3. Probability Distributions and Simulation Basics
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Preliminaries</td>
<td>1</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Basic probability distributions</td>
<td>1</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Sampling with and without replacement</td>
<td>3</td>
</tr>
<tr>
<td>3.1.3</td>
<td>* Special functions</td>
<td>4</td>
</tr>
<tr>
<td>3.2</td>
<td>Examples</td>
<td>5</td>
</tr>
<tr>
<td>3.3</td>
<td>Conditional statements</td>
<td>11</td>
</tr>
<tr>
<td>3.3.1</td>
<td>if..else</td>
<td>11</td>
</tr>
<tr>
<td>3.3.2</td>
<td>ifelse() function</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>Loops</td>
<td>13</td>
</tr>
<tr>
<td>3.4.1</td>
<td>for loop</td>
<td>13</td>
</tr>
<tr>
<td>3.4.2</td>
<td>while loop</td>
<td>13</td>
</tr>
<tr>
<td>3.4.3</td>
<td>repeat loop</td>
<td>14</td>
</tr>
<tr>
<td>3.4.4</td>
<td>A note on efficiency</td>
<td>14</td>
</tr>
<tr>
<td>3.4.5</td>
<td>replicate() function</td>
<td>15</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

---

**Info**

These tutorials are likely to contain bugs and typos. In case you find any don’t hesitate to [contact us](mailto:contact-us@tutorials.com). 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:

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Name</th>
<th>Parameters</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bin(n, p)</td>
<td>Binomial</td>
<td>(n \in \mathbb{N}, p \in (0, 1))</td>
<td>*binom</td>
</tr>
<tr>
<td>Geom(p)</td>
<td>Geometric</td>
<td>(p \in (0, 1))</td>
<td>*geom</td>
</tr>
<tr>
<td>Hyp(m, n, k)</td>
<td>Hypergeometric</td>
<td>(m, n, k \in \mathbb{N}, k \leq m)</td>
<td>*hyper</td>
</tr>
<tr>
<td>NegBin(n, p)</td>
<td>Negative Binomial</td>
<td>(n \in \mathbb{N}, p \in (0, 1))</td>
<td>*nbinom</td>
</tr>
<tr>
<td>Poi(\lambda)</td>
<td>Poisson</td>
<td>(\lambda &gt; 0)</td>
<td>*pois</td>
</tr>
<tr>
<td>B(a, b)</td>
<td>Beta</td>
<td>(a &gt; 0, b &gt; 0)</td>
<td>*beta</td>
</tr>
<tr>
<td>C(l = 0, s = 1)</td>
<td>Cauchy</td>
<td>(l \in \mathbb{R}, s &gt; 0)</td>
<td>*cauchy</td>
</tr>
<tr>
<td>X^2</td>
<td>Chi-square</td>
<td>(d \in \mathbb{N})</td>
<td>*chisq</td>
</tr>
<tr>
<td>Exp(\lambda = 1)</td>
<td>Exponential</td>
<td>(\lambda &gt; 0)</td>
<td>*exp</td>
</tr>
<tr>
<td>F(d1, d2)</td>
<td>Snedecor’s F</td>
<td>(d_1, d_2 \in \mathbb{N})</td>
<td>*f</td>
</tr>
<tr>
<td>Gamma(a, s)</td>
<td>Gamma</td>
<td>(a &gt; 0, s &gt; 0)</td>
<td>*gamma</td>
</tr>
<tr>
<td>Logis(\mu = 0, s = 1)</td>
<td>Logistic</td>
<td>(\mu \in \mathbb{R}, s &gt; 0)</td>
<td>*logis</td>
</tr>
<tr>
<td>LogN(\mu = 0, \sigma = 1)</td>
<td>Log-normal</td>
<td>(\mu \in \mathbb{R}, \sigma &gt; 0)</td>
<td>*lnorm</td>
</tr>
<tr>
<td>N(\mu = 0, \sigma = 1)</td>
<td>Normal</td>
<td>(\mu \in \mathbb{R}, \sigma &gt; 0)</td>
<td>*norm</td>
</tr>
<tr>
<td>U(a, b)</td>
<td>Uniform</td>
<td>(a &lt; b)</td>
<td>*unif</td>
</tr>
<tr>
<td>t(d)</td>
<td>Student’s t</td>
<td>(d \in \mathbb{N})</td>
<td>*t</td>
</tr>
<tr>
<td>Wei(a, s = 1)</td>
<td>Weibull</td>
<td>(a &gt; 0, s &gt; 0)</td>
<td>*weibull</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>density (PDF) (f(x)) or probability mass function (PMF) (P(X = x))</td>
</tr>
<tr>
<td>p</td>
<td>cumulative probability distribution function (CDF) (F(x) = P(X \leq x))</td>
</tr>
<tr>
<td>q</td>
<td>quantile function (\approx F^{-1}(p))</td>
</tr>
<tr>
<td>r</td>
<td>generation of random deviates</td>
</tr>
</tbody>
</table>

where \(X\) is a random variable.

For convenience, some distributions have default parameters (see the Distribution column). For example, \texttt{pnorm(3)} is the same as \texttt{pnorm(3,0,1)}, i.e. the value of the CDF of the \(N(0,1)\) (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 \texttt{p}, e.g.

\[
\texttt{pnorm(0)} \quad \# \text{ CDF of the standard normal distribution at 0}
\]

\[
\texttt{pnorm(c(1, 2, 3))} \quad \# \text{ CDF of the standard normal distribution at 1,2, and 3}
\]

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

\[ \text{ppnorm}(0, 2, 1) \]  # CDF of the N(2,1) distribution at 0
\[ \text{ppois}(10, 3) \]  # CDF of the Poi(3) distribution at 10

Also, the so-called \textit{survival function}, defined as \( S(x) = 1 - F(x) = \Pr(X > x) \), may be computed by using the \texttt{lower.tail=FALSE} parameter:

\[ \text{ppnorm}(0.2, \text{lower.tail = FALSE}) \]  # survival fun. of the std. normal distrib. at 0

Obviously, the above is equivalent to:

\[ 1 - \text{pnorm}(0.2) \]

3.1.1.2. Density function

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

\[ \text{dexp}(0) \]  # the value of \( f(0) \), where \( f \) is the PDF of Exp(1)
\[ \text{dexp}(0, 0.5, 1) \]  # \( f(0), f(0.5), f(1) \) for Exp(0.5)

\[ \text{pr} = \text{dbinom}(0:8, 8, 0.25) \]  # \( \Pr(X=i) \) for \( X \sim \text{Bin}(8, 1/4) \), \( i=0,1,...,8 \)
\[ \text{round(pr, 3)} \]  # print the results rounded to 3 decimal places

3.1.1.3. Quantile function

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

\[ \text{qt}(0.95, 5) \]  # 0.95-quantile of the t distribution with 5 degrees of freedom
\[ \text{qt}(0.95, 1, 5, 10, 15) \]  # many degrees of freedom at a time
\[ \text{qt}(0.95, \text{Inf}) \]  # the standard normal distribution
\[ \text{qnorm}(0.95) \]
\[ \text{qcauchy}(0.95) \]
\[ \text{qt}(c(0.95, 0.975, 0.99, 0.995), 5) \]  # and what is that?

Last update: December 9, 2012
3.1. PRELIMINARIES

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

\[
qbinom(c(0.4, 0.5, 0.6), 5, 0.5)
\]

## [1] 2 2 3

\[
pbinom(0:5, 5, 0.5) \quad \text{(for comparison)}
\]

## [1] 0.03125 0.18750 0.50000 0.81250 0.96875 1.00000

### 3.1.1.4. Generation of random deviates

The prefix \( \text{r} \) 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.:

\[
\text{runif}(5) \quad \text{# 5 random observations from the uniform distribution on } [0,1]
\]

## [1] 0.93595 0.06763 0.71548 0.24401 0.62898

\[
\text{runif}(10, 0, 5) \quad \text{# 10 random deviates from } U([0,5])
\]

## [1] 0.6642 1.0883 3.6624 1.4793 0.3366 3.1328 4.7619 4.9935 0.2220 3.0148

\[
\text{rpois}(20, 4)
\]

## [1] 4 7 3 5 8 6 4 2 5 7 5 9 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 \text{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 \text{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:

\[
\text{sample(c("H", "T"), 15, replace = TRUE)}
\]

## [1] "H" "H" "H" "T" "H" "T" "H" "H" "H" "H" "H" "T" "H" "H" "T"

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

\[
\text{sample(1:10)}
\]

## [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

\[ \Gamma(x) = \int_0^\infty t^{x-1}e^{-t} \, dt, \quad (3.1) \]

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 \( \Gamma \) function is available in \( \mathbb{R} \) as \texttt{gamma()}.

3.1.3.2. * Euler beta function

The *Euler beta function* is given by:

\[ B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1} \, dt \quad (3.2) \]

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. \( \binom{n}{k} = \frac{1}{\Gamma(n+1)\Gamma(n-k+1)\Gamma(k+1)} \).

The values of \( B \) may be calculated in \( \mathbb{R} \) by means of the \texttt{beta()} function.

3.1.3.3. * Incomplete and regularized beta functions

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

\[ B_i(u, x, y) = \int_0^u t^{x-1}(1-t)^{y-1} \, dt \quad (3.3) \]

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:

\[ I(u, x, y) = \frac{B_i(u, x, y)}{B(x, y)} \quad (3.4) \]

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 \texttt{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:

```r
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:

```r
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)
```

Last update: December 9, 2012
3.2. EXAMPLES

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 \( X \sim N(173, 6) \).

Firstly, we are interested in calculating \( P(X \leq 179) \):

\[
pnorm(179, 173, 6)
\]

## [1] 0.8413

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

\[
\text{pnorm}(180, 173, 6) - \text{pnorm}(167, 173, 6)
\]

## [1] 0.7197

The third question concerns \( P(X \geq 181) = P(X > 181) \):

\[
1 - \text{pnorm}(181, 173, 6) \quad \text{or equivalently:}
\]

## [1] 0.09121

\[
\text{pnorm}(181, 173, 6, \text{lower.tail} = \text{F})
\]

## [1] 0.09121

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

\[
\text{qnorm}(0.6, 173, 6)
\]

## [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:

\[
n <- 100
x <- \text{rnorm}(n) \quad \# \ n \ random \ deviates
\text{hist}(x, \text{prob} = \text{F})
\text{lines(density(x), \text{col} = "blue")}
\text{curve(dnorm(x), \text{from} = -3, \text{to} = 3, \text{col} = "red", \text{add} = \text{F})}
\]

Last update: December 9, 2012
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: Bin(10, 0.25), Bin(100, 0.25), Bin(1000, 0.25).

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

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

We perform similar calculations in case of the other distributions.

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

Let us draw the probability mass functions as bar plots.

```r
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)")
```
3.2. EXAMPLES

Bin(10,0.25)

Bin(100,0.25)

Bin(1000,0.25)

Last update: December 9, 2012
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.

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

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

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

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

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

$$F(x) = (1 - 1/x^a),$$

and hence

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

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

The random sample may be generated as follows:

```r
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:

```r
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

\[
Z_i = 1(Y_i \leq f(X_i)),
\]

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

\[
\lim_{n \to \infty} \frac{1}{n} \sum_{i=1}^{n} Z_i = \int_{0}^{1} f(x) \, dx.
\]

The method was proposed by a Polish mathematician Stanislaw 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:

```r
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 \(y = x^2, x \in [0,1]\).

```r
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 \(y = x^2\):

```r
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:

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

\[ \square \]

## 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 \texttt{if..else} statement is:

```r
if (Condition)
{
    ... statements ...
}
```
or:

\[
\text{if (Condition) }
\{ \\
\quad \ldots \text{ statements } \ldots \\
\} \text{ else } \{ \\
\quad \ldots \text{ statements } \ldots \\
\}
\]

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.

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

```r
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.

```r
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:

```r
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:

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

*statements* are performed \( \text{length(Vector)} \) 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:

```r
for (i in 1:5) # for each i=1,2,3,4,5
{
    print(i) # print i
}
```

```
## [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.

```r
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:

```r
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.

```r
i <- 0
while (2^i < 100) {
    i <- i + 1
}
```

```r
print(c(i, 2^i)) # Move back one step (why?)
```

```
## [1] 7 128
```

```r
print(c(i - 1, 2^(i - 1)))
```

```
## [1] 6 64
```
3.4. LOOPS

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.

```r
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.

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

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

```r
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:

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

Last update: December 9, 2012
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 \(^{10}\):

\[
v <- 2^{(1:10)}
\]

\[
\text{print}(v) \quad \# \text{operations on vectors only}
\]

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 \texttt{system.time()} ) of the following expressions.

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

The results were as follows (see the \texttt{user} column, which gives the real processing time in seconds\(^2\)):

\[
\begin{align*}
\text{# using the for loop:} & \\
& \text{user} \quad \text{system} \quad \text{elapsed} \\
& 4.320 \quad 0.041 \quad 4.423 \\
\text{# operations on vectors:} & \\
& \text{user} \quad \text{system} \quad \text{elapsed} \\
& 0.172 \quad 0.006 \quad 0.180
\end{align*}
\]

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

### 3.4.5. \texttt{replicate()} function

The \texttt{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:

\[
\texttt{replicate(HowManyTimes,} \quad \{ \ldots \text{different tasks, e.g. sampling, arithmetic operations etc.} \ldots \quad \text{return the result as a vector (also: a "single" number)} \}
\)

For example:

\[
\text{results} \leftarrow \text{replicate(50,} \quad \{ \text{ \texttt{sample} <- \text{rnorm(10)} \quad \# a random sample from } \text{N}(0,1) \text{ of size 10} \\
\text{ \texttt{sd(sample)}} \quad \# \text{the result of the experiment} \}
\)
\]

\[
\text{results}
\]

\(^2\)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.
Bibliography
