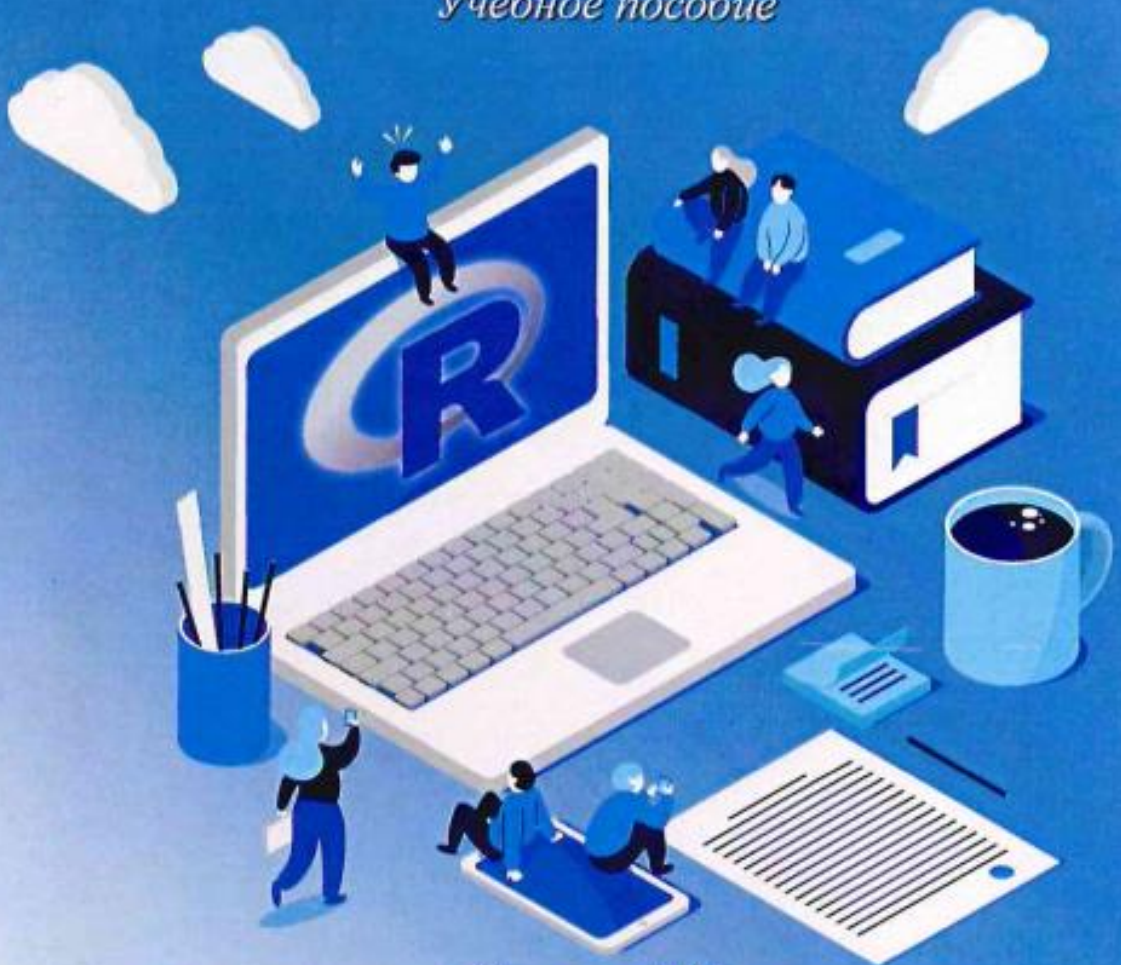
Министерство образования и науки Российской Федерации
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
УПРАВЛЕНИЯ

100 ЛЕТ

С.А. Суязова

ВВЕДЕНИЕ В ЯЗЫК СТАТИСТИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ R

Учебное пособие



Москва – 2018

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ УПРАВЛЕНИЯ»

Институт информационных систем

О д о б р е н о
Президиумом УМС ГУУ

С.А. Суязова

ВВЕДЕНИЕ В ЯЗЫК СТАТИСТИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ R

Учебное пособие

для подготовки бакалавров по направлению
01.03.02 Прикладная математика и информатика

Москва – 2018

УДК 004.432.42(075)
6Н1
С91

Ответственный редактор
заведующий кафедрой математических методов в экономике и управлении,
кандидат экономических наук, доцент
О.М. ПИСАРЕВА

Рецензенты
кандидат экономических наук, ведущий научный сотрудник
А.И. РЕЙ
(ФГБОУ ВО «Российская академия народного хозяйства и государственной службы при Президенте Российской Федерации»)

кандидат экономических наук, доцент
Е.А. СЕСЛАВИНА
(ФГБОУ ВО «Российский университет транспорта (МИИТ)»)

Суязова С.А.

С91 Введение в язык статистической обработки данных R [Текст] : учебное пособие для подготовки бакалавров по направлению 01.03.02 Прикладная математика и информатика / С.А. Суязова ; Государственный университет управления, Институт информационных систем ГУУ. – М. : Издательский дом ГУУ, 2018. – 65 с.

ISBN 978-5-215-03090-5

Учебное пособие «Введение в язык статистической обработки данных R» предназначено для студентов бакалавриата по направлению подготовки «Прикладная математика и информатика» (01.03.02) и содержит материалы по курсу дисциплины «Практикум на ЭВМ 4». Пособие призвано сформировать начальные навыки работы с языком статистической обработки данных R и поясняет принципы написания кода на этом языке на практических примерах. Рассматривается ряд вопросов, в том числе: синтаксис; работа с базовыми типами объектов: векторами, фреймами, списками; базовые графические функции; написание пользовательских функций. Описаны азы работы с графической оболочкой RStudio.

Практическое владение навыками программирования на R в дальнейшем необходимо для решения практических задач в рамках дисциплин «Эконометрика», «Математическое моделирование», а также дисциплины по выбору «Аналитический пакет R».

УДК 004.432.42(075)
6Н1

ISBN 978-5-215-03090-5

СОДЕРЖАНИЕ

Введение

Настоящее пособие адресовано студентам направления подготовки 01.03.02 Прикладная математика и информатика, которые только начинают изучать язык R. Рассматриваются базовые структуры данных: вектора, списки, матрицы, фреймы; даётся представление о базовой графике и описывается создание простых пользовательских функций. Эти вопросы достаточно полно освещены в русскоязычных учебниках по R [1, 3, 4], которые могут стать подспорьем для студентов в дальнейшем изучении языка. Цель данного пособия – помочь преодолеть первый барьер входа в R, который, как показывает практика, особенно высок, если до знакомства с R работа с данными велась лишь в аналитических пакетах с графическим интерфейсом. Изучение R, как и любого другого языка программирования – вопрос практики, и важно с первых шагов уяснить как он работает, чтобы видеть не только неудобства в отсутствии привычного интерфейса, но мощность и гибкость этого инструмента. У студентов, знакомых с другими языками обработки данных проблемы интерфейса возникнуть не должно, однако на основные принципы: векторизации, использовании функций, ограничения в использовании памяти – им всё же стоит обратить внимание.

Несколько разделов пособия посвящены стандартам оформления кода, источникам данных и понятию векторизации, особенно в той части, где векторизация заменяет циклы. Эти вопросы затронуты настолько, как это представляется нужным на начальном уровне владения R. Так, разметка скриптов с помощью RMarkdown, создание отчётов средствами библиотеки «knitr», парсинг данных из открытых источников не рассматриваются. Работа с продвинутыми графическими библиотеками, такими как «lattice» и «ggplot2», также остаётся за рамками данного пособия. Не затронуты таблицы данных («data.table») и их специальные возможности. К счастью, в последнее время литература по применению специализированных библиотек R в анализе данных появляется и на русском языке[1].

Язык программирования R – это один из наиболее популярных инструментов визуализации, статистического анализа и моделирования. В феврале 2017 года, по опросам специалистов, R был на пятом месте среди инструментов Data Science после SQL, Python, Java и Hadoop[2]. Владение этим инструментом – важное конкурентное преимущество для всех, кто намерен заниматься обработкой данных.

---

[1] Гарретт Гроулмунд, Хэдли Уикем Язык R в задачах науки о данных: импорт, подготовка, обработка, визуализация и моделирование данных. – Издательство «Вильямс», 2018 – 592 с.

[2] Robert A. Muenchen The Popularity of Data Science Software. URL: http://r4stats.com/articles/popularity/

Условные обозначения

В этом пособии можно встретить два варианта оформления кода R, и оба они выделены моноширинным шрифтом. Первый вариант подходит для знакомства с R и выглядит как команда, введённая в окно терминала: код начинается с символа закрывающей угловой скобки (>). Строки без угловой скобки показывают, что появится в качестве результата. Строка комментария всегда начинается с символа решётки (#).

```
> # сложить два числа
> 12 + 30
42
```

При вводе в терминал R длинной команды (более одной строки) каждая строка, начиная со второй, начинается со знака плюс. Однако при разборе многострочных конструкций гораздо удобнее представлять их не в режиме одной строки, а в виде скрипта. Это второй вариант оформления кода, и здесь при переходе команды на новую строку символ «+» не ставится, а символ «>» опускается. Дополнительные отступы, отражающие структуру кода, облегчают его чтение.

```
# как это выглядит в терминале
> dfTest <- data.frame(col1 = c(1, 2, 3, 4, 5),
+ col2 = c(5, 4, 3, 2, 1))
# как это выглядит в скрипте
dfTest <- data.frame(col1 = c(1, 2, 3, 4, 5),
                     col2 = c(5, 4, 3, 2, 1))
```

В формате скрипта результаты (текстовый вывод в терминале) будем выделять тройной решёткой, а неинформативные в примере строки заменять многоточием в угловых скобках (<…>). Скрипты ко всем главам и необходимые файлы данных находятся в свободном доступе по адресу: https://github.com/aksyuk/R-Practice-basics/tree/master/RScripts/manual_basics.

Код ниже демонстрирует версию R, в которой тестировались примеры этого руководства (ОС Windows 7).

```
# вывести на экран версию R, установленную на компьютере
R.version
### <…>
### version.string R version 3.5.1 (2018-07-02)
### nickname       Feather Spray
```

1. Знакомство с языком R и интерфейсом RStudio

Для работы нам понадобятся:

- дистрибутив R, который можно скачать с официального сайта проекта. Версию под Windows можно скачать по ссылке: https://cran.r-project.org/bin/windows/base/ (бесплатен, распространяется по свободной лицензии);
- дистрибутив RStudio с официального сайта: https://www.rstudio.com/products/rstudio/download/#download (версия Desktop бесплатна).

Далее под Windows достаточно запустить скачанные файлы и следовать диалогу установки. В этом диалоге можно оставить по умолчанию все настройки, кроме пути к папкам, в которые устанавливаются R и RStudio. Самый лучший вариант – устанавливать и то, и другое в корень диска, например: «C:\R» и «C:\RStudio»; при этом следует проверить, есть ли у пользователя право записи на этот диск. Для корректного взаимодействия R и RStudio в путях к ним не должно быть кириллицы, и чем короче будет путь к папкам программ, тем лучше.[3]

При запуске как чистого R, так и любой графической оболочки в консоль выводится версия языка, а также следующее приветствие:

```
R – это свободное ПО, и оно поставляется безо всяких гарантий. Вы
вольны распространять его при соблюдении некоторых условий.
Введите ‘license()’ для получения более подробной информации.

R – это проект, в котором сотрудничает множество разработчиков.
Введите ‘contributors()’ для получения дополнительной информации и
‘citation()’ для ознакомления с правилами упоминания R и его
пакетов в публикациях.

Введите ‘demo()’ для запуска демонстрационных программ, ‘help()’ –
для получения справки, ‘help.start()’ – для доступа к справке
через браузер. Введите ‘q()’, чтобы выйти из R.
```

Сам R не имеет графического интерфейса. Код вводится построчно в командную строку либо исполняется из скриптов. Текстовые результаты по умолчанию выводятся в одно терминала, графические – в отдельное окно, справка – в браузер. Существует довольно много графических оболочек,

[3] Подробная инструкция по установке R и RStudio под Windows, macos и linux доступна на гитхабе Б.Б. Демешева: https://bdemeshev.github.io/installation/r/R_installation.html

призванных облегчить жизнь пользователя R, и одна из популярных – Rstudio[4]. На рисунке **1** показан вид окна RStudio сразу после запуска. В левой верхней части находится редактор скриптов, а слева внизу – терминал. Код можно выполнять в терминале построчно, но удобнее запускать на исполнение строки из скрипта, или весь скрипт целиком.

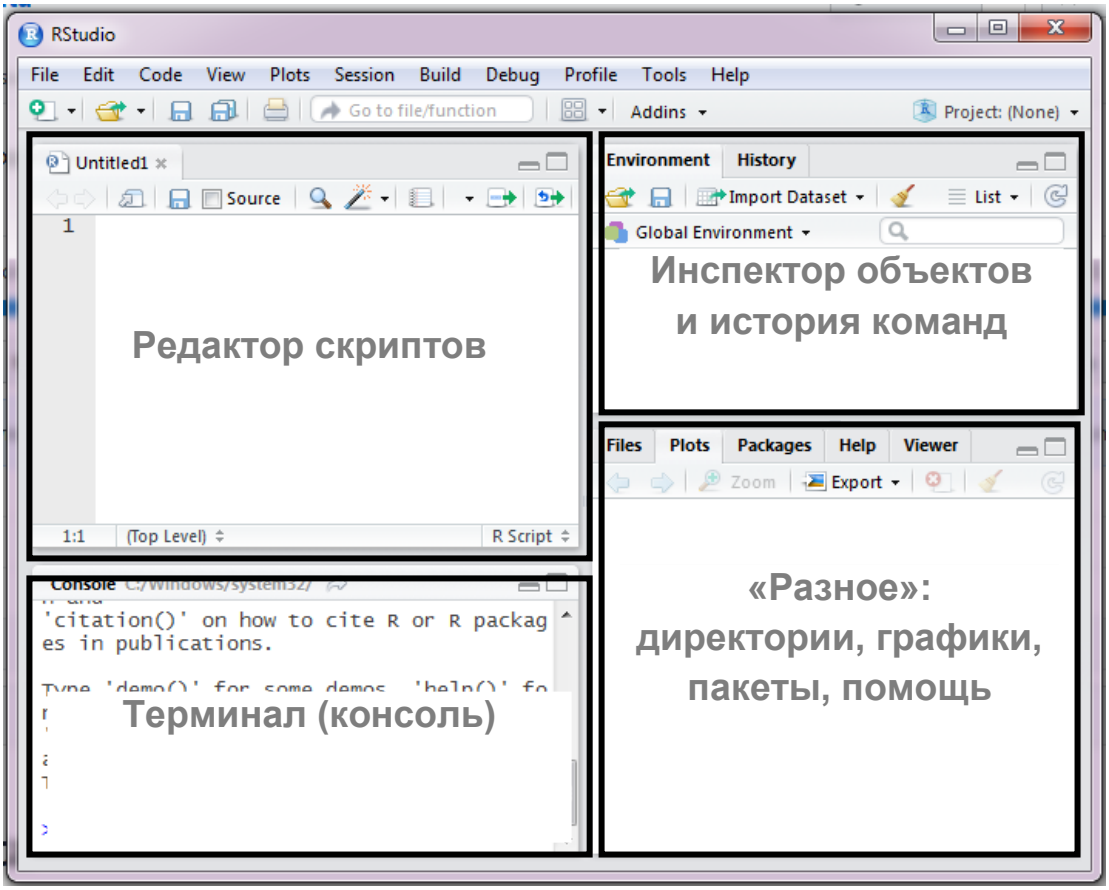


Рис. 1. Вид окна RStudio по умолчанию

Полезные горячие клавиши для редактирования и исполнения кода в RStudio:

- Alt + - (альт и минус) – оператор присваивания справа налево («<−»), который рекомендуется [**5**] использовать вместо символа равенства везде, кроме аргументов функций.
- Ctrl + Enter – запуск на исполнение текущей строки кода в скрипте. Если функция записана в несколько строк, все они будут выполнены.
- Ctrl + S – сохранить текущий скрипт.
- Ctrl + Alt + S – сохранить скрипты во всех вкладках.

[4] Сайт проекта: www.rstudio.com/products/RStudio/

– Ctrl + Shift + Alt + 0 – отобразить все панели (редактор скриптов, терминал, инспектор объектов и «разное»).
– Ctrl + L – очистить окно терминала.
– Alt + Shift + K – просмотр справки по всем сочетаниям клавиш.

Остановимся также на нескольких пунктах меню программы, полезных в начале работы с RStudio.

– File –> Reopen with Encoding… – открыть скрипт снова, с другой кодировкой. Полезно, если символы кириллицы в коде не отображаются корректно.
– Session –> Set Working Directory –> To Source File Location – делает рабочей директорию, в которой лежит открытый скрипт.
– Session –> Save Workspace As… – сохранить в файл с расширением «.RData» рабочее пространство, то есть все объекты и функции R, которые загружены в данный момент в оперативную память.
– Session –> Load Workspace… – загрузить рабочее пространство из файла.

В «Учебнике по языку R для начинающих» Бориса Демешева [**2**] рекомендуется настроить RStudio следующим образом:

1. В Tools –> Global options –> General:
   – убрать флажок «*Restore .Rdata into workspace at startup*»;
   – *Save workspace to .Rdata on exit*: выбрать «Never».

   Это позволит избежать проблем, связанных с записью или загрузкой больших объектов.
2. В Tools –> Global options –> Sweawe:
   – В разделе *Weave .Rnw files using* выбрать «knitr».

   Пакет «knitr» используется для создания текстовых отчётов на лету, по результатам вычислений.
3. В Tools –> Global options –> Code –> Diagnostics:
   – Выставить все флажки.

   Эти настройки отвечают за подсветку синтаксических ошибок и предупреждений, что удобно на этапе отладки кода.
4. В Tools –> Global options –> Code –> Saving:
   – *Default text encoding*: выбрать UTF-8.

   Для обеспечения совместимости с другими операционными системами рекомендуется сохранять скрипты в кодировке UTF-8.

1.1 Настройка рабочей директории

В течение сессии R хранит все объекты и функции в оперативной памяти компьютера. Результаты работы в виде отчётов, графиков, файлов рабочего пространства, а также файлы с исходными данными удобно сохранять в рабочей директории, чтобы не использовать в скриптах абсолютные пути.

Проверить, какая директория является рабочей в данный момент, можно функцией `getwd()` без параметров:

```
> getwd()
[1] "E:\My Documents"
```

Меняют рабочую директорию функцией `setwd()` с символьным аргументом – путём к нужной папке. Обратите внимание: в то время как в стандартной нотации Windows директории отделены символом «\», R использует Unix-нотацию с косой чертой («/»).

```
> # вариант указания пути в R:
> setwd("E:/My Documents/R-work/")
```

Обратная косая черта экранирует следующий символ, поэтому в путях её дублируют.

NB: Одинарные и двойные кавычки в R равноценны.

```
> # другой возможный вариант
> setwd('E:\\My Documents\\R-work\\')
> # проверка результата
> getwd()
 [1] "E:/My Documents/R-work"
```

Назначить рабочую директорию можно и с помощью меню RStudio. Меню Session → Set Working Directory даёт варианты:

- To Source File Location – установить рабочей директорию, в которой сохранён текущий скрипт;
- To Files Pane Location – установить рабочей ту директорию, которая открыта в инспекторе файлов (правая нижняя часть окна RStudio, вкладка Files);

– Choose Directory… – выбрать директорию.

При выборе любого из этих трёх пунктов в консоли R появится вызов функции `setwd()` с соответствующим аргументом.

1.2 Простая арифметика

Раз R – язык для работы со статистикой, попробуем что-нибудь посчитать.

```
> # сложим три числа
> 1 + 2 + 4
[1] 7
> # любые другие арифметические операции
> 7 * 8
[1] 56
> 1 / 2
[1] 0.5
```

Заметим, что разделитель целой и дробной части в R – точка, а запятая – разделитель аргументов функций. Результат появляется в консоли после метки «`[1]`:» это номер первого элемента в строке.

NB: Первый элемент чего бы то ни было в R имеет номер 1.

Дело в том, что в R операции над числами – это частный случай операций над числовыми векторами. Теперь поэкспериментируем с точностью вычислений на элементарном примере.

```
> 1 - 0.09
[1] 0.91
> 1 - 9*10^(-8)
[1] 0.9999999
> 1 - 9*10^(-9)
[1] 1
```

Как видно, граница точности проходит между 10^(-8) и 10^(-9). На практике такая точность нужна далеко не всегда, однако всегда нужно помнить о том, что, выводя результаты, R обычно показывает до пяти знаков после запятой, и таким образом прячет погрешность вычисления [**9**].

1.3 Создание числового вектора

Одна из базовых структур данных в R – вектор. Все элементы вектора относятся к одному типу. Допустим, есть данные о росте трёх сотрудников лаборатории. Создадим числовой вектор `x`:

```
> x <- c(177, 152, 164)
```

Здесь `x` – имя объекта R, «<−» – оператор присваивания, `c()` – функция создания вектора (от англ. *Concatenate*).

NB: Одной из распространённых и досадных ошибок у начинающих пользователей R является замена английской буквы «c» в имени функции на русскую «с». Если что-то не работает, проверьте это в первую очередь.

Просмотреть содержимое объекта `x` можно, набрав в терминале его имя:

```
> x
[1] 177   152   164
```

Как уже было сказано выше, единица в квадратных скобках обозначает номер первого элемента в строке. В R нет скаляров, то есть объект, который состоит из одного числа, – это вектор из одного элемента.

Векторизация вычислений – один из ключевых принципов R. На практике векторизацию можно истолковать так: вместо трёх операций с тремя отдельными числами, то есть традиционного для многих языков программирования цикла, следует применить эту операцию один раз к числовому вектору из трёх элементов. Пользователю, который встречается с векторизацией впервые, поведение обыкновенных арифметических операторов может показаться непредсказуемым. Допустим, мы хотим посчитать средний рост сотрудников лаборатории вручную. Рост первых трёх хранится в векторе `x`, и есть ещё четвёртый сотрудник с ростом 160. Попробуем сложить все четыре значения.

```
# пробуем найти сумму x и 160:
> x + 160
[1] 337 312 324          # видимо, это работает не так
```

На самом деле результат объясним: сработала векторизация. У векторов `x` и 160 разная размерность: три и один элемент соответственно. В такой ситуации R повторяет более короткий вектор столько раз, чтобы его длина совпала с длиной большего вектора. Получается (177, 152, 164) + (160, 160, 160). Эти векторы складываются поэлементно, в результате получается вектор из трёх элементов: (177 + 160 = 337, 152 + 160 = 312, 164 + 160 = 324).

Если требуется сложить именно все элементы `x`, можно обратиться к ним по отдельности с помощью квадратных скобок, функционал которых ещё будет рассмотрен подробно:

```
> x[1] + x[2] + x[3] + 160
[1] 653
```

Однако предпочтительней воспользоваться функцией суммирования, которая складывает элементы всех аргументов:

```
> #сумма
> sum(c(x, 160))
[1] 653
> # среднее
> sum(c(x, 160)) / 4
[1] 163.25
```

Ещё один пример векторизации – работа функции синуса:

```
> sin(42)
[1] -0.9165215
> sin(c(0, pi/2, 3*pi/2, 2*pi))
[1]  0.000000e+00  1.224606e-16 -1.000000e+00 -2.449213e-16
```

Во второй строке кода к вектору из четырёх аргументов, созданному с помощью функции `c()`, применяется функция взятия синуса `sin()`. Аргумент должен быть выражен в радианах, и для наглядности элементы вектора выражены через $\boldsymbol{\pi}$, которое R хранит в константе `pi`. Мы видим, что при этом происходит потеря точности. Для удобства отображения округлим результат, применив функцию округления `round()`. В качестве первого аргумента передадим ей результат функции синуса, а второй аргумент – количество знаков после запятой – зададим как 4:

```
> round(sin(c(0, pi/2, 3*pi/2, 2*pi)), 4)
[1]  0  1 -1  0
```

И `sin()`, и `round()` поддерживают векторизацию аргументов, и нам не нужно перебирать вектор, чтобы применить к его элементам одно и то же преобразование. Векторизация в R лежит в основе большинства преобразований данных, поэтому стоит сразу учиться воспринимать вектора не как массивы, а как самостоятельные объекты вычислений, не требующие обращения к элементам через циклы.

*NB: Создавая свой код на R, стоит опираться на векторизацию, а циклы использовать только когда другого выхода просто нет (и в 95% таких случаев на самом деле альтернативный вариант с векторизацией существует) [**9**].*

1.4 Просмотр структуры объекта

Что содержится внутри условного x? Этот вопрос возникает довольно часто, вне зависимости от того, импортированные это данные или результат, который возвратила функция. Часто для того, чтобы разобраться с ошибкой или предупреждением нужно проверить, к какому типу относится содержимое переменной. Каждый объект R имеет свою структуру, просмотреть которую можно с помощью функции `str()`.

```
> str(x)
num [1:3] 177 152 164
```

Вывод начинается с `num [1:3]`, что означает: объект – числовой вектор из трёх элементов, пронумерованных от 1 до 3. Далее перечисляются первые несколько элементов вектора. Бывают объекты с более сложной структурой. Например, функция `cor.test()` возвращает результаты расчёта коэффициента корреляции Пирсона в виде списка, который содержит как сам коэффициент, так и статистики для проверки его значимости.

```
# зададим ядро генератора случайных чисел
set.seed(42)
x <- rnorm(n = 15)  # x = 15 значений случ. величины N~(0, 1)
y <- rnorm(n = 15)  # y тоже
xy.correlation <- cor.test(x, y)      # корреляция x и y
str(xy.correlation)                   # смотрим структуру
```

```
### List of 9                              # список из 9 объектов
### $ statistic  : Named num 0.471
###   ..- attr(*, "names")= chr "t"
###  $ parameter  : Named int 13
###   ..- attr(*, "names")= chr "df"
###  $ p.value    : num 0.645
###  $ estimate   : Named num 0.13
###   ..- attr(*, "names")= chr "cor"
###  $ null.value : Named num 0
###   ..- attr(*, "names")= chr "correlation"
###  $ alternative: chr "two.sided"
###  $ method     : chr "Pearson's product-moment correlation"
### <…>
```

В коде выше функция `set.seed()` используется для обеспечения воспроизводимости результатов: устанавливая ядро равное одной и той же величине, можно получать один и тот же набор значений случайной величины с заданными характеристиками.

Возвращаемое значение функции может быть пустым. Функция `str()` лишь выводит результат в консоль, поэтому следующее выражение лишено практического смысла (объект `y` пуст):

```
> y <- str(x)
> y
NULL
```

1.5 Вызов справки

С полным списком атрибутов функции и примерами использования можно ознакомиться в справке, для вызова которой служит `help()` и её более короткий вариант – символ «?».

```
# два способа вызвать справку по функции str
help(str)
?str
```

Если название нужной функции неизвестно, можно использовать нечёткий поиск по ключевому слову:

```
??structure
```

Однако в этом случае поиск функции осуществляется только по пакетам, установленным в текущей версии R. Можно сделать запрос на поиск в Интернете по сайту проекта search.r-project.org прямо из терминала R:

```
RSiteSearch('structure')
```

Здесь аргумент является поисковым запросом, поэтому он взят в кавычки. Справка открывается либо во вкладке «Help» правого нижнего раздела окна RStudio, либо в браузере. Вызвать стартовую страницу справки можно так:

```
help.start()
```

Благодаря тому, что R постоянно пополняется новыми пакетами, одни и те же методы могут быть реализованы в нескольких функциях. Например, на запрос по ключевому слову «correlation» (корреляция) появляется довольно длинный список пакетов, в справке к которым упоминается это понятие, и в этом списке можно найти ещё несколько альтернатив функции `corr.test()` из предыдущего примера. Функции отличаются форматом вывода, наличием дополнительных статистик, видом графиков и составом опций – пользователь волен выбрать ту, которая его устраивает. Либо написать свою собственную.

1.6 Сохранение результатов

Скрипты (файлы с кодом) сохраняются с расширением «.R». Объекты сессии R – все загруженные в оперативную память таблицы и функции – также можно сохранять между сессиями в файлах с расширением «.RData». Однако далеко не всегда целесообразно сбрасывать в такой файл всю сессию целиком, лучше после обработки данных сохранить только нужные таблицы в повсеместно используемых форматах, например, в «.csv».

Все диалоги сохранения скриптов в RStudio находятся в меню File:

- Save – сохранить текущий скрипт;
- Save As… – сохранить текущий скрипт с другим именем;
- Save with Ecoding… – сохранить текущий скрипт в другой кодировке;
- Save All – сохранить все скрипты, открытые во вкладках.

Для сохранения и загрузки рабочего пространства можно воспользоваться функциями `save.image()` и `load()`, а в качестве аргумента передать имя файла вместе с расширением. Либо можно сделать это из меню RStudio:

- Session → Save Workspace As… – сохранить рабочее пространство;
- Session → Load Workspace – загрузить рабочее пространство из файла.

NB: В ОС Windows R испытывает проблемы при обращении к файлам и папкам, в абсолютном пути к которым есть кириллица. По умолчанию R устанавливается в папку документов, и если имя пользователя в системе содержит русские буквы, ошибки возникнут почти наверняка. Обойти это можно, назначая рабочей папку, в пути к которой не встречается кириллица. Кроме того, стоит избегать слишком длинных путей к рабочей директории.

1.7 Стандарты оформления кода

С самого начала работы с новым языком стоит приучать себя к стандартам оформления, соблюдение которых облегчает понимание кода так же, как грамотность письменной речи облегчает понимание текста. Существуют рекомендации по оформлению кода на R от Google [5], в которых прописаны основные соглашения относительно именования переменных, файлов, функций, использования элементов разметки и некоторые синтаксические ограничения.

Имена переменных в R не могут начинаться с цифры. Согласно рекомендациям, в именах переменных не нужно использовать знаки подчёркивания «_» и дефисы «−». Кириллицу в именах функций и переменных R воспринимает и обрабатывает нормально, хотя RStudio может неправильно выравнивать такие переменные при переносе кода на следующую строку. Всё же, из соображений совместимости с другими кодировками желательно использовать латиницу. Предпочтительная форма имени переменной – буквы в нижнем регистре, разделённые точками (variable.name). Имена функций начинаются с заглавной буквы и не содержат точек (FunctionName), константы именуются так же как функции, но начинаются с «k».

Напротив, в именах файлов разделять слова рекомендуется символом подчеркивания. Кириллица здесь вполне допустима, хотя это зависит от того, насколько широко вы хотите делиться своим кодом. Сами имена рекомендуется делать ёмкими и содержательными: «код_к_первой_главе.R», «подготовка_данных_по_регионам.R».

Ещё несколько правил, которых мы будем придерживаться далее:

- ставить пробел после запятой и не ставить пробел перед ней;
- не переносить открывающую фигурную скобку «{» на следующую строку;
- не окружать код внутри круглых и квадратных скобок пробелами;
- использовать для присваивания справа налево «<-» вместо «=» везде, кроме аргументов функций;
- использовать `attach` как можно реже;
- заключать условие `else` в фигурные скобки, даже если оно состоит из одной строки;
- комментировать свой код.

В скриптах стоит придерживаться общей последовательности оформления:

1. Сообщение об авторских правах (копирайт).
2. Комментарий автора.
3. Комментарий с описанием файла, включая назначение программы, описание входных данных и отчётов.
4. Инструкции `source()` и `library()`.
5. Определения функций.
6. Вызов функций и исполняемые команды (например, `print`, `plot`), если они используются [5].

2. Виды объектов и типы данных

Немного освоившись с интерфейсом, перейдём к основным типам объектов. В R есть две базовых структуры: вектор и список. В отличие от вектора, в список могут входить элементы разных типов. На базе векторов и списков можно построить структуру любой сложности. Далее в записи кода мы будем использовать нотацию скриптов, опуская символ ввода в консоль «>».

2.1 Последовательности

Запись через двоеточие «1:8» – это последовательность натуральных чисел от 1 до 8. Это один из способов, которым можно задать последовательность чисел, необязательно положительных:

```
-3:6
### [1] -3 -2 -1  0  1  2  3  4  5  6
```

Более общая форма этой записи – функция `seq()` (от «sequence»):

```
seq(from = 1, to = 8)
### [1] 1 2 3 4 5 6 7 8
# названия аргументов from и to можно опускать:
seq(-7, -5)
### [1] -7 -6 -5
seq(-5, 1)
### [1] -5 -4 -3 -2 -1  0  1
```

Шаг последовательности задаёт аргумент «by», по умолчанию равный 1.

```
seq(from = 1, to = 8, by = 2)
### [1] 1 3 5 7
# будьте внимательны с отрицательным by
seq(from = 1, to = 8, by = -2)
### Ошибка в seq.default(from = 1, to = 8, by = -2) :
###  from < to и by < 0
seq(8, 1, -2)        # теперь верно: from > to
### [1] 8 6 4 2
```

Если задать число элементов «length», интервал будет подогнан:

```
seq(10, 15, length = 4)
### [1] 10.00000 11.66667 13.33333 15.00000
```

Ещё одна полезная функция для создания последовательностей – `rep()` (от *repeat* – повторить). Первый аргумент функции – это всегда вектор, который нужно повторить. Второй по порядку неименованный аргумент задаёт количество повторений.

```
# повторить вектор (1, 2) 3 раза
rep(c(35, 14, 8), 3)
### [1] 35 14  8 35 14  8 35 14  8
# своё число повторений для каждого элемента
rep(c(35, 14, 8), 1:3)
### [1] 35 14 14  8  8  8
# аргументы распознаются автоматически по порядку
rep(1:2, c(2, 6))
### [1] 1 1 2 2 2 2 2 2
# ключевое слово each: повторить каждый элемент трижды
rep(c(35, 14, 8), each = 3)
### [1] 35 35 35 14 14 14  8  8  8
```

2.2 Копирование и сравнение объектов

Чтобы скопировать вектор x, подойдёт оператор присваивания «<-». Тот же результат даст знак равенства, но стилистически такой вариант хуже:

```
x <- c(177, 152, 164)    # снова сохраняем рост сотрудников
y <- x
y = x    # запись, аналогичная предыдущей
```

Теперь вектор y повторяет вектор x:

```
y
### [1] 177 152 164
is.vector(y)              # y – это вектор
### [1] TRUE
is.vector(y,"logical")    # тип вектора: логический
### [1] FALSE
is.vector(y,"numeric")    # тип вектора: числовой
### [1] TRUE
```

Здесь TRUE и FALSE – логические значения. Как и все единичные значения в R, они являются векторами размерностью 1.

NB: TRUE и FALSE можно сокращать до T и F соответственно.

Оператор присваивания -> работает слева направо, но пользоваться им не рекомендуется из соображений стиля [5]. Он полезен в конструкциях, построенных с использованием пайплайнов, применение которых в данном пособии не рассматривается.

```
# так тоже можно, но не нужно
x -> z                    # присвоить z значение x
z
### [1] 177 152 164
```

Объекты сравнивают между собой двойным знаком равенства:

```
y == x
### [1] TRUE TRUE TRUE
z == x
### [1] TRUE TRUE TRUE
```

Как видно, сравнение происходит поэлементно. Сравнивать можно векторы разной длины, но размерность одного должна быть кратна размерности другого. Без опасений так можно сравнивать любой вектор только с единичным:

```
y == 177
### [1]  TRUE FALSE FALSE
```

Если же проверяется наличие элементов одного вектора в другом, используйте оператор `%in%`:

```
y %in% c(177, 164)
### [1] TRUE FALSE  TRUE
```

Кавычки в R используются для обозначения символьных значений. При сравнении срабатывает преобразование типов, и если символы текстовой строки совпадают с цифрами числового значения, получаем `TRUE`:

```
y == '177'
### [1]  TRUE FALSE FALSE
y == "177"
### [1]  TRUE FALSE FALSE
```

Однако в арифметических операциях использовать и числа, и строки без явного преобразования нельзя:

```
y
### [1] 177 152 164
y + 1
### [1] 178 153 165
y + '2'
### Error in y + "2" : non-numeric argument to binary operator
```

Поскольку запись `y+1` не содержит оператора присваивания, содержимое вектора y не изменилось.

```
y
[1] 177 152 164
```

2.3 Обращение к элементам вектора, преобразование типов

Запись `y[1]` означает обращение к первому элементу вектора `y`.

```
y[1] == 177
### [1] TRUE
y[2]
### [1] 152
```

Внутри квадратных скобок может быть вектор любого типа. Возможности различных аргументов перечислены в таблице **1**.

Используя оператор `[]` и присваивание, можно изменять отдельные значения вектора.

```
y[3] <- 177
y
### [1] 177 152 177
```

Таблица 1 – Выборка в зависимости от вектора внутри квадратных скобок [**1**]

В квадратных скобках	Результат
Положительный числовой вектор	Элементы вектора с перечисленными номерами
Отрицательный числовой вектор	Элементы вектора кроме перечисленных номеров
Символьный вектор	Элементы вектора с перечисленными именами (или именами измерений матрицы)
Логический вектор	Элементы, соответствующие значениям TRUE и пропущенные значения NA
Пусто	Все элементы вектора

Для явного преобразования типов используются функции `as.numeric()`,`as.character()`. Преобразовать значения `y` в текстовые:

```
y <- as.character(y)
is.vector(y, "numeric")      # проверяем результат
### [1] FALSE
y
### [1] "177" "152" "177"
```

Арифметические операторы с текстовыми значениями не работают:

```
x[1] + x[2]      # вектор x содержит числа
### [1] 329

y[1] + y[2]      # вектор y содержит символы
### Error in y[1] + y[2] : non-numeric argument to binary operator
```

Но можно использовать явное преобразование типов:

```
as.numeric(y[1]) + as.numeric(y[2]) # символы – в числа
### [1] 329
```

Для совмещения двух текстовых значений используется функция `paste()`. По умолчанию между значениями ставится разделитель – пробел. С помощью аргумента «sep» этот разделитель можно менять:

```
paste(y[1], y[2])
### [1] "177 152"
paste(y[1], y[2], sep = "")
### [1] "177152"

paste0(y[1], y[2])          # даёт тот же результат, что sep = ""
### [1] "177152"
paste(y, collapse = ";")    # склеить все элементы вектора
### [1] "177;152;177"
```

При склеивании двух векторов функция `paste()` преобразует все значения в символьные и совмещает элементы попарно:

```
paste(x, y)
### [1] "177 177" "152 152" "164 177"
```

Если размерность векторов различна, сработает векторизация, в результате чего меньший вектор будет циклически повторяться и длина результата будет равна длине наибольшего из векторов. Это сработает даже если размерности векторов не кратны друг другу.

```
# размерности векторов не равны и не кратны друг другу
paste0(c(1, 2, 3, 4, 5, 6, 7), c('A', 'B', 'C'))
### [1] "1A" "2B" "3C" "4A" "5B" "6C" "7A"
```

2.4 Матрицы

Для создания матрицы служит функция `matrix()`. Создадим из вектора `x` матрицу a размерностью 2 на 3, для этого повторим вектор дважды:

```
x
### [1] 177 152 164
a <- matrix(rep(x, 2), 2, 3)
a
### [,1] [,2] [,3]
### [1,]  177  164  152
### [2,]  152  177  164
```

По умолчанию элементы записываются в матрицу *по столбцам*: сначала первый столбец, потом второй, и т.д. Чтобы заполнять матрицу построчно, нужно воспользоваться аргументом «byrow».

```
A <- matrix(rep(x, 2), 2, 3, byrow = T)    # заполняем построчно
A
### [,1] [,2] [,3]
### [1,]  177  152  164
### [2,]  177  152  164
```

NB: R чувствителен к регистру, a и A – это разные объекты.

Размерность матрицы задаётся в специальном свойстве объекта, которое можно просматривать и изменять функцией `dim()`:

```
# просмотреть размерность матрицы A
dim(A)
### [1] 2 3
# создать вектор w – копию x
w <- rep(x, 2)
dim(w) <- c(3, 2)      # задать размерность вектора z...
w                      # ...что превращает его в матрицу
### [,1] [,2]
### [1,]  177  177
### [2,]  152  152
### [3,]  164  164
```

Транспонировать матрицу можно с помощью функции `t()`.

```
t(w)                          # транспонирование
### [,1] [,2] [,3]
### [1,]  177  152  164
### [2,]  177  152  164
```

Строкам и столбцам матрицы можно присваивать имена:

```
rownames(w) <- c("obs1", "obs2", "obs3")
colnames(w) <- c("V1", "V2")
w
### V1  V2
### obs1 177 177
### obs2 152 152
### obs3 164 164
```

Для перемножения матриц именно как матриц нужно использовать специальный оператор `%*%`. Сравните:

```
# оператор * перемножает соответствующие элементы
w * w
### V1    V2
### obs1 31329 31329
### obs2 23104 23104
### obs3 26896 26896

# матричное умножение, результат w^2
w.sq <- w %*% t(w)
w.sq
###       obs1  obs2  obs3
### obs1 62658 53808 58056
### obs2 53808 46208 49856
### obs3 58056 49856 53792
```

Полезны функции, которые возвращают главную диагональ и треугольники матрицы. Причём читатель может убедиться, что они срабатывают не только на квадратных матрицах. Здесь для примера используем квадратную матрицу `w.sq`.

```
# главная диагональ
diag(w.sq)
###  obs1  obs2  obs3
### 62658 46208 53792
```

```
# треугольник под главной диагональю: логические флаги
lower.tri(w.sq)
###       [,1]  [,2]  [,3]
### [1,] FALSE FALSE FALSE
### [2,]  TRUE FALSE FALSE
### [3,]  TRUE  TRUE FALSE

# обнулить всё, кроме верхнего треугольника
w.sq[!upper.tri(w.sq)] <- 0
w.sq
###      obs1  obs2  obs3
### obs1    0 53808 58056
### obs2    0     0 49856
### obs3    0     0     0
```

Обратную матрицу можно найти функцией `solve()`.

```
# создадим квадратную матрицу
m.sq <- matrix(c(1, 0, 0, 1, 1, 0, 1, 1, 1), 3, 3, byrow = T)
m.sq
###      [,1] [,2] [,3]
### [1,]    1    0    0
### [2,]    1    1    0
### [3,]    1    1    1

# найдём обратную матрицу
solve(m.sq)
###      [,1] [,2] [,3]
### [1,]    1    0    0
### [2,]   -1    1    0
### [3,]    0   -1    1
```

2.5 Просмотр списка и удаление объектов, создание класса

В именах объектов можно использовать точку, и эта точка не несёт никакой специальной смысловой нагрузки, как в некоторых объектно-ориентированных языках программирования. Например:

```
x.label <- "Рост сотрудников"
x
### [1] 177 152 164
x.label
### [1] "Рост сотрудников"
```

Получить список созданных объектов можно с помощью функции `ls()`:

```
ls()
### [1] "a"      "A"      "m.sq"      "w" "w.sq"      "x"
### [7] "x.label"      "xy.correlation"      "y"      "z"
```

С аргументом `pattern` можно просмотреть имена, содержащие «x»:

```
ls(pattern = "x")
### [1] "x"        "x.label"        "xy.correlation"
```

Память полезно периодически чистить от объектов, которые больше не понадобятся.

```
# удалим ненужные объекты
rm(a, A, m.sq, w, w.sq, xy.correlation, z)
```

Можно создавать и свои собственные списки из объектов любых типов:

```
# создать список из двух объектов
mylist <- list(x = x, x.label = x.label)

# посмотреть результат
mylist
## $x
### [1] 177 152 164
### $x.label
### [1] "Рост сотрудников"
```

К элементам списка можно обращаться через символ $.

```
# имя_списка$имя_объекта
mylist$x
### [1] 177 152 164
```

Альтернатива – использование двух пар квадратных скобок: [['Имя элемента списка']]. Отличие заключается в том, что после символа “$” R ожидает имя, а не строку символов. Ниже дан пример обращения к элементам списка. Обратите внимание на вектор `letters`, который содержит все строчные буквы английского алфавита.

```
# список из двух элементов
mylist.2 <- list(aaa = 1:5, bbb = letters[1:5])

mylist.2$aaa
### [1] 1 2 3 4 5

mylist.2[[1]]  # обратиться по номеру
### [1] 1 2 3 4 5

mylist.2[['bbb']]  # обратиться по имени
### [1] "a" "b" "c" "d" "e"

subv <- 'bbb'      # можно хранить имя в переменной
mylist.2[[subv]]
### [1] "a" "b" "c" "d" "e"
```

Помещённые в список, `x` и `x.label` остаются двумя различными объектами: числовым вектором и символьным. Логичнее было бы объединить числовой вектор и название соответствующего показателя в один объект. Для этого можно создать пользовательский класс функцией `setClass()`:

```
# создать новый класс объектов «customVector»
#  в объект входят «слоты»: сам числовой вектор (val)
#  и его название (label)
customVector <- setClass("customVector",
                         slots = c(val = "numeric",
                                   label = "character"))

# создать объект нового класса
#  здесь  customVector() – уже конструктор класса
x1 <- customVector(val = x, label = x.label)
# просмотреть созданный объект
x1
### An object of class "customVector"
### Slot "val":
### [1] 177 152 164

### Slot "label":
### [1] "Рост сотрудников"

# просмотреть структуру созданного объекта
str(x1)
### Formal class 'customVector' [package ".GlobalEnv"] with 2 slots
###   ..@ val  : num [1:3] 177 152 164
###   ..@ label: chr "Рост сотрудников"
```

Теперь обратиться к отдельным элементам объекта класса «customVector» можно через имя объекта и имя элемента, разделённые символом @:

```
x1@val
### [1] 177 152 164

x1@label
### [1] "Рост сотрудников"

# слот val – вектор, обратимся к его элементу
x1@val[2]
### [1] 152
```

Для работы с пользовательскими классами можно создавать методы функцией `setMethod()`. Например, назначим новый метод обработки оператора сложения («+») для случая, когда одно из слагаемых – объект пользовательского класса `customVector`, а другое – число (`numeric`). Аргумент `function` – функция, описывающая действия, которые нужно проделать со слагаемыми; в скобках перечисляются аргументы функции (сами слагаемые), их типы задаются в аргументе `signature`:

```
# создание метода сложения с числом
setMethod("+", signature(e1 = "customVector", e2 = "numeric"),
          function(e1, e2) e1@val + e2)

# использование метода
x1 + 42
### [1] 219 194 206
```

Также полезно бывает переписывать для своего класса стандартный метод вывода в окне консоли. Когда в консоль вводится имя переменной, срабатывает функция `show()`. Чтобы правильно переписать её, в аргументе `signature` нужно указать не любое имя входной переменной, а «object». Обратите внимание: если в теле функции несколько команд, они должны быть заключены в фигурные скобки.

Непосредственно за вывод в консоль отвечает функция `print()`.

```
setMethod("show", signature(object="customVector"),
          function(object) {print(object@label);
                            print(object@val)})
```

```
# проверяем, как теперь выводится содержимое x1
x1
### [1] "Рост сотрудников"
### [1] 177 152 164
```

Для удаления объектов используется функция `rm()`:

```
# удалить ненужные объекты
rm(mylist, mylist.2, subv, y)
# просмотреть список оставшихся объектов
ls()
### [1] "customVector"      "x"      "x1"      "x.label"
```

Чтобы удалить все созданные пользователем объекты, не перечисляя их имён можно воспользоваться аргументом `list` (список):

```
# удалить всё
rm(list=ls())
ls()                  # теперь список объектов окажется пустым
### character(0)
```

2.6 Класс объекта: фрейм данных; импорт из .csv

Вектор позволяет хранить значения одного показателя определённого типа: числового, символьного, логического. Привычному представлению исходных данных в виде таблицы в R соответствует фрейм данных, `data.frame`. Далее мы будем называть такой класс фреймом, а не таблицей, чтобы отличать от более продвинутой структуры – таблицы данных, или `data.table`. Фрейм данных может содержать значения нескольких показателей, расположенных в столбцах, а также заголовки строк и столбцов. Физически фрейм – это одномерный список из векторов одинаковой длины [1]. Для импорта данных из файла служит универсальная функция `read.table()` и её специальные версии `read.csv()`, `read.csv2()` и другие. В качестве аргументов `read.table()` следует указать имя файла данных (полный путь, либо относительно рабочей директории R), разделитель значений, разделитель целой и дробной части чисел и несколько переменных разметки таблицы:

- `header`: логическое значение. Если TRUE, первая строка таблицы читается как названия столбцов (показателей);
- `stringsAsFactors`: логическое значение. Если TRUE, то

символьные данные преобразовываются в числовые факторы, а значит, каждому уникальному символьному значению будет присвоен его номер по порядку. Если FALSE, такие столбцы останутся символьными;

- `row.names`: либо вектор названий наблюдений (строк), либо номер столбца таблицы данных, который содержит эти названия;
- `fileEncoding`: текстовая строка, которая задаёт кодировку импортируемого файла. Некоторые варианты значений: "unicode", "UTF-8", "cp1251" (русская раскладка Windows).

Функция `read.table()` возвращает фрейм данных, который содержит результаты импорта. Импортируем таблицу со сведениями о сотрудниках лаборатории приборов будущего: файл FGLab.csv, который лежит в рабочей директории R (установленной в начале занятия с помощью функции `setwd()`). Вид таблицы показан на рисунке **2**.

Стоит иметь в виду, что большие файлы функция `read.table()` будет обрабатывать долго, поскольку ей требуется пройти весь текст в поисках заданных синтаксических символов. Если заранее известно, что файл записан как текст с разделителями, можно использовать `read.csv()` и `read.csv2()`. В этих вариантах функции разделители данных и разрядов заданы по умолчанию.

```
# импорт фрейма данных из .csv
dfLab <- read.table("FGLab.csv", header = T, sep = ",",
                    dec = ".", stringsAsFactors = F,
                    row.names = 1)
# импорт фрейма с помощью read.csv()
dfLab <- read.csv("FGLab.csv", stringsAsFactors=F , row.names=1)
```

	A	B	C	D	E	F	G
1	№ п/п	Имя	Пол	Рост	Вес	Возраст	Размер майки
2	001	Окабе Ринтаро	муж	177	59	18	L
3	002	Шиина Маюри	жен	152	45	16	S
4	003	Хашида Итару	муж	164	98	19	XXL
5	004	Макисе Курису	жен	160	45	18	S
6	005	Урушибара Рука	муж	161	44	16	XS
7	006	Фейрис Нян-Нян (Акиха Румико)	жен	143	43	17	S
8	007	Кирию Моэка	жен	167	54	20	M
9	008	Сузуха Амане	жен	163	51	18	M

Рис. 2. Вид таблицы с данными о сотрудниках лаборатории

```
# просмотр результата
dfLab
###                              Имя Пол Рост Вес Возраст Размер.майки
### 1                  Окабе Ринтаро муж  177  59      18            L
### 2                    Шиина Маюри жен  152  45      16            S
### 3                   Хашида Итару муж  164  98      19          XXL
### 4                  Макисе Курису жен  160  45      18            S
### 5                 Урушибара Рука муж  161  44      16           XS
### 6 Фейрис Нян-Нян (Акиха Румико) жен  143  43      17            S
### 7                    Кирию Моэка жен  167  54      20            M
### 8                   Сузуха Амане жен  163  51      18            M

# просмотр структуры фрейма
str(dfLab, vec.len = 2)
### 'data.frame':  8 obs. of  6 variables:
###  $ Имя          : chr  "Окабе Ринтаро" "Шиина Маюри" ...
###  $ Пол          : chr  "муж" "жен" ...
###  $ Рост         : int  177 152 164 160 161 ...
###  $ Вес          : int  59 45 98 45 44 ...
###  $ Возраст      : int  18 16 19 18 16 ...
###  $ Размер.майки: chr  "L" "S" ...
```

Верная интерпретация кириллицы при импорте зависит от кодировки самого файла и от кодировки операционной системы. Если кодировка импортируемой таблицы известна, её нужно указать явно в аргументе «fileEncoding».

NB: Для обеспечения универсальности и лёгкого переноса данных с кириллицей между разными операционными системами стоит использовать кодировку UTF-8.

Фрейм данных можно отредактировать в графическом режиме, воспользовавшись функцией `edit()`.

```
dfLab <- edit(dfLab)
```

Появится таблица, показанная на рисунке **3**, которая позволяет добавлять новые наблюдения и изменять существующие. Щелчок левой кнопкой мыши по заголовку столбца открывает контекстное меню, в котором можно изменить тип данных («Real» или«Character») и переименовать столбец («Change Name»).

Графический режим пригоден для просмотра и мелких правок небольших таблиц, но не обладает функциональностью команд в терминале. Кроме того, графический режим редактирования бесполезен при создании пользовательских функций обработки данных. Использовать эту опцию, вообще говоря, опасно, потому что результаты действий пользователя не записываются в скрипт.

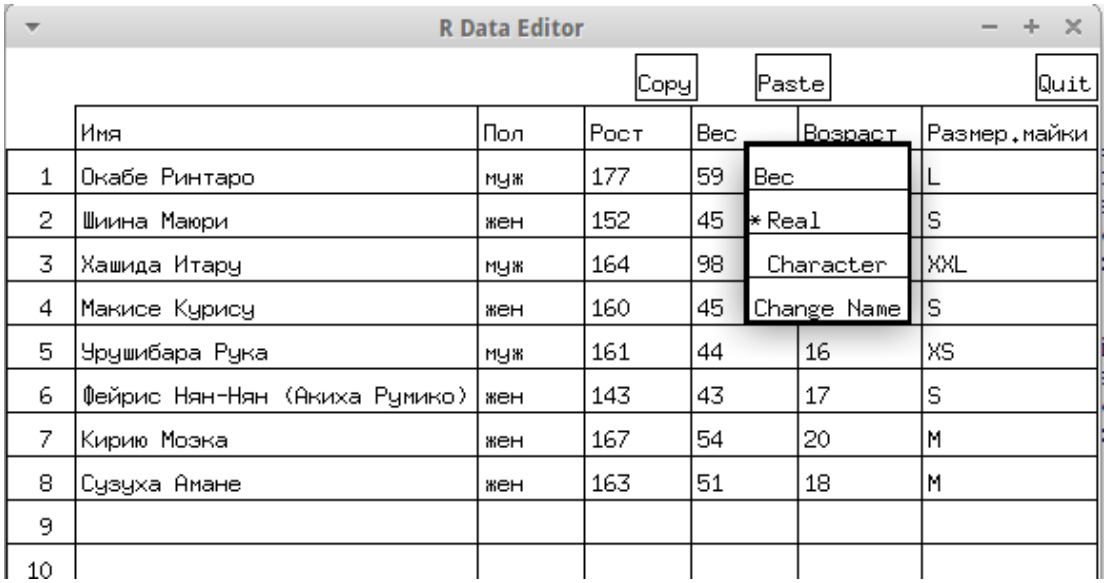


Рис. 3. Фрейм данных в режиме редактирования

Покажем, как можно редактировать фрейм через скрипт. Допустим, нужно заменить первое значение в столбце «Размер майки» с «L» на 56. Для этого воспользуемся операторами присваивания и обращения к элементам векторов и списков. Доступ к значениям столбца даёт, например, оператор квадратных скобок.

NB: В коде на языке R при обращении к отдельным столбцам фрейма данных имя столбца отделяется от имени фрейма символом «$». Это стандартное обращение к элементу списка, поскольку фрейм – это список векторов (столбцов фрейма).

```
# первые три значения в столбце «Размер майки»
dfLab$Размер.майки[1:3]
### [1] "L"   "S"   "XXL"
```

Покажем, как заменить отдельное значение.

```
# фрейм до внесения изменений
str(dfLab, vec.len = 2)
### 'data.frame': 8 obs. of  7 variables:
### <…>
###  $ Размер.майки  : chr  "L" "S" ...
```

При замене в столбце с символьными данными символьного значения на числовое тип столбца не изменится, а число будет преобразовано в текст:

```
dfLab$Размер.майки[1] <- 56   # собственно замена значения
str(dfLab, vec.len = 2)         # структура после замены
### 'data.frame': 8 obs. of  6 variables:
### <…>
###  $ Размер.майки: chr  "56" "S" ...
```

Это происходит потому, что в одном столбце всегда содержатся данные одного типа. Изменить тип после создания фрейма можно только с помощью явной инструкции `as.numeric()`, `as.character()` и т. д.

```
# меняем обратно
dfLab$Размер.майки[1] <- 'L'

dfLab$Размер.майки
### [1] "L"   "S"   "XXL" "S"   "XS"  "S"   "M"   "M"
```

2.7 Тип шкалы: порядковая; тип данных: фактор

Рассмотрим подробнее факторы, в которые R пытается преобразовывать все символьные столбцы импортируемых таблиц. Так он перекодирует данные из номинальной шкалы в порядковую, чтобы сделать для них доступными методы количественного анализа. Делать такое преобразование по умолчанию для всех нечисловых показателей некорректно, но иногда такая операция целесообразна. Известно, например, что размер одежды L («large») больше, чем S («small»), то есть значения в столбце «Размер майки» выражены не в номинальной, а в порядковой шкале. Чтобы превратить этот показатель в фактор, нужно воспользоваться функцией `ordered()`, указав метки – уникальные значения размера – в порядке возрастания в аргументе `levels`.

```
# столбец "Размер майки", данные в порядковой шкале
dfLab$Размер.майки <- ordered(dfLab$Размер.майки,
                  levels=c("XS", "S", "M", "L", "XL", "XXL"))

# просмотр структуры фрейма
str(dfLab, vec.len=2)
### 'data.frame': 8 obs. of  6 variables:
###  $ Имя          : chr  "Окабе Ринтаро" "Шиина Маюри" ...
### <…>
###  $ Размер.майки: Ord.factor w/ 6 levels
              "XS"<"S"<"M"<"L"<..: 4 2 6 2 1 ...
```

Обратите внимание: при создании фактора мы добавили в уровни значение «XL», которое в таблице не встречается. Убедиться в этом можно, посчитав частоту встречаемости уникальных значений в этом столбце с помощью функции `table()`.

```
# значения размеров маек
table(dfLab$Размер.майки)
### XS   S   M   L  XL XXL
###  1   3   2   1   0   1
```

Мы добавили уровень «XL» для того, чтобы в будущем такое наблюдение можно было добавить в таблицу. Дело в том, что после того как уровни фактора заданы, при добавлении значений производится проверка новых данных, и варианты не из списка уровней добавить уже нельзя.

Нельзя не упомянуть и более простой способ создания фактора из числового вектора: функции `factor()` и `as.factor()`.

```
# создать фактор из числового вектора
test.f <- as.factor(c(3, 2, 1, 3, 2, 2))
test.f
### [1] 3 2 1 3 2 2
### Levels: 1 2 3

# можно добавлять существующий уровень фактора
test.f[1] <- 2
test.f
### [1] 2 2 1 3 2 2
### Levels: 1 2 3

# создать фактор из символьного вектора
test.f <- factor(dfLab$Имя)
test.f
### [1] Окабе Ринтаро   Шиина Маюри   Хашида Итару
### [4] Макисе Курису Урушибара Рука Фейрис Нян-Нян (Акиха Румико)
### [7] Кирию Моэка   Сузуха Амане
### 8 Levels: Кирию Моэка   Макисе Курису   Окабе Ринтаро ...

# нельзя добавить новый уровень фактора
test.f[1] <- 'Киситани Синра'             # ошибка
### Warning message:
### In `[<-.factor`(`*tmp*`, 1, value = "Киситани Синра") :
###   invalid factor level, NA generated
```

2.8 Выбор отдельных столбцов и строк фрейма данных

Структуру фрейма можно менять: переименовывать столбцы, изменять их содержимое и количество, добавлять новые.

Чтобы выбрать только два столбца фрейма данных, создадим вспомогательный вектор `keep`, содержащий имена нужных столбцов. Аргумент `drop = FALSE` не является обязательным; он показывает, что в случае, если пользователь оставит только один столбец, фрейм останется фреймом, а не превратится в вектор.

```
# оставить только столбцы «Имя» и «Возраст»
keep <- c('Имя', 'Возраст')

# посмотреть результат
dfLab[, keeps, drop = FALSE]
###                               Имя Возраст
### 1                  Окабе Ринтаро      18
### 2                    Шиина Маюри      16
### 3                   Хашида Итару      19
### <…>
### 6 Фейрис Нян-Нян (Акиха Румико)      17
### 7                    Кирию Моэка      20
### 8                   Сузуха Амане      18
```

Можно выбирать столбцы и строки фрейм, указывая их номера в квадратных скобках.

```
# элемент в третьей строке, первом столбце фрейма
dfLab[3, 1]
### [1] "Хашида Итару"
```

В квадратных скобках стоят числа, но вспомним, что любое число в R – это единичный вектор, поэтому в качестве номеров нужных строк и столбцов можно использовать вектора:

```
dfLab[3:5, 1:2]     # строки с 3 по 5, столбцы с 1 по 2
###              Имя Пол
### 3   Хашида Итару муж
### 4  Макисе Курису жен
### 5 Урушибара Рука муж
```

Можно выбирать строки и столбцы, которые не идут друг за другом:

```
dfLab[c(1, 4), c(1, 5)]       # строки 1 и 4, столбцы 1 и 5
###             Имя Возраст
### 1 Окабе Ринтаро      18
### 4 Макисе Курису      18
```

Работают все возможности оператора квадратных скобок, перечисленные в таблице **1** (стр. 13). Так, используя отрицательные числа в квадратных скобках, можно выбрасывать из фрейма ненужные строки и (или) столбцы:

```
dfLab[c(-1, -6), -3:-7] # строки без 1 и 6, столбцы без 1 и 5
###               Имя Пол
### 2    Шиина Маюри жен
### 3   Хашида Итару муж
### 4  Макисе Курису жен
### 5 Урушибара Рука муж
### 7    Кирию Моэка жен
### 8   Сузуха Амане жен
```

Вместо векторов с номерами нужных строк можно задавать логические вектора, которые являются результатом проверки условия на значения:

```
dfLab[dfLab$Возраст >= 18, c(1, 5)]   # сотрудники
                                      #не младше 18
###             Имя Возраст
### 1 Окабе Ринтаро      18
### 3  Хашида Итару      19
### 4 Макисе Курису      18
### 7   Кирию Моэка      20
### 8  Сузуха Амане      18
```

Чтобы задать несколько условий отбора строк, можно использовать логические операторы: `&` – логическое И; `|` – логическое ИЛИ; `!` – логическое НЕ.

```
# сотрудники-мужчины старше 16
dfLab[dfLab$Пол == 'муж' & dfLab$Возраст > 16, c(1, 2, 5)]
###            Имя  Пол Возраст
### 1 Окабе Ринтаро муж      18
### 3  Хашида Итару муж      19
```

Фреймы большой размерности неудобно просматривать в консоли. Здесь удобно пользоваться функциями `head()` и `tail()`.

```
head(dfLab)                 # первые несколько строк
tail(dfLab, n = 3)          # последние три строки
```

2.9 Расчёт описательных статистик

В R есть функции для расчёта среднего, дисперсии, квартилей и других описательных статистик. Однако статистики выборки необязательно рассчитывать по отдельности. Функция `summary()` с аргументом – именем фрейма данных или вектора – выдаёт таблицу описательных статистик по каждому из столбцов, в зависимости от типа данных. Первые два столбца фрейма «dfLab» содержат символьные значения, и статистики по ним не рассчитываются. Ниже для числовых столбцов рассчитаны: минимум (Min.), первый квартиль (1st Qu.), медиана (Median), среднее арифметическое (Mean), третий квартиль (3rd Qu.), максимум (Max.). Последний столбец – фактор, поэтому для него подсчитаны частоты каждого из значений.

```
summary(dfLab, digits = 0)
###      Имя                Пол                Рост
### Length:8          Length:8          Min.   :143
### Class :character  Class :character  1st Qu.:158
### Mode  :character  Mode  :character  Median :162
###                                     Mean   :161
###                                     3rd Qu.:165
###                                     Max.   :177
###      Вес         Возраст      Размер.майки
### Min.   :43   Min.   :16    XS :1
### 1st Qu.:45   1st Qu.:17    S  :3
### Median :48   Median :18    M  :2
### Mean   :55   Mean   :18    L  :1
### 3rd Qu.:55   3rd Qu.:18   XL  :0
### Max.   :98   Max.   :20   XXL :1
```

Отдельные описательные статистики можно рассчитывать с помощью специальных функций (в скобках достаточно указать вектор, по которому делается расчёт):

```
x <- dfLab$Рост
```

```
mean(x)        # среднее арифметическое
### [1] 160.875

median(x)      # медиана
### [1] 162

range(x)       # вектор из 2 значений: минимум и максимум
### [1] 143 177

cumsum(x)      # вектор накопленных сумм элементов
### [1]  177  329  493  653  814  957 1124 1287

cumprod(x)     # вектор накопленных произведений элементов
### [1] 1.770000e+02   2.690400e+04   4.412256e+06
### [4] 7.059610e+08   1.136597e+11   1.625334e+13
### [7] 2.714308e+15   4.424321e+17

var(x)         # дисперсия
### [1] 101.5536

quantile(x, c(25, 75)/100) # квантили заданных вероятностей
###    25%    75%
### 158.00 164.75
```

Обычно статистики рассчитываются по столбцам фрейма. Что делать, если необходимо найти все средние по строкам? Здесь на помощь приходит функция `apply()` и её различные вариации. Один из вариантов – функция `lapply()` – применяет заданную функцию к списку. Допустим, для начала нам нужно определить, какие столбцы фрейма являются числовыми. Для этого применим функцию `as.numeric()` ко всем столбцам:

```
# какие из столбцов фрейма числовые?
lapply(dfLab, is.numeric)
$`Имя`
[1] FALSE

$Пол
[1] FALSE

$Рост
[1] TRUE

$Вес
[1] TRUE

$Возраст
```

```
[1] TRUE

$Размер.майки
[1] FALSE
```

Мы не указали явно, что функцию нужно применить к столбцам фрейма, и в этом нет необходимости, поскольку фрейм – это список векторов. Функция `lapply()` возвращает список, что не всегда удобно. Чтобы получить тот же результат в виде вектора, используем `sapply()` (*s* – от *simple*):

```
# то же в виде вектора
sapply(dfLab, is.numeric)
###  Имя      Пол    Рост     Вес  Возраст Размер.майки
###  FALSE  FALSE    TRUE    TRUE    TRUE       FALSE
```

Теперь мы можем отобрать только числовые столбцы фрейма:

```
dfLab[, sapply(dfLab, is.numeric)]
###   Рост Вес Возраст
### 1  177  59      18
### 2  152  45      16
### 3  164  98      19
### 4  160  45      18
### 5  161  44      16
### 6  143  43      17
### 7  167  54      20
### 8  163  51      18
```

И применить к ним функцию вычисления среднего:

```
# среднее по числовым столбцам
sapply(dfLab[, sapply(dfLab, is.numeric)], mean)
###    Рост     Вес Возраст
### 160.875  54.875  17.750
```

Причём с помощью `apply()` можно применять усреднение как к столбцам, так и к строкам фрейма. За это отвечает второй аргумент функции, указывающий, к какому измерению применить преобразование. Значение 1 соответствует строкам фрейма, значение 2 – столбцам.

```
# среднее по числовым столбцам: второй аргумент = 2
```

```
apply(dfLab[, sapply(dfLab, is.numeric)], 2, mean)
###    Рост     Вес Возраст
### 160.875  54.875  17.750

# среднее по числовым строкам: второй аргумент = 1
apply(dfLab[, sapply(dfLab, is.numeric)], 1, mean)
     1       2       3       4       5       6       7       8
84.667  71.000  93.667  74.333  73.667  67.667  80.333  77.333
```

2.10 Откуда ещё брать данные

Методов получения данных в R несколько:

1. Искусственное создание последовательностей с помощью функций `seq()`, `rep()`;
2. Генерация случайных величин, например, с помощью функции `rnorm()` – случайная величина, распределённая по нормальному закону;
3. Загрузка встроенных данных: функция `data()`. В скобках указывается имя набора данных. Например, `data(iris)` – набор данных для тестирования многомерных статистических методов «ирисы Фишера».
4. Импорт из внешних файлов на компьютере. Файлы необязательно могут находиться в рабочей директории. Для выбора файла с помощью диалога можно воспользоваться функцией `file.choose()`.
5. Загрузка файлов из интернета с помощью функции `download.file()`. Для работы функции через защищённый протокол «https» под Mac OS нужно обязательно указывать аргумент method = 'curl'.
6. Синтаксический анализ (парсинг) файлов и веб-страниц, в процессе которого таблица заполняется содержимым тегов разметки, отобранных по заданным правилам. Для этого используются специальные пакеты, например, XML, jQuery, rjson.

2.11 Задачи, в которых может встать вопрос использования циклов

Как уже было сказано выше, циклы в R не приветствуются. Главные причины – это векторизация и наличие уже встроенных функций, которые неявно реализуют циклы, дают компактное представление кода и верны идеологически. Ниже перечислены задачи обработки данных, для выполнения которых пользователь, знакомый с некоторыми другими языками программирования, может сначала обратиться к циклам, однако делать это стоит далеко не всегда.

1. Если нужно сложить / упорядочить / посчитать тангенс от каждого элемента вектора / столбца таблицы – цикл не нужен. R – функциональный язык программирования, и для всего этого уже написаны функции. Сложить все элементы вектора друг с другом поможет `sum()`. Сделать сортировку – `sort()`. Отобрать уникальные – `unique()`. Если нужно применить функцию тангенса к каждому элементу вектора, поможет векторизация. Большинство подобных функций принимают в качестве аргумента вектор, а в R отдельное число – это вектор единичной длины. Поэкспериментируйте, чтобы привыкнуть.
2. Если нужно «пройтись» по таблице (фрейму) с данными и заменить / удалить / вставить только значения, удовлетворяющие условию – цикл не нужен. Для этого есть оператор квадратных скобок `[]`, внутри которого записывается нужное условие на данные. Оператор извлечёт только подлежащие замене / удалению / вставке значения, и перебор строк или столбцов в цикле здесь не нужен.
3. Если нужно сделать агрегирование таблицы данных не просто по заданному столбцу, но в группах по значениям другого столбца таблицы. Например, посчитать средний стаж сотрудников по отделам организации, и все данные в одном файле, причём отделов 60. Оператор `[]` даёт нечитабельный и громоздкий код. Здесь цикл тоже не нужен. В данном случае сначала стоит разбить фрейм по классификационной переменной (в примере – по отделам) с помощью функции `split()`. Получится список фреймов, обработать который помогут функции, которые применяют другие функции к спискам: `lapply()`, `sapply()`, `tapply()`. Чтобы разобраться в их работе, нужно приложить некоторые усилия, однако результат того стоит. Другой вариант, опять же без

использования циклов – использовать объект типа data.table (таблица данных).

4. Нужно перебрать по порядку длинный список. Например, список гиперссылок на файлы с данными для загрузки, импортированный из текстового файла. Вы уверены, что функции lapply(), sapply(), tapply() и подобные не работают? Проверьте ещё раз.
5. Нужно перебрать файлы в заданной директории, причём их названия заранее неизвестны, либо не содержат системы. Либо речь о переборе гиперссылок при парсинге веб-страницы, которые, опять же, априори не определены. Вот это одна из немногих задач, в которой использование цикла может быть обосновано.

3. Базовая графика

Многие пользователи ценят R именно за его графические возможности. Существует много библиотек, которые позволяют строить графики в R, и по мере появления новых методов визуализации их список пополняется:

- Базовая библиотека (пакет «base» входит в базовую сборку);
- Пакет «lattice», предназначенный для визуализации многомерных данных;
- Пакет «ggplot2», реализующий грамматику графиков Лэланда Вилкинсона и позволяющий строить красивые статические графики практически всех видов (в т.ч. карты);
- Пакеты gVis, leaflet, shiny, plotly и т.п. предоставляют драйвера для сторонних библиотек, которые строят интерактивные графики;
- Пакеты под экзотические визуализации: круговые графики, трубопроводы, мозаичные графики и т.д.;
- Пакеты для 3D визуализаций.

С помощью базовой графики в R можно собрать любой статический график с нуля, последовательно дорисовывая элементы на полотно. График передаётся на устройство вывода: экран, файл, отчёт. Неудобство базовой графической системы в том, что графики, которые рисуют встроенные функции, довольно примитивно оформлены. Чтобы настроить нужный для статьи или отчёта формат отображения, обычно требуется очень много кода. Итоговый результат нельзя сохранить как объект, как это можно делать с графиками, построенными средствами других библиотек. С другой стороны, базовые функции охватывают все элементы настройки и оформления, и с их помощью можно построить график любой сложности.

3.1 Точечный график

Простейший график разброса можно построить с помощью функции `plot()`. В качестве исходных данных функции подаются два числовых вектора: `x` с координатами по горизонтальной оси и `y` с координатами по вертикальной. При этом если x опущен и функция вызвана только с одним числовым вектором в качестве аргумента, по горизонтальной оси будет отложен номер наблюдения.

Для примера сгенерируем вектор `y` из 100 элементов, взятых из распределения, соответствующего стандартному нормальному закону.

```
set.seed(42)
y <- rnorm(100)        # создать вектор y ~ N(0, 1)
plot(y, main = 'Выборка из значений случайной величины Y',
     xlab = 'Номер наблюдения', ylab = 'Y')      # Рисунок 4
```

По умолчанию `plot()` строит для кросс-секционной выборки график разброса. В данном случае мы построили график разброса одной переменной: она отложена по вертикали, тогда как по горизонтали расположены номера наблюдений. Логичнее использовать точечный график для поиска взаимосвязей двух случайных величин. Создадим числовой вектор `x`, статистически связанный с вектором `y`, и построим график разброса.

```
set.seed(4)
x <- y*42 + rnorm(100, mean = 2, sd = 15)  # создать вектор x
# график разброса
plot(x, y, main = 'Разброс Y относительно X')  # Рисунок 5а
```

Функция `abline()` добавляет на график прямую: наклонную (аргументы `a`, `b`), горизонтальную (аргумент `h`) или вертикальную (аргумент `v`). Добавим на график линии сетки, соответствующие делениям осей:

```
abline(v = seq(-100, 100, by = 50), h = -2:3,
       col = "lightgray", lty = 3)             # Рисунок 5б
```

Аргументы `col` и `lty` относятся к графическим параметрам. Первый задаёт цвет (color), второй – тип линии (line type). Цвета, обозначаемые ключевыми словами, можно найти по ссылке [8]. Аргумент `col` также может быть числовым.

Аргумент `lty` (сокращение от *line type*) – числовой, типы линий в зависимости от его значения показаны на рисунке 6а. Ещё один полезный графический аргумент – `pch` (*point character*). Его значение должно быть числовым вектором, задающим тип маркеров для каждого наблюдения. Значения показаны на рисунке 6б. Ещё один числовой аргумент `lwd` (*line width*) задаёт толщину линии.

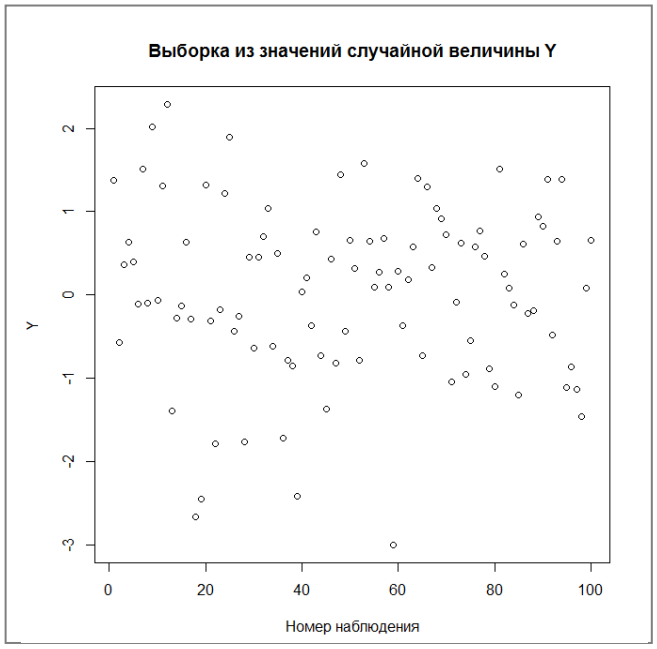


Рис. 4. Простой график разброса (оформление по умолчанию)

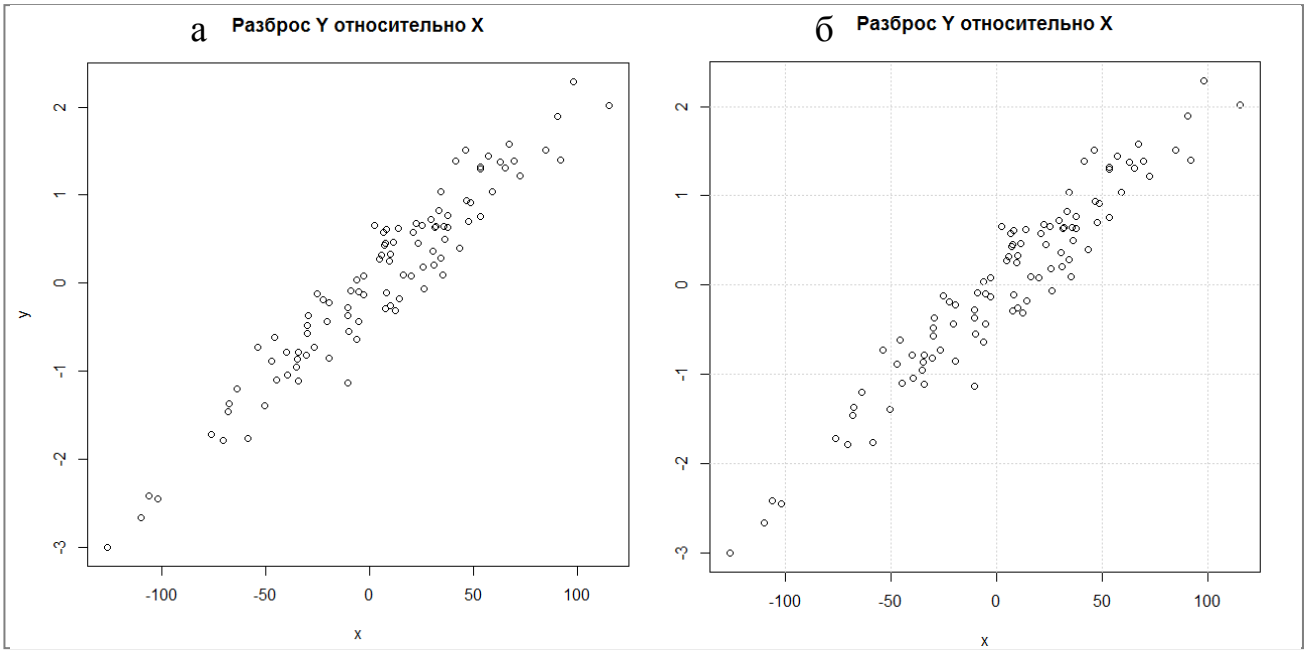


Рис. 5. Графики взаимного разброса двух показателей

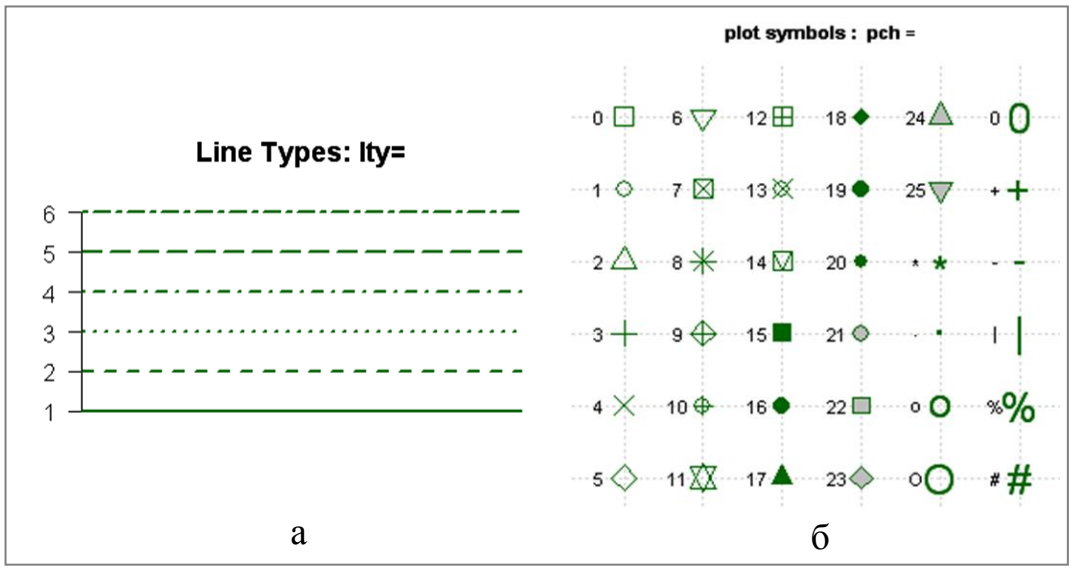


Рис. 6. Типы линий и маркеров [10]

Выделим на графике точки, которые соответствуют значениям `x` выше среднего. Для этого создадим вектор `dots`, элемент которого равен 1, если соответствующий `x` больше среднего, и перестроим график с аргументом `pch`:

```
dots <- as.numeric(x > mean(x))
plot(x, y, main = 'Разброс Y относительно X',
     pch = dots*2 + 3)
# добавим на график сетку
abline(v = seq(-100, 100, by = 50), h = -2:3,
       col = "lightgray", lty = 3)                    # Рисунок 7
```

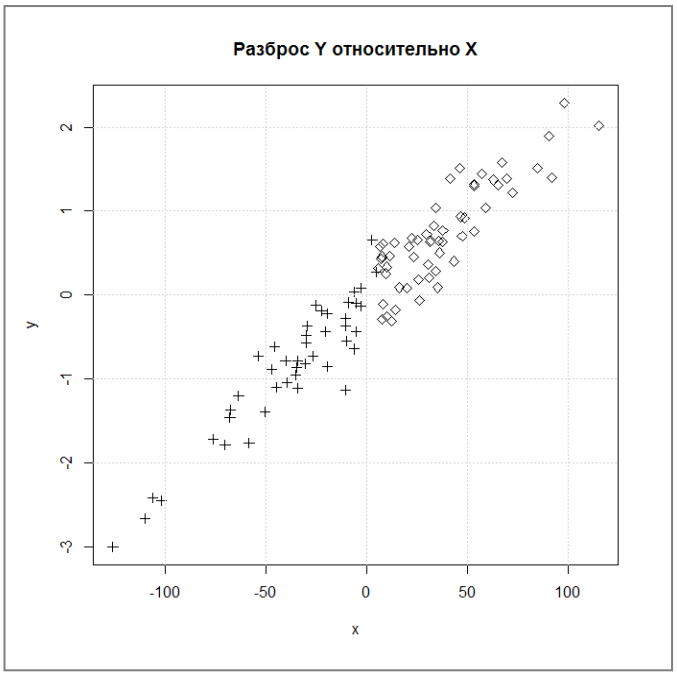


Рис. 7. График взаимного разброса с разными маркерами

3.2 Коробчатая диаграмма, или «ящик с усами»

Коробчатая диаграмма служит для изображения разброса значений одного показателя. Для созданной ранее случайной величины Y:

```
boxplot(y, horizontal = TRUE)                    # Рисунок 8а
```

По этому графику можно определить, есть ли в выборке аномальные наблюдения. Они отмечаются точками за пределами «усов». Аргумент `range` определяет, какие наблюдения считать аномальными: по умолчанию он равен 1.5, и все наблюдения, которые отклоняются от среднего более, чем на 1.5 *межквартильных расстояния*[5], относятся к аномальным. Меняя значение `range`, этот интервал можно сузить или расширить.

```
# сузить интервал, в котором наблюдения считаются
# неаномальными, до 1 межквартильного расстояния
boxplot(y, horizontal = TRUE, range = 1)      # Рисунок 8б
```

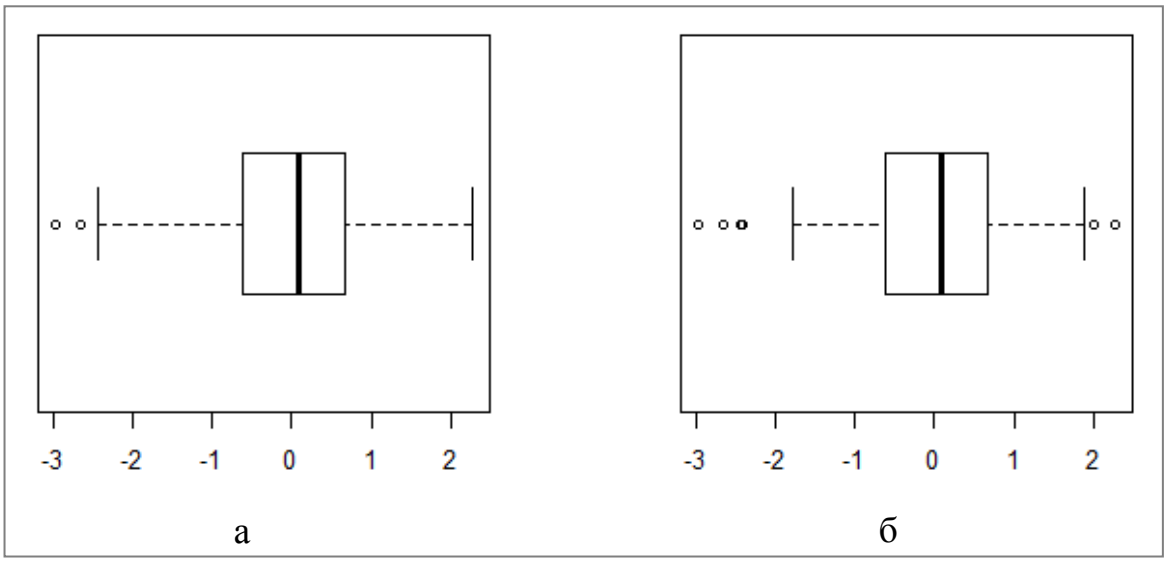


Рис. 8. Коробчатые диаграммы с разным размахом «усов»

3.3 Совмещаем нескольких графиков

5 Третий квартиль распределения минус первый квартиль.

По умолчанию на устройство вывода подаётся только один график. Это можно изменить с помощью функции `par()`. Нарисуем две гистограммы друг под другом (рисунок 9).

```
# изменяем размещение графиков: 2 строки, 1 столбец
par(mfrow = c(2, 1))
hist(y)                    # строим гистограмму y
hist(x)                    # строим гистограмму x
par(mfrow = c(1, 1))       # меняем размещение на исходное
```

Добавим к гистограммам кривые плотности распределения: теоретическую плотность нормального закона и эмпирическую – для данной случайной величины.

```
par(mfrow = c(2, 1))

# строим гистограмму x, по вертикали откладываем вероятности
hist(x, freq = F, col = 'grey')
# добавляем плотность теоретического распределения с такими
#  же числовыми характеристиками, как у x
curve(dnorm(x, mean = mean(x), sd = sd(x)),
      col = "darkblue", lwd = 2, add = T)
# добавляем плотность фактического распределения x
lines(density(x), col = "red", lwd = 3)
# то же самое для y
hist(y, freq = FALSE, col = 'grey')
curve(dnorm(x, mean = mean(y), sd = sd(y)),
      col = "darkblue", lwd = 2, add = TRUE)
lines(density(y), col = "red", lwd = 3)       # Рисунок 10
par(mfrow = c(1, 1))
```

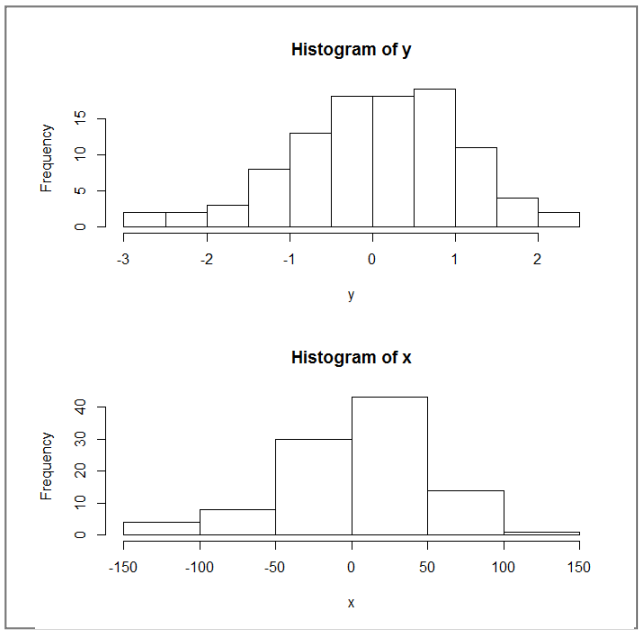


Рис. 9. Две гистограммы на одном полотне

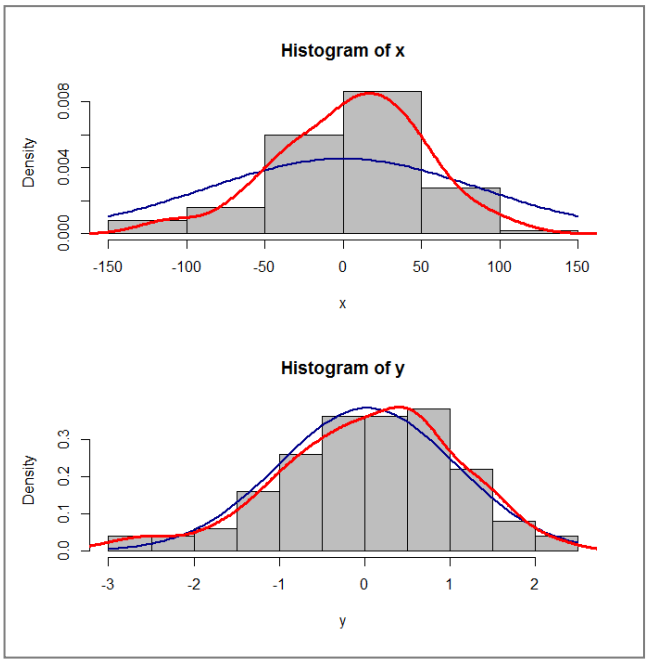


Рис. 10. Гистограммы с теоретической плотностью нормального закона (синяя кривая) и с фактической плотностью (красная кривая)

Чтобы сделать более сложный совместный график, воспользуемся функцией `layout()`.

```
# присоединяем столбцы фрейма к пространству имён
attach(dfLab)
# расположение графиков
layout(matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE))
# строим гистограммы
```

```
hist(Возраст)                # 1...
hist(Рост)                   # ...2...
hist(Вес)                    # ...3.                    Рисунок 11
```

Размеры участков диаграммы можно настраивать с помощью аргументов `widths` (вектор с шириной каждого участка) и `heights` (вектор с высотой каждого участка):

```
# расположение графиков с указанием ширины
layout(matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE),
       widths = c(3, 1), heights = c(1, 2))
hist(Возраст)                # строим графики: 1...
hist(Рост)                   # ...2...
hist(Вес)                    # ...3.                    Рисунок 12
```

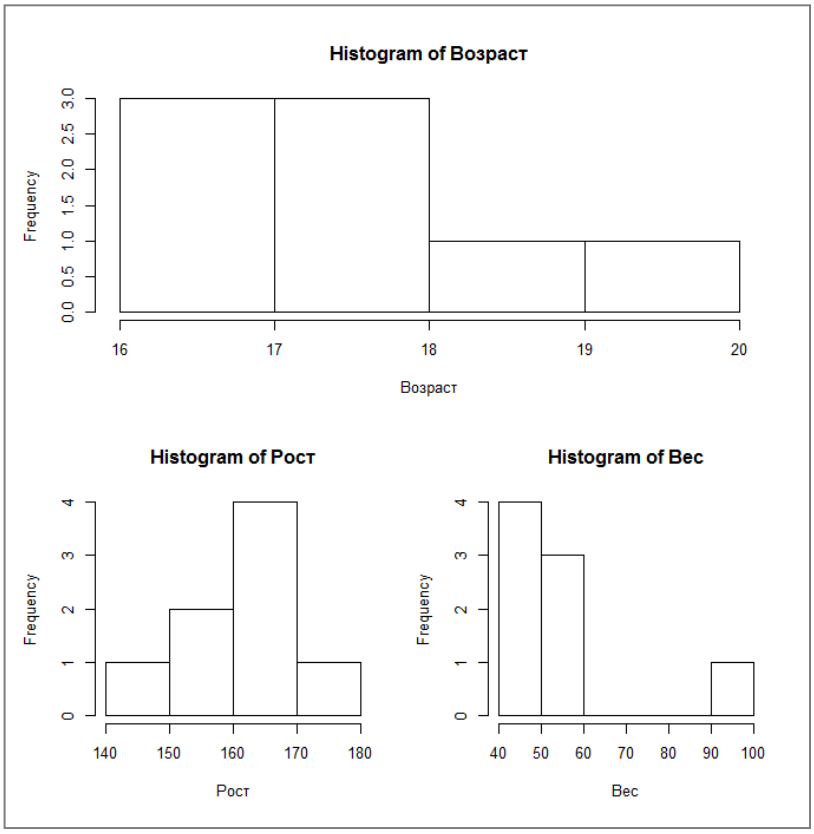


Рис. 11. Разбивка полотна графика на три части

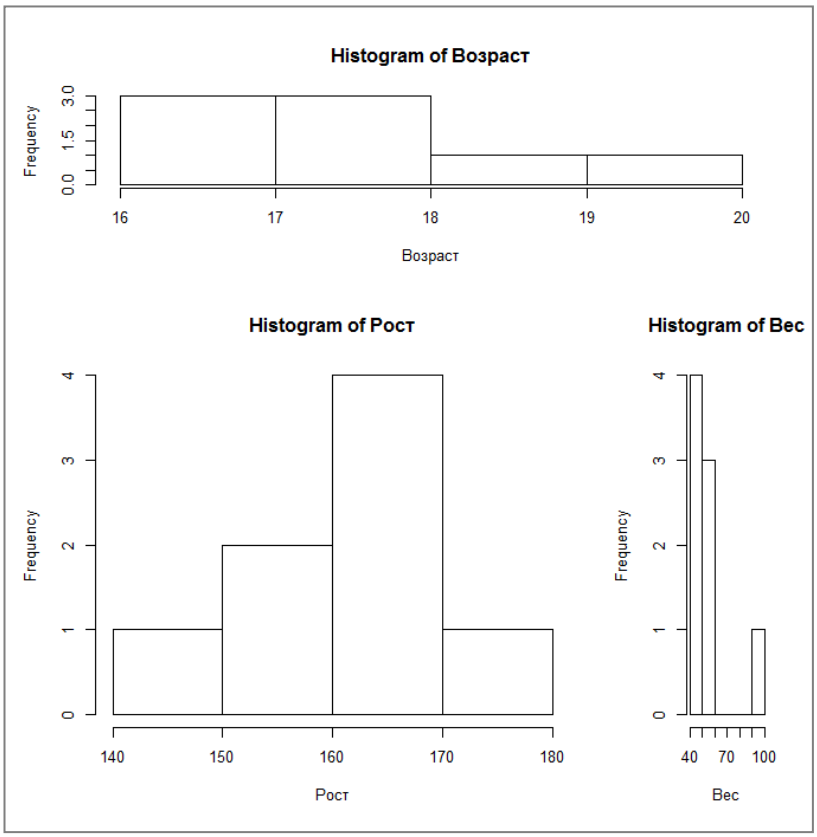


Рис. 12. Разбивка полотна графика с настройкой высоты и ширины

Не забываем отсоединять фрейм от пространства имён и сбрасывать значения графических параметров на исходные.

```
detach(dfLab)
par(mfrow = c(1, 1))
```

3.4 Сохраняем рисунок в файл

Изображение может быть сохранено в графический файл, например, в формате «png». Файл создаётся в рабочей директории.

```
# открыть поток вывода в файл pic01.png, фон прозрачный
png(filename = 'pic01.png', bg = 'transparent')

# дальше код создания графика
layout(matrix(c(1, 1, 2, 3), 2, 2, byrow = TRUE),
       widths = c(3, 1), heights = c(1, 2))
hist(Возраст)
hist(Рост)
hist(Вес)
# закрыть поток вывода в файл
dev.off()
```

3.5 Низкоуровневые графические функции и графические параметры

Функция `plot()` даёт R команду нарисовать новое поле для графика. Если указаны аргументы `x`, `y` они будут изображены на этом поле. Далее добавлять элементы на этот график можно с помощью графических функций низкого уровня:

- `abline(a, b | v | h)` – строит прямую по константе и коэффициенту (a, b), либо горизонтальную (h), либо вертикальную (v);
- `points(x, y)` – точки, с координатами x, y;
- `lines(x, y)` – ломаные: точки с координатами x, y соединяются отрезками;
- `arrows(x, y)` – стрелки: точки с координатами x, y соединяются стрелками;
- `text(x, y, label)` – пишет текст из вектора label в точках x, y на графике;
- `segments(x0, y0, x1, y1)` – несвязанные сегменты линий с координатами начала x0, y0 и координатами конца x1, y1;
- `rect(xbottom, ybottom, xtop, ytop)` – наносит на график прямоугольник(и) по вершинам: левой нижней и правой верхней;
- `polygon(x, y)` – рисует многоугольник по координатам x, y.

Все аргументы перечисленных функций векторные, это позволяет одним вызовом функции наносить на график несколько линий/сегментов/точек. Функции настройки элементов графика:

- `axis()` – настройка отображения оси;
- `mtext()` – пишет строку текста на границе графика;
- `legend()` – добавляет на график легенду.

Стоит также обратить внимание на графические параметры, которые отвечают за поля графика на полотне. На рисунке 13 красным выделена область графика, зелёным – границы графика, синим – внешние границы. Ширина границ задаётся графическими параметрами `mar` и `oma`. Например, чтобы убрать область внешних граница графика, необходимо с помощью функции `par()` присвоить параметру `oma` вектор из четырёх нулей.

NB: Стороны графика всегда нумеруются в одном и том же порядке: 1 – нижняя граница, 2 – левая граница, 3 – верхняя граница, 4 – правая граница.

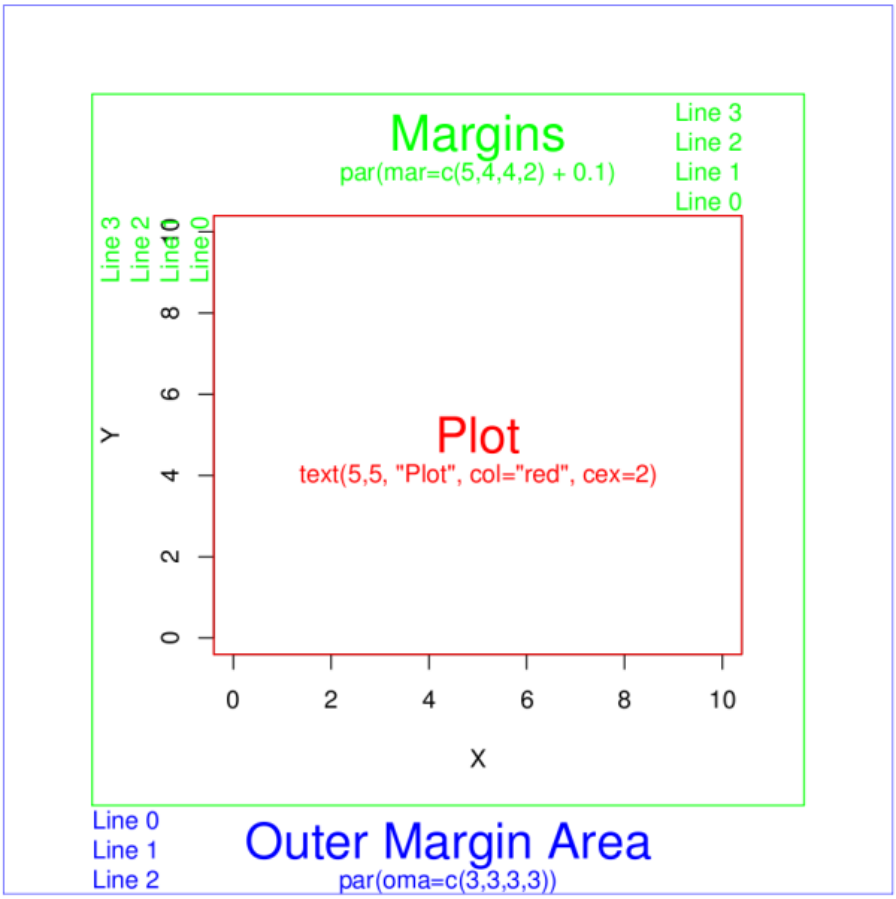


Рис. 13. Параметры полотна графика [12]

Следующий код демонстрирует использование некоторых перечисленных выше графических функций на данных о сотрудниках лаборатории.

```
# присоединяем фрейм
attach(dfLab)

# внешние границы полотна графика
par(mar = c(5, 4, 1, 1) + 0.1)

# рисуем пустой график
plot(Рост, Вес, type = 'n', axes = F, xlim = c(140, 185),
     ylim = c(40, 105),
     main = 'Рост и вес сотрудников лаборатории')

# номера маркеров наблюдений делаем из столбца пола
dots <- as.numeric(factor(Пол))
# наносим точки на график
points(Рост, Вес, pch = dots + 15, cex = 1.2)

# рисуем оси
axis(1, at = seq(140, 180, by = 10), pos = 40)
axis(2, at = seq(40, 100, by = 10), las = 2, pos = 140)
```

```
# стрелки на концах осей
arrows(x0 = c(180, 140), y0 = c(40, 100), x1 = c(185, 140),
       y1 = c(40, 105),  length = 0.15)

# легенда
legend(170, 100, pch = unique(dots) + 15, cex = 1.2,
       legend = c(unique(Пол)))

# подпишем сотрудника с максимальным весом
text(Рост[Вес == max(Вес)], max(Вес),
     labels = Имя[Вес == max(Вес)], pos = 2, font = 3)

# нижняя подпись
mtext('Источник данных: http://ru.steins-gate.wikia.com',
      side = 1, line = 4, adj = 1, cex = 0.8, font = 3)

# рамка вокруг полотна графика
box('figure')

# отсоединяем фрейм
detach(dfLab)
```

Результат показан на рисунке 14.

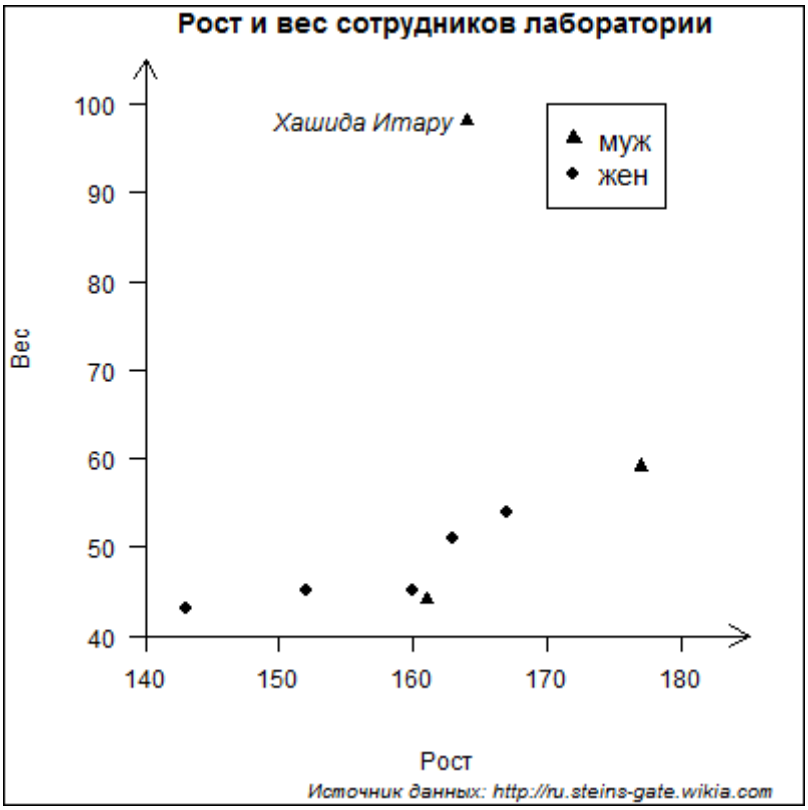


Рис. 14

4. Определение функции

R – функциональный язык программирования, в его основе лежит идея хранения кода в функциях, которые изменяют объекты. Это избавляет от необходимости сохранять все состояния программы. Пакеты, которые дополняют возможности базового R, состоят из функций. При загрузке пакета в оперативную память эти функции становятся доступны и позволяют нам выполнять вычислительные процедуры, строить модели и делать визуализации. И все эти пакеты когда-то, ещё до попадания в общедоступный архив CRAN, были пользовательскими функциями.

В предыдущих практиках мы выполняли код построчно, и при внесении корректировок приходилось прогонять участки кода заново. После отладки скрипта удобно объединять участки кода, которые выполняют различные этапы одной и той же задачи, в функции. Функция в R – это объект типа «function». Для вызова функции используется специальный оператор – круглые скобки. Даже если функция вызывается без аргументов, оператор круглых скобок необходим, чтобы отличить функцию от объекта другого типа. Создать функцию можно с помощью вызова функции `function()` [1]:

```
имя.функции <- function(аргументы) {действия}
```

Это уже знакомая конструкция: ранее с определением функции мы сталкивались, переназначая метод пользовательского класса. Теперь рассмотрим создание собственных функций на простых примерах.

4.1 Примеры простых функций

Рассмотрим пример функции `add()`, которая получает в качестве аргументов два числа (`x` и `y`) и возвращает их сумму. Тело функции заключено в фигурные скобки и состоит всего из одной строки – функции `return()`, которая возвращает сумму аргументов. Если явного вызова функции `return()` нет, будет возвращён результат последней строки функции.

```
# Функция сложения двух чисел
# объявление функции
add <- function(x, y) {
    return(x + y)}
add(1, 1)          # вызов функции
```

Теперь создадим функцию, которая вычисляет среднее геометрическое аргументов. Аргументом функции будет вектор `x`. В теле будет всего одна

строка, и `return()` на этот раз явно прописывать не будем. Формула для расчёта среднего геометрического (n – число элементов вектора x):

$$G(x) = \sqrt[n]{\prod_{i=1}^{n} x_i} \quad (1)$$

```
# Функция расчёта среднего геометрического
# объявление функции
sgeom <- function(x) {
      prod(x) ^ (1 / length(x))
}
sgeom(c(1, 3, 9))                # использование функции
### [1] 3
sgeom(c(1, 3, 9, -3))            # использование функции
### [1] NaN
```

В случае если в векторе `x` чётное количество элементов, а их произведение меньше нуля, функция возвращает `NaN`. Это означает, что полученное значение не является числом (в данном случае, рациональным). Усложним пример, добавив проверку знака аргумента функции.

4.2 Функция с проверкой условия

Проверка условия осуществляется конструкцией `if … else`. Если условие задаётся вектором, для проверки будет взят только первый элемент этого вектора, а в терминале появится соответствующее предупреждение. Выдавать промежуточные результаты будем с помощью функции `print()`, а функцией `warning()` печатать предупреждения.

```
# создадим вектор x из двух элементов
x <- c(12, 42)
# для проверки условия надо свести вектор к скаляру
if (all(x >= 0)) {
    print('x содержит только неотрицательные элементы')
} else {
    warning('среди элементов x есть отрицательные или 0')
}
### [1] "x содержит только неотрицательные элементы"
```

Обратите внимание на круглые скобки, в которых заключено проверяемое условие, и фигурные, в которых содержатся выполняемые

действия. Проверим, какой результат выдаст проверка условия, если в `x` будет отрицательное значение. Заметим, что предупреждения в терминале выводятся ярким шрифтом и без номеров элементов.

```
x <- c(12, 42, -3)
if (all(x >= 0)) {
    print('x содержит только неотрицательные элементы')
} else {
    warning('среди элементов x есть отрицательные или 0')
}
### Warning message:
### среди элементов x есть отрицательные или 0
```

Добавим проверку условия в функцию `sgeom()` и заставим её выводить результат, даже если он не является вещественным числом. В базовом R есть пакет «complex», в котором определены правила записи и вычислений комплексных чисел. Мнимая единица i обозначается как `1i`. В нашем случае, если произведение элементов отрицательно, а их количество чётно, результат – комплексное число, которое состоит только из мнимой части:

$$\left.\begin{array}{l}\prod_{i=1}^{n} x_i < 0, \\ n \text{ чётное.}\end{array}\right| \Rightarrow \quad G(x) = i \cdot \sqrt[n]{\left|\prod_{i=1}^{n} x_i\right|} \qquad (2)$$

```
# добавим проверку неотрицательности произведения аргументов
sgeom <- function(x) {
    if (prod(x) >= 0) {
        prod(x) ^ (1 / length(x))
    } else {
        warning('Комплексный результат')
        abs(prod(x)) ^ (1 / length(x)) * 1i
    }
}

sgeom(c(1, 3, 9, -3))           # использование функции
### [1] 0+3i
### Warning message:
### In sgeom(c(1, 3, 9, -3)) : Получен комплексный результат
```

4.3 Функции и пространства имён

В R действуют стандартные правила видимости объектов: каждая функция оперирует только своими аргументами и не имеет доступа к объектам глобального пространства имён. Убедимся в этом на примере работы функции `add()`. Введёт переменные z: одну в глобальном пространстве имён, одну – внутри функции, для хранения результата суммирования.

```
add <- function(x, y) {
    z <- x + y                # локальная переменная z
    return(z)
}

z <- 124                      # глобальная переменная z
```

Запуск функции `add()` не изменяет содержимого глобальной переменной:

```
add(1, 2)
### 3
z
### 124
```

Однако есть способ обратиться из функции к объекту в глобальном пространстве имён: это глобальные операторы присваивания `->>` и `<<-`. Их использование не приветствуется, поскольку нарушает логику выделения участков кода в функции и может привести к ошибкам.

```
add <- function(x, y) {
    z <<- x + y
    return(z)
}
z <- 124
add(1, 2)
### 3
z
### 3
```

На самом деле R умеет неявно обращаться из функции к глобальному пространству имён. Изменим add(), убрав из неё аргумент x, и зададим `x` глобально.

```
x <- 56                   # глобальная переменная x
add <- function(y) {      # убираем x из аргументов
    return(x + y)         # но упоминаем его в функции
```

```
}

add(2)
### [1] 58
```

Произошло следующее: не обнаружив `x` среди своих внутренних переменных, функция обратилась к пространству имён, из которого была вызвана, т.е. к глобальному. Там обнаружилась переменная с таким именем, и её значение было использовано для суммирования. Если переменная не найдена в глобальном пространстве имён, выполнение прервётся с ошибкой.

```
add <- function(y) {
    return(w + y)
}
add(2)
### [Error in add(2) : object 'w' not found
```

4.4 Функция идентификации аномальных наблюдений

Построим функцию для поиска аномальных наблюдений в данных, которая построит график разброса отклонений наблюдений относительно среднего, как на рисунке 15.

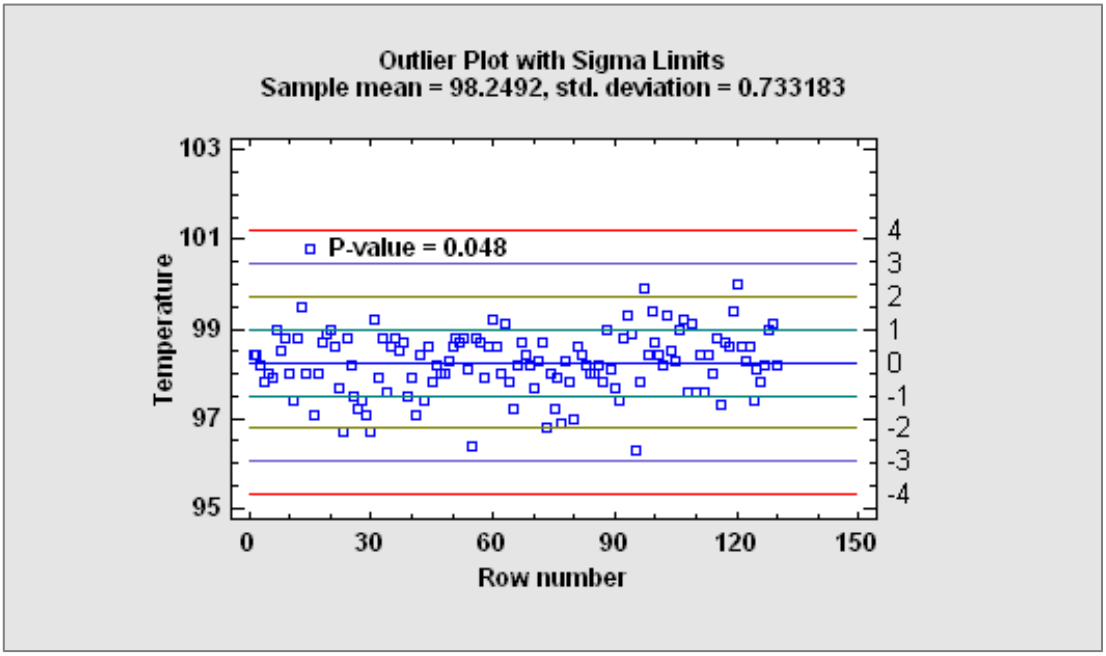


Рис. 15. График идентификации аномальных наблюдений из пакета Statgraphics

Ниже приводится текст функции, которая строит этот график и возвращает список аномальных наблюдений.

```
# Функция, которая строит график аномальных наблюдений.
# Аргументы:  x        — значения показателя,
#             x.labels — названия точек,
#             bound    - натуральное число, задаёт, при отклонении
#                        на сколько СКО от среднего считать наблюдение
#                        аномальным (значение по умолчанию: 3).
# Строит график и возвращает список меток аномальных наблюдений.
OutliersPlot <- function(x, x.labels, bound = 3) {

    # стандартизация значений показателя
    x <- scale(x)

    # генерируем номера объектов по порядку
    num.x <- seq_along(x)

    # Рисуем график: точки и заголовки осей.
    plot(num.x, x, ylim = c(-6, 6),
         xlab = 'Номер объекта',
         ylab = 'Разброс значений  показателя')
        # рисуем интервалы от -6 до +6 СКО
    abline(h = -6:6, col = c(2:7, grey(0.5), 7:2))
    # работаем с метками наблюдений
    #  После стандартизации среднее = 0, СКО = 1
    #  Тогда bound — граница отсечения: всё, что отклоняется
    #   от среднего меньше, чем на это число, НЕ аномалия,
    #   пишем там NA
    x.labels[abs(x - mean(x)) <= bound] <- NA

    # рисуем на том же графике метки всех наблюдений:
    #  если текста метки нет (NA), подпись не будет выведена
    #  на график. Метки остались только у аномальных.
    text(num.x, x, labels = x.labels, cex = 1.3)

    # В переменной result сохраняем итоговый список
    #  аномальных наблюдений. Функция na.omit() выбрасывает
    #  из вектора все значения NA.
    result <- data.frame(out.values = x[!is.na(x.labels)],
                         out.labels = na.omit(x.labels))

    # возвращаем список аномальных наблюдений
    return(result)
}
```

Запустив этот код, вы создадите объект типа «функция», и его имя (`OutliersPlot`) появится в списке объектов рабочего пространства. Чтобы запустить функцию, необходимо вызвать её с обязательными аргументами, перечислив их в круглых скобках. Приведём пример на условных данных.

```
# значения показателя
set.seed(123)
x <- rnorm(sd = 15, mean = 151, n = 19)
# создадим два аномальных значения
x[3] <- 256
x[7] <- -15.4

# метки наблюдений -- буквы английского алфавита
x.labels <- letters[1:19]

# запуск функции
OutliersPlot(x, x.labels)                    # Рисунок 16а
###   out.values out.labels
### 1  -3.394453          g
OutliersPlot(x, x.labels, bound = 2)         # Рисунок 16б
###   out.values out.labels
### 1   2.216678          c
### 2  -3.394453          g
```

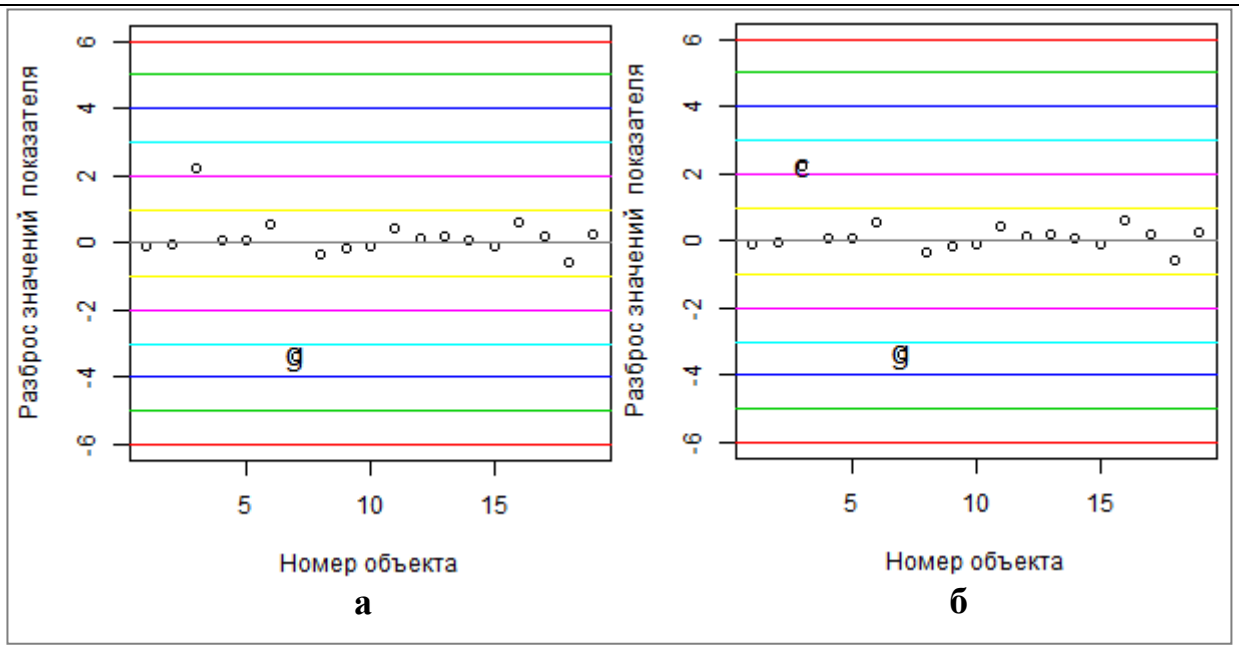


Рис. 16. Графики, которые строит функция по данным примера:
(а) – `bound = 3`; (б) – `bound = 2`

При этом функция не меняет исходные векторы `x` и `x.label`, в чём предлагается убедиться самостоятельно.

4.5 Преимущества векторизации по сравнению с циклом на примере

В функции `OutliersPlot()` нам пришлось сравнивать элементы вектора `x` с неким пороговым значением, и для этого мы воспользовались оператором квадратных скобок – применили векторизацию.

Стоит ещё раз напомнить, что в R следует забыть о переборе элементов внутри векторе с помощью цикла. Для примера возьмём вектор из пяти чисел и проверим условие: элемент вектора больше нуля. Вот так это делается в цикле:

```
x <- c(3, -6, 2, 7, -1)
for (i in 1:5) {
    if(x[i]<0) {
        print(x[i])
    }}
### [1] -6
### [1] -1
```

Эта конструкция понятна, но недопустима в R. Векторизация позволяет проверить условие в одно действие. Сравните:

```
# Само сравнение элементов с нулём.
x < 0
### [1] FALSE  TRUE FALSE FALSE  TRUE

# Теперь используем результаты сравнения
#  как логический вектор в квадратных скобках.
x[x < 0]
### [1] -6 -1
```

Так одна строка заменяет весь цикл. Стоит ли говорить, что такой подход даёт значительное преимущество при большом объёме данных:

```
# Случайная нормальная величина с СКО = 15
#  и средним = 0 (значение по умолчанию), 5000 наблюдений.
x <- rnorm(n = 5000, sd = 15)
#.....................................................
# ЦИКЛ
# Функция Sys.time() возвращает текущее время.
timer <- Sys.time()
for (i in 1:5000) {
    if(x[i]<0) {
       print(x[i])
    }}
```

```
### [1] -13.412656786
### [1] -10.503057153
<…>

Sys.time() - timer
Time difference of 2.430138 secs
#.....................................................
# ВЕКТОРИЗАЦИЯ
timer <- Sys.time()
x[x < 0]
### [1] -13.412656786 -10.503057153 -11.366034136 <…>
> Sys.time() - timer
Time difference of 0.03125 secs
```

Проверочные тесты

Для проверки своих знаний студентам предлагаются тестовые вопросы.

Выберите один верный вариант ответа

Вопрос 1. Какая из команд **не присвоит** переменной x значение 14?

А. `x == 14`

Б. `x <- 14`

В. `x = 14`

Г. `14 -> x`

Вопрос 2. К какому типу относится объект x, если:

```
> is.vector(x)
FALSE
> is.list(x)
TRUE
```

А. вектор

Б. список

В. скаляр

Г. матрица

Вопрос 3. Фрейм (таблица) данных в R – это:

А. вектор списков

Б. список векторов

В. матрица с заголовками

Г. файл .csv

Выберите все верные варианты ответа

Вопрос 4. Укажите верный вызов функции, который посчитает стандартное отклонение по строкам фрейма df (фрейм содержит только числовые столбцы):

А. `lapply(df, 1, sd)`

Б. `apply(df, 1, sd)`

В. `sapply(df, sd)`

Г. `sapply(as.data.frame(t(as.matrix(df))), sd)`

Вопрос 5. С помощью каких строк кода можно извлечь из фрейма данных (таблица ниже) только значение 13, если другие объекты не заданы?

```
> DF
   a        b     c
3 12 -0.8088404 3
4 13  1.2283235 4
5 14 -0.1095382 3
6 15  0.5304482 4
```

А. `DF[4, 1]`

Б. `DF["4", 1]`

В. `DF[2, 'a']`

Г. `DF[rownames(DF) == 4, "a"]`

Вопрос 6. Выберите верные определения пользовательской функции `myFun`, которая создаёт переменную x выводит её на экран.

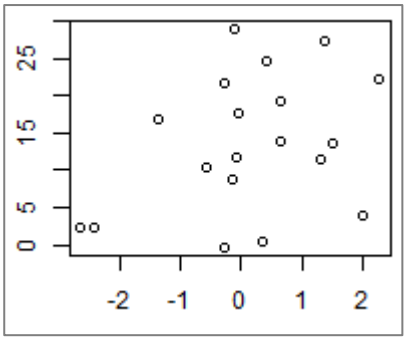
А.
```
myFun <- function () {x <- rnorm(15); show(x)}
```

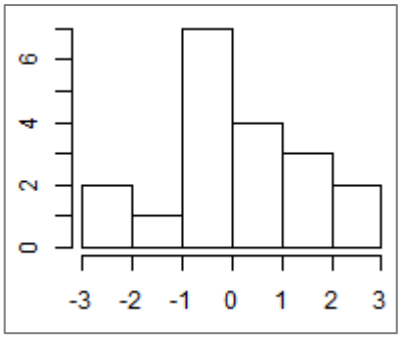
Б.
```
myFun <- function {
    x <- rnorm(15)
    show(x)}
```

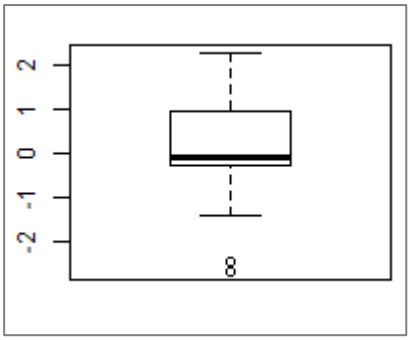
В.
```
myFun <- function ()x <- rnorm(15); show(x)
```

Г.
```
myFun <- function ()
    x <- rnorm(15)
    show(x)
```

Вопрос 7. Установите верные соответствия между командой и её результатом.







А. boxplot(x) **Б.** plot(x, y) **В.** hist(x)

Литература

Основная литература

1. *А.Б. Шипунов, Е.М. Балдин, П.А. Волкова, А.И. Коробейников, С.А. Назарова, С.В. Петров, В.Г. Суфиянов* Наглядная статистика. Используем R! – М.: ДМК пресс, 2012. – 298 с.
2. *Борис Демешев* Учебник по языку R для начинающих. URL: https://bdemeshev.github.io/r_manual_book/index.html (дата обращения: 01.09.2017)
3. *Мастицкий С.Э.*, *Шитиков В.К.* Статистический анализ в визуализация данных с помощью R. – М.: ДМК Пресс, 2015. – 496 с.: цв. ил.
4. *Роберт И. Кабаков* R в действии. Анализ и визуализация данных на языке R. – М.: ДМК Пресс, 2014. – 588 с.
5. Программирование в стиле R (руководство от Google), перевод на русский язык: https://sites.google.com/a/kiber-guu.ru/r-practice/links/R_style_from_Google.pdf?attredirects=0&d=1

Дополнительная литература

6. *John Fox* and *Milan Bouchet-Valat* Getting Started With the R Commander, версия 2.0-1 (последнее изменение: 8 ноября 2013). – 26 с.
7. *J H Maindonald* Using R for Data Analysis and Graphics . Introduction, Code and Commentary – URL: cran.r-project.org/doc/contrib/usingR.pdf, 2008. – 96 с.
8. *Earl F. Glynn*, Stowers Institute for Medical Research. R Colors. http://research.stowers-institute.org/efg/R/Color/Chart/ColorChart.pdf
9. *Patrick Burns* The R Inferno, version 30th April 2011. – URL: http://www.burns-stat.com/pages/Tutor/R_inferno.pdf, 2011. – 125 с.
10. Quick R: Graphical Parameters. URL: http://www.statmethods.net/advgraphs/parameters.html
11. *W. J. Owen* The R Guide version 2.5, Department of Mathematics and Computer Science University of Richmond, 2010 – 57 с.
12. R Graphics. A project from the Center for Limnology – UW Madison. URL: http://rgraphics.limnology.wisc.edu/rmargins_sf.php

Ответственность за сведения, представленные в издании, несет автор.

Учебное издание

СУЯЗОВА Светлана Андреевна

ВВЕДЕНИЕ В ЯЗЫК СТАТИСТИЧЕСКОЙ ОБРАБОТКИ ДАННЫХ R

Редактор *Н.А. Домнина*
Дизайн обложки *А.А. Николаева*
Компьютерная верстка и техническое редактирован *З. Кутумова*
Тематический план внутривузовских изданий ГУУ 2018 г.

Подп. в печ. 25.12.2018. Формат 60x90/16. Объем 4,25 п.л.
Бумага офисная. Печать цифровая. Гарнитура Times.
Уч.-изд. л. 2,13. Изд. № 118/2018. Тираж 500 экз. (1-й завод 50 экз.)
Заказ № 1141.

ФГБОУ ВО «Государственный университет управления»
Издательский дом ФГБОУ ВО ГУУ
109542, Москва, Рязанский проспект, 99, учебный корпус, ауд. 106
Тел./факс: (495) 371-95-10
e-mail: id@guu.ru, roguu115@gmail.com
www.id.guu.ru, www.guu.ru