

Statistical computing mainly treats useful simulation methods.

Statistical Computing

We first look at *random generation* methods. Lots of simulation methods are built based on this random numbers.

Sampling from a fininte population

Generating random numbers is like sampling. From finite population, we can sample data with or without replacement. For example of sampling with replacement, we toss coins 10 times.

```r
sample(0:1, size = 10, replace = TRUE)
```

```
 [1] 1 0 0 1 0 1 1 0 1 1
```

Sampling without replacement: Choose some lottery numbers which consist of 1 to 100.

```r
sample(1:100, size = 6, replace = FALSE)
```

```
[1] 61 83 50 74 34 35
```

Random generators of common probability distributions

`R` provides some functions which generate random numbers following famous distributions. Although we will learn some skills generating these numbers in basis levels, these functions do the same thing more elegantly.

```r
gg_curve(dbeta, from = 0, to = 1, args = list(shape1 = 3, shape2 = 2)) +
  geom_histogram(
    data = tibble(
      rand = rbeta(1000, 3, 2),
      idx = seq(0, 1, length.out = 1000)
    ),
    aes(x = rand, y = ..density..),
    position = "identity",
    bins = 30,
    alpha = .45,
    fill = gg_hcl(1)
  )
```

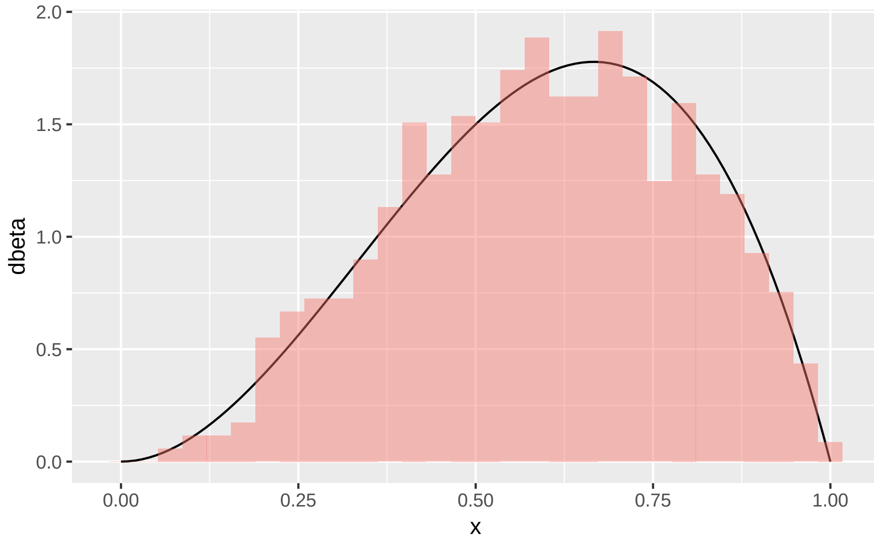


Figure 1: Beta(3,2) random numbers

Figure 1 shows that `rbeta()` function generate random numbers very well. Histogram is of the random number, and the curve is the true beta distribution.

Chapter 1

Methods for Generating Random Variables

1.1 Introduction

Most of the methods so-called *computational statistics* requires generation of random variables from specified probability distribution. In hand, we can spin wheels, roll a dice, or shuffle cards. The results are chosen randomly. However, we want the same things with computer. Here, `r`. As we know, computer cannot generate complete uniform random numbers. Instead, we generate **pseudo-random** numbers.

1.2 Pseudo-random Numbers

Definition 1.1 (Pseudo-random numbers)**.** Sequence of values generated deterministically which have all the appearances of being independent $unif(0,1)$ random variables, i.e.

$$x_1, x_2, \ldots, x_n \stackrel{iid}{\sim} unif(0,1)$$

- behave *as if* following $unif(0,1)$
- typically generated from an *initial seed*

1.2.1 Linear congruential generator

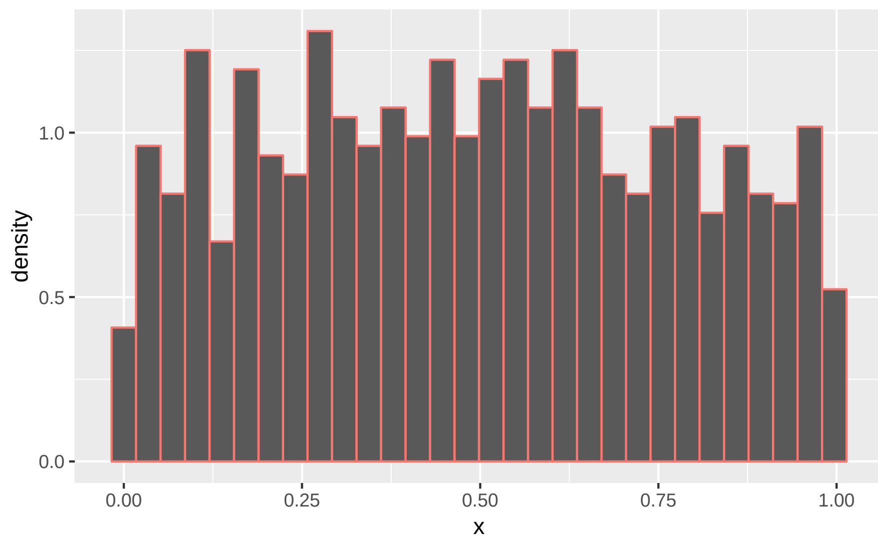
Let $x_0, x_1, \ldots \in \mathbb{Z}_+$.

1. Set x_0 as initial seed.
2. Generate $x_n, n = 1, 2, \ldots$ recursively:
   a. $x_n = (ax_{n-1} + c) \mod m$
   b. where $a, c \in \mathbb{Z}_+, m :$ modulus
3. Compute $u_n = \frac{x_n}{m} \in (0,1)$

Then $u_1, u_2, \ldots \sim unif(0,1)$

```
lcg <- function(n, seed, a, b, m) {
  x <- rep(seed, n + 1)
  for (i in 1:n) {
    x[i + 1] <- (a * x[i] + b) %% m
  }
  x[-1] / m
}
```

```
tibble(
  x = lcg(1000, 0, 1664525, 1013904223, 2^32)
) %>%
  ggplot(aes(x = x)) +
  geom_histogram(aes(y = ..density..), bins = 30, col = gg_hcl(1))
```



1.2.2 Multiplicative congruential generator

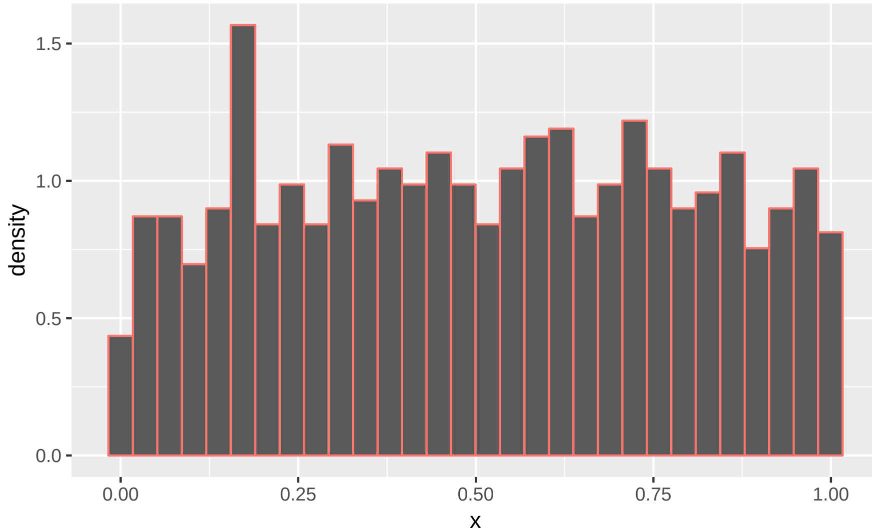
As we can expect from its name, this is congruential generator with $c = 0$.

1. Set x_0 as initial seed.
2. Generate $x_n, n = 1, 2, \ldots$ recursively:
    a. $x_n = a x_{n-1} \mod m$
    b. where $a \in \mathbb{Z}_+, m$: modulus
3. Compute $u_n = \frac{x_n}{m} \in (0, 1)$

Then $u_1, u_2, \ldots \sim unif(0, 1)$

We just set `b = 0` in our `lcg()` function. The **seed must not be zero**.

```
tibble(
  x = lcg(1000, 5, 1664525, 0, 2^32)
) %>%
  ggplot(aes(x = x)) +
  geom_histogram(aes(y = ..density..), bins = 30, col = gg_hcl(1))
```



1.2.3 Cycle

Generate LCG $n = 32$ with $a = 1$, $c = 1$, and $m = 16$ from the seed $x_0 = 0$.

```
lcg(32, 0, 1, 1, 16)
```

```
 [1] 0.0625 0.1250 0.1875 0.2500 0.3125 0.3750 0.4375 0.5000 0.5625 0.6250
[11] 0.6875 0.7500 0.8125 0.8750 0.9375 0.0000 0.0625 0.1250 0.1875 0.2500
[21] 0.3125 0.3750 0.4375 0.5000 0.5625 0.6250 0.6875 0.7500 0.8125 0.8750
[31] 0.9375 0.0000
```

Observe that we have the cycle after m-th number. Against this problem, we give different seed from every $(im + 1)$th random number.

1.3 The Inverse Transform Method

Definition 1.2 (Inverse of CDF)**.** Since some cdf F_X is not strictly increasing, we difine $F_X^{-1}(y)$ for $0 < y < 1$ by

$$F_X^{-1}(y) := inf\{x : F_X(x) \geq y\}$$

Using this definition, we can get the following theorem.
Theorem 1.1 (Probability Integral Transformation)**.** *If X is a continuous random variable with cdf $F_(x)$, then*

$$U \equiv F_X(X) \sim unif(0, 1)$$

Probability Integral Transformation. Let $U \sim unif(0, 1)$. Then

$$\begin{aligned} P(F_X^{-1}(U) \leq x) &= P(\inf\{t : F_X(t) = U\} \leq x) \\ &= P(U \leq F_X(x)) \\ &= F_U(F_X(x)) \\ &= F_X(x) \end{aligned}$$

□

Thus, to generate n random variables $\sim F_X$,

1. form of $F_X^{-1}(u)$
2. For each $i = 1, 2, \ldots, n$:
   a. Generate $u_i \sim unif(0, 1)$
   b. $x_i = F_X^{-1}(u_i)$

Collect $x_1, x_2, \ldots, x_n \overset{iid}{\sim} F_X$.

1.3.1 Continuous case

Denote that the *probability integral transformation* holds for a continuous variable. When generating continuous random variable, applying above algorithm might work.

Example 1.1 (Exponential distribution)**.** If $X \sim Exp(\lambda)$, then $F_X(x) = 1 - e^{-\lambda x}$. We can derive the inverse function of cdf

$$F_X^{-1}(u) = \frac{1}{\lambda} \ln(1 - u)$$

Note that

$$U \sim unif(0, 1) \Leftrightarrow 1 - U \sim unif(0, 1)$$

Then we just can use U instead of $1 - U$.

```
inv_exp <- function(n, lambda) {
  -log(runif(n)) / lambda
}
```

If we generate $x_1, \ldots, x_{500} \sim Exp(\lambda = 1)$,

```
gg_curve(dexp, from = 0, to = 10) +
  geom_histogram(
    data = tibble(x = inv_exp(500, lambda = 1)),
    aes(x = x, y = ..density..),
    bins = 30,
    fill = gg_hcl(1),
    alpha = .5
  )
```

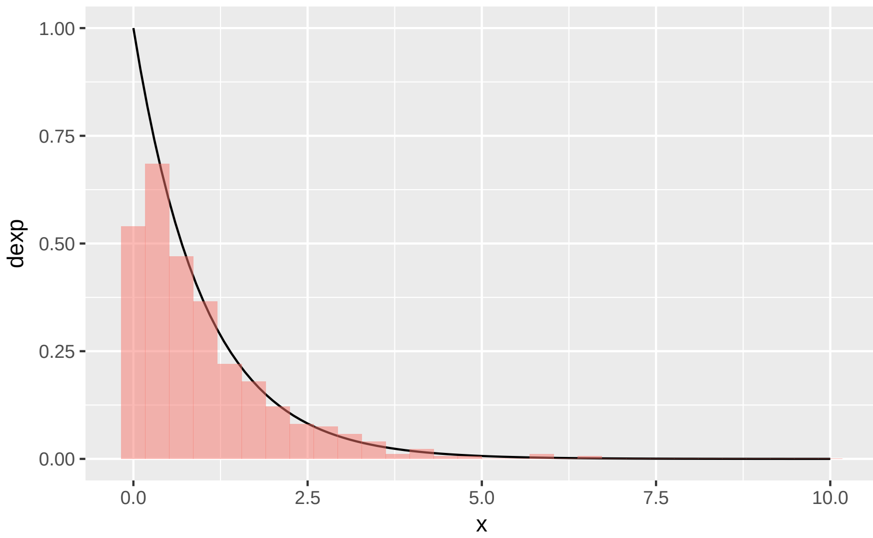


Figure 1.1: Inverse Transformation: Exp(1)

1.3.2 Discrete case

1. For each $i = 1, 2, \ldots, n$:
    a. Generate $u_i \sim unif(0, 1)$
    b. Take x_i s.t. $F_X(x_{i-1}) < U \leq F_X(x_i)$

Collect $x_1, x_2, \ldots, x_n \sim F_X$.

```
pmf <-
  tibble(
    x = 0:4,
    p = c(.1, .2, .2, .2, .3)
  )
```

Table 1.1: Example of a Discrete Random Variable

x	0.0	1.0	2.0	3.0	4.0
p	0.1	0.2	0.2	0.2	0.3

Example 1.2 (Discrete Random Variable)**.** Consider a discrete random variable X with a mass function as in Table 1.1.

i.e.

```
pmf %>%
  ggplot() +
  geom_segment(aes(x = x, y = 0, xend = x, yend = p)) +
  ylab(expression(p(x)))
```

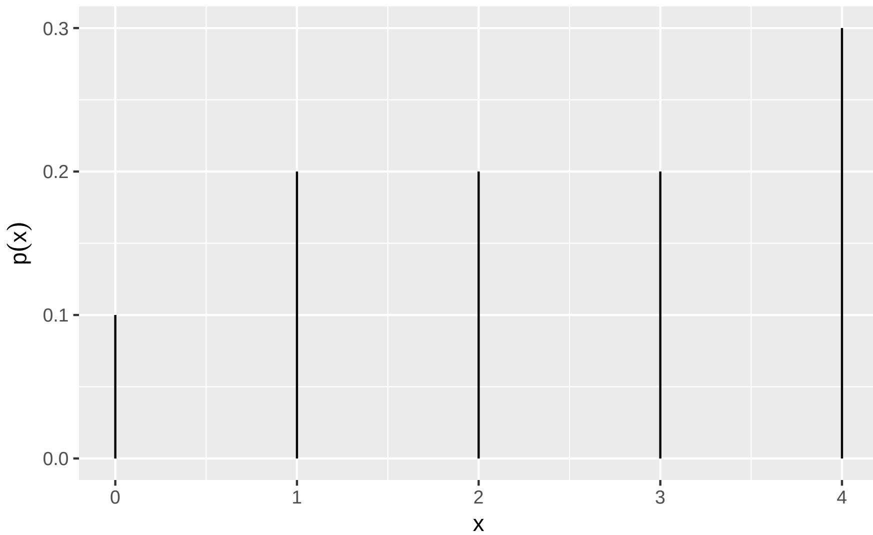


Figure 1.2: Probability Mass Function

Then we have the cdf

```
pmf %>%
  mutate(
    fx = cumsum(p),
    x_end = lead(x, default = 5),
    u = fx,
    u = ifelse(u == .5, .6, u),
    fx1 = lead(fx, default = 1),
    rand = u > fx & u <= fx1
  ) %>%
  ggplot() +
  geom_segment(aes(x = x, y = fx, xend = x_end, yend = fx)) +
  ylab(expression(F(x))) +
  geom_segment(
    aes(x = 0, y = u, xend = x_end, yend = u, colour = rand),
    linetype = "dashed",
    arrow = arrow(length = unit(.5, "cm")),
    show.legend = FALSE
  ) +
  geom_segment(
    aes(x = x_end, y = u, xend = x_end, yend = 0, colour = rand),
    linetype = "dashed",
    arrow = arrow(length = unit(.5, "cm")),
    show.legend = FALSE
  ) +
  scale_colour_manual(
    values = c("TRUE" = gg_hcl(1), "FALSE" = "#00000000")
  )
```

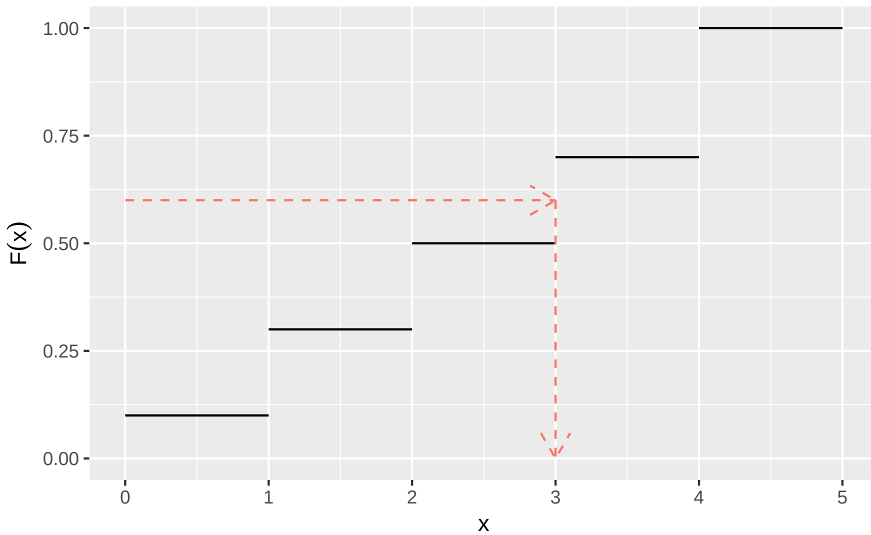


Figure 1.3: CDF of the Discrete Random Variable: Illustration for discrete case

Remembering the algorithm, we can implement `dplyr::case_when()` here.

```
rcustom <- function(n) {
  tibble(u = runif(n)) %>%
    mutate(
      x = case_when(
        u > 0 & u <= .1 ~ 0,
        u > .1 & u <= .3 ~ 1,
        u > .3 & u <= .5 ~ 2,
        u > .5 & u <= .7 ~ 3,
        TRUE ~ 4
      )
    ) %>%
    select(x) %>%
    pull()
}
```

```
tibble(
  x = rcustom(100)
) %>%
  ggplot(aes(x = x)) +
  geom_histogram(aes(y = ..ndensity..), binwidth = .1)
```

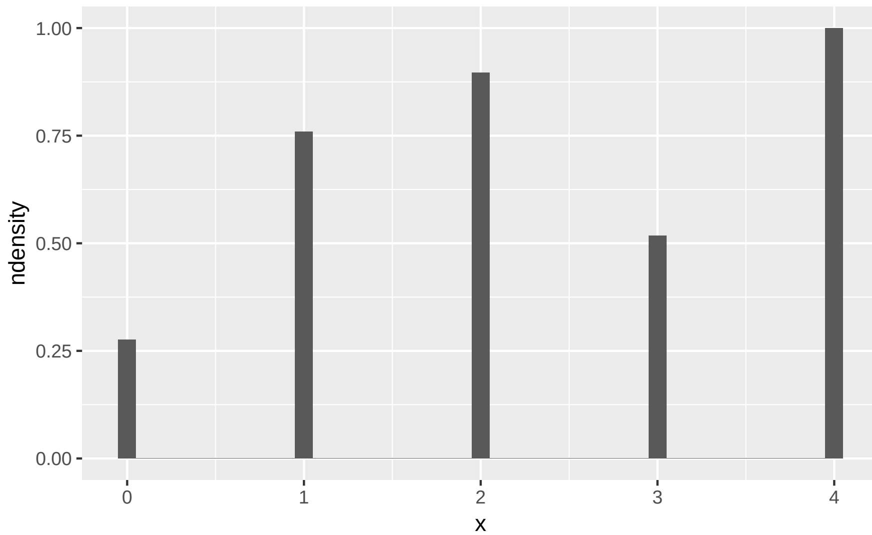


Figure 1.4: Generated discrete random numbers

See Figure 1.2 and 1.4. Comparing the two, the result can be said okay.

1.3.3 Problems with inverse transformation

Examples 1.1 and 1.2. We could generate these random numbers because we aware of

1. analytical F_X
2. F^{-1}

In practice, however, not all distribution have analytical F. Numerical computing might be possible, but it is not efficient. There are other approaches.

1.4 The Acceptance-Rejection Method

Acceptance-rejection method does not require analytical form of cdf. What we need is our *target* density (or mass) function and *proposal* density (or mass) function. Target function is what we want to generate. Propsal function is of any random variable that is *easy to generate random numbers*. From this approach, we can generate any distribution while computation is not efficient.

pdf or pmf	target or proposal
f	target
g	proposal - easy to generate random numbers

First of all, g should satisfy that

$$spt f \subseteq spt g$$

Next, for some (pre-specified) $c > 0$

$$\forall x \in spt f : \frac{f(x)}{g(x)} \le c$$

1.4.1 A-R algorithm

For $i = 1, \ldots, n$

1. $Y \sim g(Y)$
2. $U \sim unif(0, 1) \perp\!\!\!\perp Y$
3. Accept-Reject step
    a. Accept: $U \le \frac{f(Y)}{cg(Y)} \Rightarrow x_i = Y$
    b. Reject: otherwise, go to step 1

Collect $x_1, x_2, \ldots, x_n \overset{iid}{\sim} f(x)$.

1.4.2 Efficiency

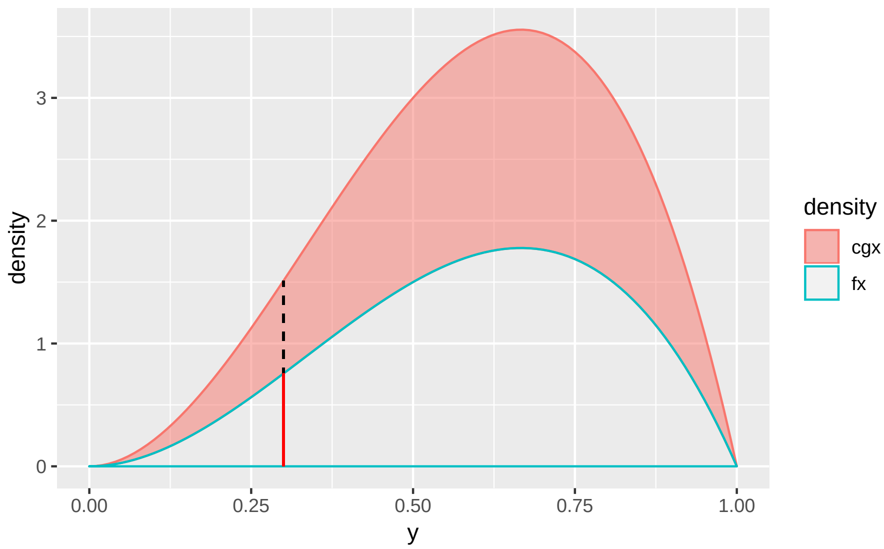


Figure 1.5: Property of AR method

See Figure 1.5. This illustrates the motivation of A-R method. Lower one is $f(x)$ and the upper one is $cg(x)$ which covers f. The algorithm takes random number from $Y \sim g$ in each recursive step i, which is represented as a line in the figure. We can see that

$$0 < \frac{f(x)}{cg(x)} \le 1$$

Chapter 2

Monte Carlo Integration and Variance Reduction