Marek Gągolewski
Konstancja Bobecka-Wesołowska
Przemysław Grzegorzewski

Computer Statistics with R

3. Probability Distributions and Simulation Basics

Faculty of Mathematics and Information Science
Warsaw University of Technology
[Last update: December 9, 2012]

Contents

Info

These tutorials are likely to contain bugs and typos. In case you find any don't hesitate to *contact us*! Thanks in advance!

3.1. Preliminaries

3.1.1. Basic probability distributions

R has a built-in support for calculating e.g. the values of functions related to the following well-known probability distributions:

Distribution	Name	Parameters	Identifier
$\mathrm{Bin}(n,p)$	Binomial	$n \in \mathbb{N}$, $p \in (0,1)$	`*binom`
$\mathrm{Geom}(p)$	Geometric	$p \in (0,1)$	`*geom`
$\mathrm{Hyp}(m,n,k)$	Hypergeometric	$m,n,k \in \mathbb{N}$, $k \leq m$	`*hyper`
$\mathrm{NegBin}(n,p)$	Negative Binomial	$n \in \mathbb{N}$, $p \in (0,1)$	`*nbinom`
$\mathrm{Poi}(\lambda)$	Poisson	$\lambda > 0$	`*pois`
$\mathrm{B}(a,b)$	Beta	$a > 0$, $b > 0$	`*beta`
$\mathrm{C}(l=0,s=1)$	Cauchy	$l \in \mathbb{R}$, $s > 0$	`*cauchy`
$\chi^2{}_d$	Chi-square	$d \in \mathbb{N}$	`*chisq`
$\mathrm{Exp}(\lambda=1)$	Exponential	$\lambda > 0$	`*exp`
$\mathrm{F}^{[d_1,d_2]}$	Snedecor's F	$d_1, d_2 \in \mathbb{N}$	`*f`
$\Gamma(a,s)$	Gamma	$a > 0$, $s > 0$	`*gamma`
$\mathrm{Logis}(\mu=0,s=1)$	Logistic	$\mu \in \mathbb{R}$, $s > 0$	`*logis`
$\mathrm{LogN}(\mu=0,\sigma=1)$	Log-normal	$\mu \in \mathbb{R}$, $\sigma > 0$	`*lnorm`
$\mathrm{N}(\mu=0,\sigma=1)$	Normal	$\mu \in \mathbb{R}$, $\sigma > 0$	`*norm`
$\mathrm{U}(a=0,b=1)$	Uniform	$a < b$	`*unif`
$\mathrm{t}^{[d]}$	Student's t	$d \in \mathbb{N}$	`*t`
$\mathrm{Wei}(a,s=1)$	Weibull	$a > 0$, $s > 0$	`*weibull`

The function prefix, `*`, may be one of the following:

Prefix	Meaning
`d`	density (PDF) $f(x)$ or probability mass function (PMF) $\mathrm{P}(X=x)$
`p`	cumulative probability distribution function (CDF) $F(x) = \mathrm{P}(X \leq x)$
`q`	quantile function $\simeq F^{-1}(p)$
`r`	generation of `r`andom deviates

where X is a random variable.

Info

For convenience, some distributions have default parameters (see the *Distribution* column). For example, `pnorm(3)` is the same as `pnorm(3,0,1)`, i.e. the value of the CDF of the $\mathrm{N}(0,1)$ (standardized normal) distribution at 3.

3.1.1.1. Cumulative distribution function

The value of the CDF, $F(x)$, of a chosen probability distribution may be calculated by choosing the prefix `p`, e.g.

```
pnorm(0)  # CDF of the standard normal distribution at 0
## [1] 0.5
pnorm(c(1, 2, 3))  # CDF of the standard normal distribution at 1,2, and 3
## [1] 0.8413 0.9772 0.9987
```

Further function arguments determine parameters of the distribution, e.g.:

```
pnorm(0, 2, 1)  # CDF of the N(2,1) distribution at 0
## [1] 0.02275
ppois(10, 3)  # CDF of the Poi(3) distribution at 10
## [1] 0.9997
```

Also, the so-called *survival function*, defined as $S(x) = 1 - F(x) = \mathrm{P}(X > x)$, may be computed by using the `lower.tail=F` parameter:

```
pnorm(0.2, lower.tail = F)  # survival fun. of the std. normal distrib. at 0
## [1] 0.4207
```

Obviously, the above is equivalent to:

```
1 - pnorm(0.2)
## [1] 0.4207
```

3.1.1.2. Density function

The prefix `d` preceding the distribution identifier stands for a *probability density function* (in case of continuous random variables) or a *probability mass function* (in case of discrete distributions), e.g.:

```
dexp(0)  # the value of f(0), where f is the PDF of Exp(1)
## [1] 1
dexp(c(0, 0.5, 1), 0.5)  # f(0), f(0.5), f(1) for Exp(0.5)
## [1] 0.5000 0.3894 0.3033
pr <- dbinom(0:8, 8, 0.25)  # Pr(X=i) for X~Bin(8, 1/4), i=0,1,...,8
round(pr, 3)  # print the results rounded to 3 decimal places
## [1] 0.100 0.267 0.311 0.208 0.087 0.023 0.004 0.000 0.000
```

3.1.1.3. Quantile function

Theoretical quantiles may be calculated using the `q` prefix. The first argument of each such function is the quantile order, e.g.

```
qt(0.95, 5)  # 0.95-quantile of the t distribution with 5 degrees of freedom
## [1] 2.015
qt(0.95, c(1, 5, 10, 15))  # many degrees of freedom at a time
## [1] 6.314 2.015 1.812 1.753
qt(0.95, Inf)  # the standard normal distribution
## [1] 1.645
qnorm(0.95)
## [1] 1.645
qt(0.95, 1)  # the standard Cauchy distribution
## [1] 6.314
qcauchy(0.95)
## [1] 6.314

qt(c(0.95, 0.975, 0.99, 0.995), 5)
## [1] 2.015 2.571 3.365 4.032
qt(c(0.95, 0.975, 0.99, 0.995), c(1, 5, 10, 15))  # and what is that?
## [1] 6.314 2.571 2.764 2.947
```

If the selected probability distribution of a random variable X is not continuous, then the quantile function at q returns the smallest number $x \in \mathrm{supp}(X)$, for which $\mathrm{P}(X \leq x) \geq q$, where $\mathrm{supp}(X)$ is the support of X.

```
qbinom(c(0.4, 0.5, 0.6), 5, 0.5)
## [1] 2 2 3
pbinom(0:5, 5, 0.5)  # (for comparison)
## [1] 0.03125 0.18750 0.50000 0.81250 0.96875 1.00000
```

3.1.1.4. Generation of random deviates

The prefix `r` stands for a procedure for generation of (pseudo[1]-)random numbers. The desired number of observations to be generated should be passed as the first function argument, e.g.:

```
runif(5)  # 5 random observations from the uniform distribution on [0,1]
## [1] 0.93595 0.05763 0.71548 0.24401 0.62898
runif(10, 0, 5)  # 10 random deviates from U([0,5])
##  [1] 0.6642 1.0883 3.6624 1.4793 0.3366 3.1328 4.7619 4.9935 0.2220 3.0148
rpois(20, 4)
##  [1] 4 7 3 5 8 6 4 2 5 7 5 5 9 3 3 3 1 5 4 4
```

Many useful information on R-built-in pseudo-random number generators may be found in the manual, see `?set.seed`.

It is worth noting that a generator may be initialized with a given seed by using the `set.seed()` function. This leads to repeatable results, which may be sometimes desirable. By default, the seed is current-time based and hence the generated deviates appear as "random".

3.1.2. Sampling with and without replacement

To take a random sample (without replacement) of specified size n from a set S, we call `sample(`S`, `n`)`. Sampling with replacement may be done by using additional `replace=TRUE` parameter.

For example, $n = 15$ coin tosses may be simulated by calling:

```
sample(c("H", "T"), 15, replace = TRUE)
##  [1] "H" "H" "H" "T" "H" "T" "H" "T" "H" "H" "H" "T" "H" "T" "H"
```

The parameter n may be omitted — then we get a random permutation of a given set, e.g.:

```
sample(1:10)
##  [1]  1  7  5  6 10  8  4  9  2  3
```

[1] *Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number — there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method* (John von Neumann, 1951). However, such numbers *behave* just as they were random (with respect to several testable criteria). The reader interested in algorithmic pseudo-random number generators is referred to [1; 2].

3.1.3. ⋆ Special functions

3.1.3.1. ⋆ Gamma function

The *gamma function* was first defined by Legendre as

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t}\, dt, \tag{3.1}$$

for $x > 0$.

Here are some of its basic properties.

1. $\Gamma(1) = 1$,
2. $\Gamma(x+1) = x\Gamma(x)$,
3. $n \in \mathbb{N} \Rightarrow \Gamma(n) = (n-1)!$,
4. $\Gamma(x) = \int_0^1 \left(\ln \frac{1}{t}\right)^{x-1} dt$.

The Γ function is available in R as `gamma()`.

3.1.3.2. ⋆ Euler beta function

The Euler *beta function* is given by:

$$B(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1}\, dt \tag{3.2}$$

for $x > 0$ and $y > 0$.

It may be shown that the following properties hold.

1. $B(x,y) = B(y,x)$,
2. $B(x,y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$,
3. $\binom{n}{k} = \frac{1}{(n+1)B(n-k+1,k+1)}$.

The values of B may be calculated in R by means of the `beta()` function.

3.1.3.3. ⋆ Incomplete and regularized beta functions

The *incomplete beta function* is a generalization of the B function:

$$B_i(u,x,y) = \int_0^u t^{x-1}(1-t)^{y-1}\, dt \tag{3.3}$$

for $x > 0, y > 0, u \in [0,1]$.

Obviously, $B_i(1,x,y) = B(x,y)$.

The *regularized beta function* is defined as:

$$I(u,x,y) = \frac{B_i(u,x,y)}{B(x,y)} \tag{3.4}$$

for $x > 0, y > 0$ and $u \in [0,1]$.

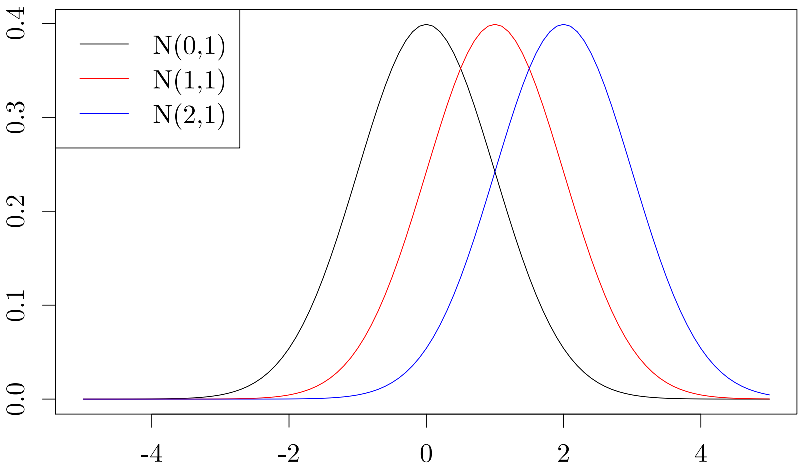
It is easily seen that $I(u,x,y)$ is equivalent to the value of the CDF of the beta $B(x,y)$ distribution at u. Therefore, it may be calculated with the `pbeta()` function.

3.2. Examples

Ex. 3.1. Draw the PDF and the CDF of the following distributions: a) $N(0, 1)$, b) $N(1, 1)$, c) $N(2, 1)$.

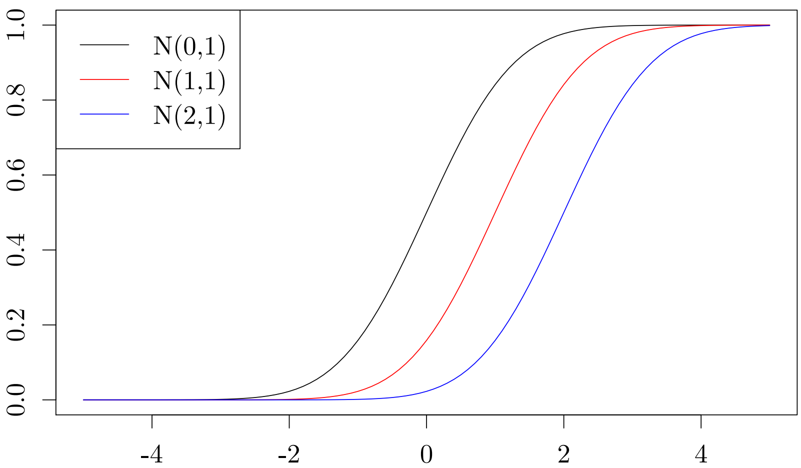
Solution.
Let us plot the probability density functions for the normal distributions with different location parameters:

```
x <- seq(-5, 5, by = 0.1)
plot(x, dnorm(x), type = "l", col = 1, ylab = "", main = "")
lines(x, dnorm(x, 1, 1), col = 2)  # adds another curve
lines(x, dnorm(x, 2, 1), col = 4)  # and another one
legend("topleft", c("N(0,1)", "N(1,1)", "N(2,1)"), col = c(1, 2, 4), lty = 1)
```



The plots of the CDFs may be created in a similar way:

```
x <- seq(-5, 5, by = 0.1)
plot(x, pnorm(x), col = 1, main = "", ylab = "", type = "l")
lines(x, pnorm(x, 1, 1), col = 2)
lines(x, pnorm(x, 2, 1), col = 4)
legend("topleft", c("N(0,1)", "N(1,1)", "N(2,1)"), col = c(1, 2, 4), lty = 1)
```



⊡

Ex. 3.2. The height of a group of people is described by the normal distribution with expectation of 173 cm and standard deviation of 6 cm.

1. Calculate the probability that the height of a randomly selected person is less than or equal to 179 cm.
2. Calculate the fraction of people of height between 167 and 180 cm.
3. What is the probability that a person's height is not less than 181 cm?
4. Calculate the height value not exceeded by 60% of the population.

Solution.
The height of a randomly selected person is described by a random variable $X \sim \mathrm{N}(173, 6)$.

Firstly, we are interested in calculating $\mathrm{P}(X \leq 179)$:

```
pnorm(179, 173, 6)
## [1] 0.8413
```

Next we determine $\mathrm{P}(167 \leq X \leq 180)$. However, as X is a continuous random variable, it holds $\mathrm{P}(X = 167) = 0$. Thus, it suffices to calculate $\mathrm{P}(167 < X \leq 180)$:

```
pnorm(180, 173, 6) - pnorm(167, 173, 6)
## [1] 0.7197
```

The third question concerns $\mathrm{P}(X \geq 181) = \mathrm{P}(X > 181)$:

```
1 - pnorm(181, 173, 6)  # or equivalently:
## [1] 0.09121
pnorm(181, 173, 6, lower.tail = F)
## [1] 0.09121
```

Lastly, the $q_{0.6}$ quantile of the $\mathrm{N}(173, 6)$ distribution is equal to:

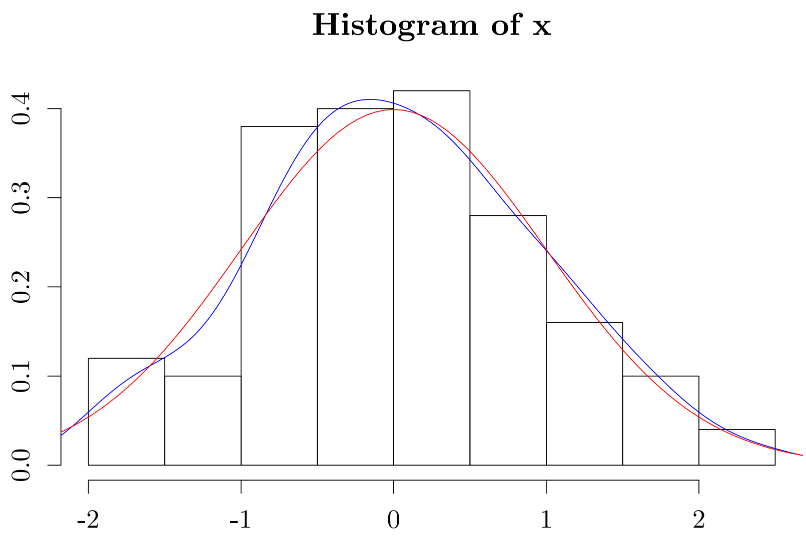
```
qnorm(0.6, 173, 6)
## [1] 174.5
```

⊡

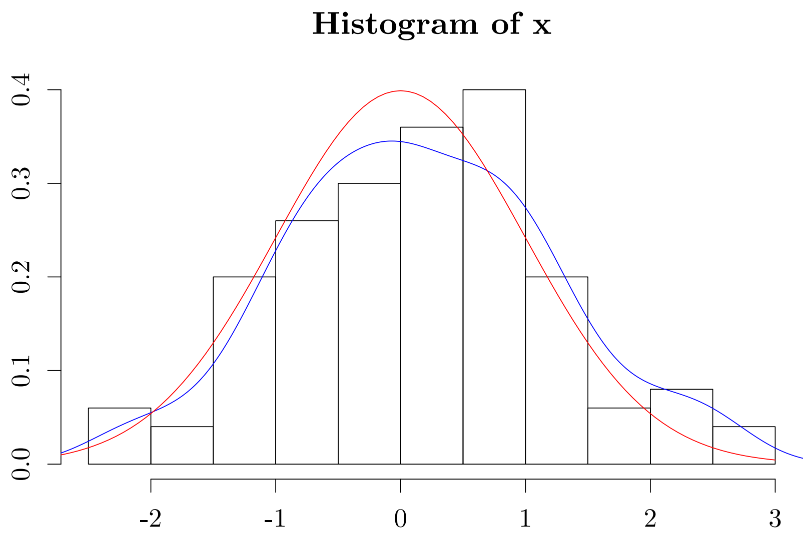
Ex. 3.3. Generate $n = 100$ random deviates from the standard normal distribution. Draw a histogram, a kernel density estimator, and the theoretical density. Discuss the results.

Solution.
The solution to this exercise is quite simple:

```
n <- 100
x <- rnorm(n)  # n random deviates
hist(x, prob = T)
lines(density(x), col = "blue")
curve(dnorm(x), from = -3, to = 3, col = "red", add = T)
```



Obviously, another random sample will (almost surely) consist of different observations. Therefore, it is advised to examine the outputs of a few replications of the experiment (by calling the above code several times).



⊡

Ex. 3.4. Draw a plot of probability mass functions of the following binomial distributions: $\mathrm{Bin}(10, 0.25), \mathrm{Bin}(100, 0.25), \mathrm{Bin}(1000, 0.25)$.

Solution.
First we calculate $\mathrm{P}(X = k)$ for $k = 0, 1, \ldots, 10$ and $X \sim \mathrm{Bin}(10, 0.25)$:

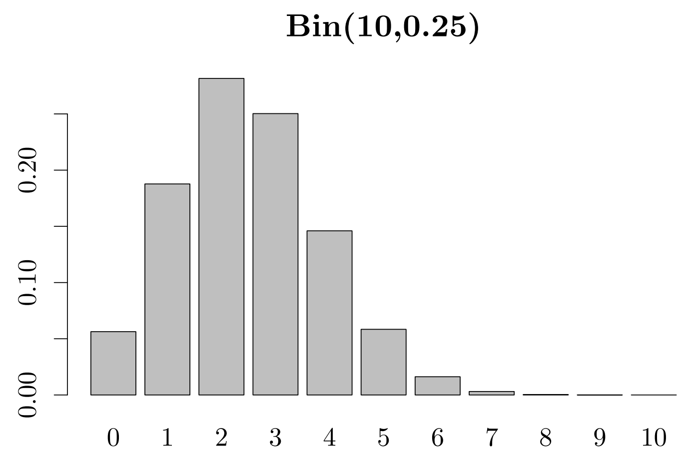
```
x <- dbinom(0:10, 10, 0.25)
```

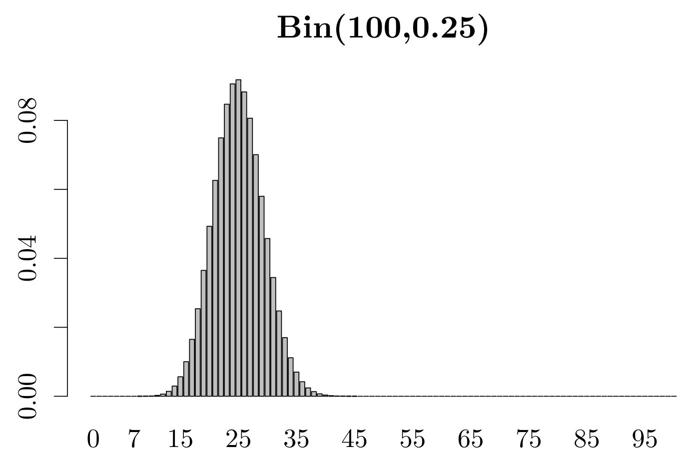
We perform similar calculations in case of the other distributions.

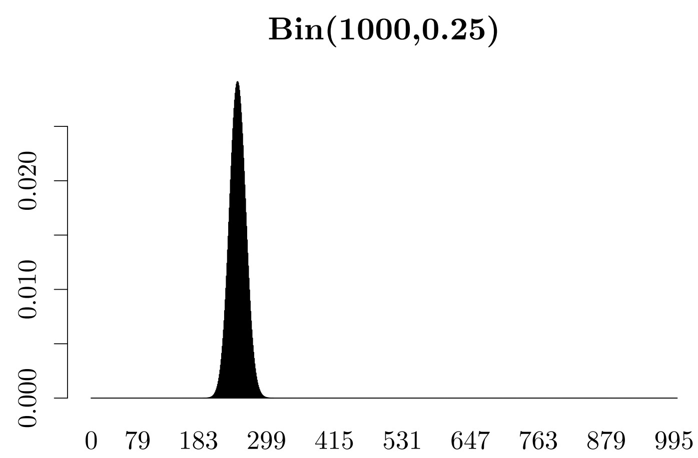
```
y <- dbinom(0:100, 100, 0.25)
z <- dbinom(0:1000, 1000, 0.25)
```

Let us draw the probability mass functions as bar plots.

```
barplot(x, names.arg = 0:10, main = "Bin(10,0.25)")
barplot(y, names.arg = 0:100, main = "Bin(100,0.25)")
barplot(z, names.arg = 0:1000, main = "Bin(1000,0.25)")
```


Bin(10,0.25)
0.00
0.10
0.20
0 1 2 3 4 5 6 7 8 9 10


Bin(100,0.25)
0.00
0.04
0.08
0 7 15 25 35 45 55 65 75 85 95


Bin(1000,0.25)
0.000
0.010
0.020
0 79 183 299 415 531 647 763 879 995

Task

Recall one of the Central Limit Theorems. What do these figures illustrate?

⊡

Ex. 3.5. Given a random number generator (RNG) from the uniform distribution on $(0, 1)$, generate random deviates form the Pareto distribution with parameter $a = 2$.

Solution.

Note

Theorem. Let F be the CDF of a continuous random variable X. Then $X = F^{-1}(U)$, where $U \sim \mathrm{U}(0, 1)$.

The described method is called *inverse transform sampling*. It allows for generating random deviates from many distributions by using the $\mathrm{U}(0, 1)$ random number generator. Inverse transform sampling

The PDF of a random variable X from the Pareto distribution with shape parameter $a \geq 0$ is defined as

$$f(x) = \frac{a}{x^{a+1}}, \tag{3.5}$$

for $x > 1$. The CDF is given by

$$F(x) = (1 - 1/x^a), \tag{3.6}$$

and hence

$$F^{-1}(u) = (1 - u)^{-1/a}. \tag{3.7}$$

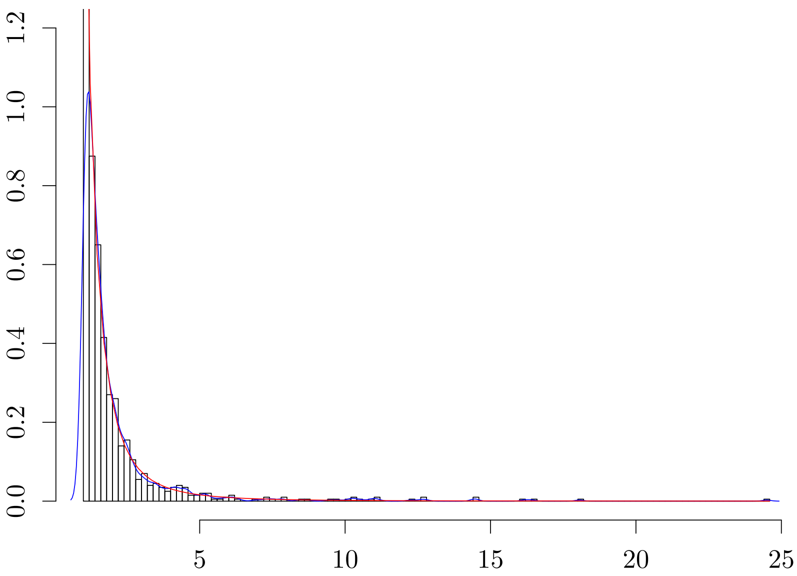
Therefore the random variable $F^{-1}(U) = (1 - U)^{-1/2}$, where $U \sim \mathrm{U}(0, 1)$, has the Pareto distribution with shape parameter $a = 2$.

The random sample may be generated as follows:

```
n <- 1000
u <- runif(n)
x <- (1 - u)^(-0.5)
# or: x <- u^(-0.5) # note: 1-u and u has the same distributions
```

Let us draw a histogram, a kernel density estimator, and the theoretical PDF:

```
hist(x, prob = T, main = NA, ylim = c(0, 1.2), breaks = 100)
lines(density(x), col = "blue")
curve(2/x^3, add = T, col = "red", from = 1)
```



⊡

Ex. 3.6. Calculate the area of $A = \{(x, y) \in \mathbb{R}^2 : 0 < x < 1; 0 < y < x^2\}$ using the Monte Carlo Integration method.

Solution.

Note

Monte Carlo Integration. Let $X_1, Y_1, X_2, Y_2, \ldots$ be independent random variables with the uniform distribution $U([0, 1])$. For a given continuous function $f : [0, 1] \to [0, 1]$ we define

$$Z_i = \mathbf{1}(Y_i \le f(X_i)), \tag{3.8}$$

where $\mathbf{1}(\cdot)$ is the indicator function. Then, from the strong law of large numbers, it almost surely holds

$$\lim_{n\to\infty} \frac{1}{n} \sum_{i=1}^{n} Z_i = \int_0^1 f(x)\, dx. \tag{3.9}$$

The method was proposed by a Polish mathematician Stanisław Ulam, who participated in the famous Manhattan Project.

Task

The generalization of this method for different (interval-based) domains and co-domains is left to the reader as an easy exercise.

The area of A is equal to

$$\int_A dx\, dy = \int_0^1 \left(\int_0^{x^2} dy \right) dx = \int_0^1 x^2\, dx = \frac{1}{3}.$$

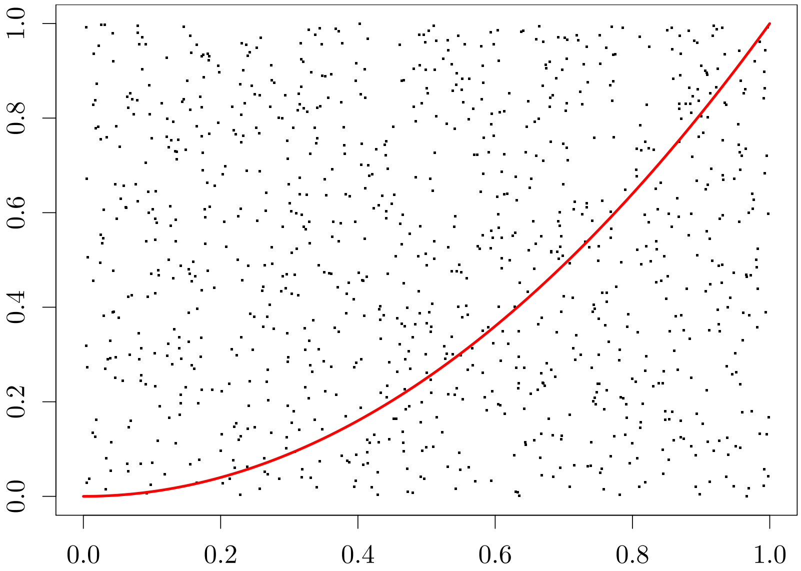
Let us calculate its approximate value by the Monte Carlo Integration method.

Firstly, we generate a random sample $(U_1, V_1, \ldots, U_n, V_n)$ from the uniform distribution:

```
n <- 1000  # the larger the number, the better the approximation
u <- runif(n)
v <- runif(n)
```

Let us plot the points $(U_1, V_1), \ldots, (U_n, V_n)$ and the function $y = x^2$, $x \in [0, 1]$.

```
plot(u, v, xlim = c(0, 1), ylim = c(0, 1), pch = ".")
curve(x * x, col = "red", type = "l", lwd = 3, add = T)
```



Then we count the number of points which fall below the graph of $y = x^2$:

```
z <- (v <= u * u)  # a logical vector
sum(z)  # recall that TRUE=1 and FALSE=0
## [1] 329
```

Therefore the area is approximately equal to:

```
mean(z)
## [1] 0.329
```

⊡

3.3. Conditional statements

Conditional statements allow us to branch an algorithm's control flow. They work in much the same way as their `C/C++` versions.

3.3.1. `if..else`

The syntax of the `if..else` statement is:

```
if (Condition)
{
    ... statements ...
}
```

or:

```
if (Condition)
{
    ... statements ...
} else {
    ... statements ...
}
```

Note

Note that the `else` keyword must be put in the same line as the `if`-block's closing brace — otherwise R's parser will not interpret it correctly.

```
a <- runif(1)
if (a < 0.5) print("less")
else print("more")  # ERROR: unexpected 'else'
```

```
a <- runif(1)
if (a < 0.5) print("less") else print("more")
## [1] "less"
```

3.3.2. `ifelse()` function

`ifelse` returns a vector of values chosen among two possibilities according to a given conditioning vector.

Consider the following example.

```
test <- (1:10)%%2 == TRUE
yes <- rep("yes", 10)
no <- rep("no", 10)
ret <- ifelse(test, yes, no)
ret
##  [1] "yes" "no"  "yes" "no"  "yes" "no"  "yes" "no"  "yes" "no"
```

Therefore, `ifelse` statement may be considered as a "vectorized" case of `if..else`. It is similar to the C's `?:` operator.

Here is an interesting illustration from the R manual:

```
x <- c(6:-4)
sqrt(x)  # gives warning
## Warning:  NaNs produced
##  [1] 2.449 2.236 2.000 1.732 1.414 1.000 0.000   NaN   NaN   NaN   NaN
sqrt(ifelse(x >= 0, x, NA))  # no warning
##  [1] 2.449 2.236 2.000 1.732 1.414 1.000 0.000    NA    NA    NA    NA
ifelse(x >= 0, sqrt(x), NA)  # warning - why?
## Warning:  NaNs produced
##  [1] 2.449 2.236 2.000 1.732 1.414 1.000 0.000    NA    NA    NA    NA
```

3.4. Loops

3.4.1. `for` loop

The `for` loop iterates through all elements of a vector. A loop variable is used to control the execution of a given code block.

The syntax is:

```
for (Variable in Vector)
{
    ... statements ...
}
```

`statements` are performed `length(Vector)` times. In each iteration of the loop, `Variable` is being assigned one of the consecutive values from `Vector`, that is: `Vector[1]`, `Vector[2]`, ... This is similar to the `foreach` loop in `C#`.

Example:

```
for (i in 1:5)           # for each i=1,2,3,4,5
{                        # do:
    print(i)             # print i
}                        # end
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

The brackets {·} may of course be omitted in case of only one statement to be iterated.

```
for (i in 1:5) print(2^i)
## [1] 2
## [1] 4
## [1] 8
## [1] 16
## [1] 32
```

3.4.2. `while` loop

Here is the syntax of the `while` loop:

```
while (Condition)
{
    ... statements ...
}
```

`commands` are executed until the `Condition` is false (while `Condition` is true).

Let us find the greatest power of 2 smaller than 100.

```
i <- 0
while (2^i < 100) {
    i <- i + 1
}
print(c(i, 2^i))  # Move back one step (why?)
## [1]   7 128
print(c(i - 1, 2^(i - 1)))
## [1]  6 64
```

Details

`break` breaks out of a loop of any type. The control is transferred to the first statement outside the currently executed loop.

`next` halts the processing of the current iteration and advances to the next.

Both `next` and `break` apply only to the inner-most loop in case of nested loops.

```
i <- 0
sumEven <- 0
while (i < 10) {
    i <- i + 1
    if (i%%2 == 1)
        next
    print(i)
    sumEven <- sumEven + i
}
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
print(sumEven)
## [1] 30
```

3.4.3. `repeat` loop

The syntax for the `repeat` loop is as follows.

```
repeat
{
    ... statements ...
}
```

`statements` are executed until we `break` out of the loop implicitly (with the `break` statement).

```
i <- 0
repeat {
    if (2^(i + 1) >= 100)
        break
    i <- i + 1
}
print(c(i, 2^i))
## [1]  6 64
```

3.4.4. A note on efficiency

In many applications, the use of loops in R is highly inefficient. We should use other solutions where possible.

Consider the following example:

```
v <- numeric(10)
for (i in 1:10) v[i] <- 2^i
v
```

```
## [1]    2    4    8   16   32   64  128  256  512 1024
```

We apply a vector (*sic!*) operator ^ 10 times — for each element of `v`. That is OK in imperative languages like C++. In R (a higher-level language), it would be better to express the above example using only *one* (optimized for speed) call to ^:

```
v <- 2^(1:10)
print(v)  # operations on vectors only
## [1]    2    4    8   16   32   64  128  256  512 1024
```

Does it really matter? One more example: we want to calculate a vector of numbers $a_1, \ldots, a_n$ where $a_i = \left(1 + \frac{1}{i}\right)^i$ (consecutive approximations to the number e).

Compare the run times (returned by `system.time()`) of the following expressions.

```
> n <- 1000000
> a1 <- numeric(n) # empty vector of size n
> system.time( { for (i in 1:n) a1[i] <- (1+1/i)^i } ) # using for loop
> system.time( { a2 <- (1+1/(1:n))^(1:n)           } ) # operations on vectors
```

The results were as follows (see the `user` column, which gives the real processing time in seconds[2]):

```
# using the for loop:
   user  system elapsed
  4.320   0.041   4.423
# operations on vectors:
   user  system elapsed
  0.172   0.006   0.180
```

However, some tasks, due their *iterative* nature, cannot be performed without explicit usage of looping statements.

3.4.5. `replicate()` function

The `replicate()` function is designed to perform e.g. some random experiment several times. It returns all results as a vector or a matrix.

It is very convenient and will be often used throughout our course.

Here is its syntax:

```
replicate(HowManyTimes,
{
    ... different tasks, e.g. sampling, arithmetic operations etc. ...
    return the result as a vector (also: a "single" number)
})
```

For example:

```
results <- replicate(50, {
    sample <- rnorm(10)  # a random sample from N(0,1) of size 10
    sd(sample)  # the result of the experiment
})
results
```

[2]The results were obtained on GNU/Linux 2.6.40.6-0.fc15.x86_64 SMP, model name : Intel(R) Core(TM) i5 CPU M 430 2.27GHz, cache size : 3072 KB, MemTotal: 4 GB.

```
##  [1] 0.8347 1.2021 0.7779 1.3208 1.0446 0.9824 0.9982 1.1797 1.2201 1.0453
## [11] 0.9548 0.4724 0.7428 0.5856 1.3047 1.0624 0.9653 1.2153 1.0636 0.9744
## [21] 0.8121 0.7267 1.1399 0.9268 0.9934 1.2465 1.0432 1.6088 1.0500 0.9551
## [31] 1.3184 0.6913 0.7772 0.9299 0.9923 1.2695 0.7269 0.8675 1.0583 1.3358
## [41] 1.3599 0.9471 0.9415 0.9958 0.8521 0.9697 0.5135 1.0761 0.7168 1.1381
mean(results)
## [1] 0.9985
```

Bibliography

[1] D.E. Knuth. *Sztuka programowania. Tom II. Algorytmy seminumeryczne.* WNT, 2002.

[2] R. Wieczorkowski and R. Zieliński. *Komputerowe generatory liczb losowych.* WNT, 1997.