Анализ данных с R.

© А. Б. Шипунов*, Е. М. Балдин†

*dactylorhiza@gmail.com
†e-mail: E.M.Baldin@inp.nsk.su

Эмблема **R** взята с официального сайта проекта `http://developer.r-project.org/Logo/`

Краткое оглавление

Оглавление

Глава 1

Введение в R

Вы ещё анализируете данные по старинке методом внимательного вглядывания в набор точек на графике, изредка подгоняя их к прямой с помощью многочисленных «мышекликов»? Пусть это делает за Вас компьютер, и **R** — это именно тот язык, который позволит разъяснить ему Вашу проблему.

1.1. Что такое R

Прежде всего **R** — язык программирования для статистической обработки данных и работы с графикой, но в тоже время это свободная программная среда с открытым исходным кодом, развиваемая в рамках проекта GNU. В любом дистрибутиве GNU/Linux, если его целью не является размещения всего дистрибутива на дискетке, можно найти эту среду[1].

R применяется везде, где нужна работа с данными. Это не только статистика в узком смысле слова, но и «первичный» анализ (графики, таблицы сопряжённости), и продвинутое математическое моделирование. **R** без особых проблем может использоваться и там, где сейчас принято использовать коммерческие программы анализа уровня MatLab/Octave. С другой стороны вполне естественно, что основная вычислительная мощь **R** лучше всего его проявляется при статистическом анализе: от вычисления средних величин до вейвлет-преобразований временных рядов.

География использования **R** очень разнообразна. Трудно найти американский или западноевропейский университет, где бы не работали бы с **R**. Очень многие серьёзные компании (например, Boeing) устанавливают **R** для работы. **R** для статистиков — это действительно глобально.

[1] В Debian GNU/Linux базовый пакет носит имя **r-base**, а подключаемые модули проще искать по акрониму «cran».

1.2. Немного истории

R возник как свободный аналог среды S-PLUS, которая в свою очередь является коммерческой реализацией языка расчётов S.

Язык S — довольно старая разработка (почти как TEX). Он возник ещё в 1976 году в компании Bell Labs, и был назван, естественно, «по мотивам» языка C. Первая реализация S была написана на FORTRAN и работала под управлением операционной системы GCOS. В 1980 г. реализация была переписана под UNIX, и с этого момента S стал распространяться, пока ещё в основном в научной среде. Начиная с третьей версии (1988 г.), коммерческая реализация S называется S-PLUS. Последняя в настоящее время распространяется компанией Insightful, и доступна под Windows и различные версии UNIX, естественно, за плату, причём весьма и весьма немаленькую[2]. Собственно говоря, именно высокая цена и сдерживала широкое распространение этого во многих отношениях замечательного продукта. Тут-то и начинается история **R**.

В августе 1993 г. двое молодых новозеландских учёных анонсировали свою новую разработку, которую они назвали **R**. По замыслу создателей (Robert Gentleman и Ross Ihaka), это должна была быть новая реализация языка S, отличающаяся от S-PLUS некоторыми деталями, например, обращением с глобальными и локальными переменными, а также работой с памятью. Фактически, они создали не полный аналог S-PLUS, а новую «ветку» на «дереве S». Многие вещи, которые отличают **R** от S-PLUS, связаны с влиянием языка Scheme[3].

Сначала проект развивался довольно медленно, но когда в нём появилось достаточно возможностей, в том числе уникальная по лёгкости система написания дополнений или пакетов, всё большее количество людей стало переходить с S-PLUS на **R**. Когда же, наконец, были устранены свойственные первым версиям проблемы с памятью, то среди пользователей **R** стали появляться и «любители» других статистических пакетов (прежде всего тех, которые имеют интерфейс командной строки: SAS, Stata, SYSTAT). Количество книг, написанных про **R**, за последние годы выросло в несколько раз, а количество пакетов уже приближается к полутора тысячам.

Идея центральной системы хранения и распространения пакетов — CRAN известного как Comprehensive R Archive Network (`http://cran.r-project.org/`) была заимствована из TEX-сообщества (CTAN, или Comprehensive TeX Archive Network; аналогичной схемой пользуется и Perl-сообщество: CPAN или Comprehensive Perl Archive Network). Все три упомянутых проекта объединяет одно: стабильная база и множество дополнений. В отличие от добавления новой функциональности в монолитную программу, качественный пакет может сравнительно легко написать один человек за вполне обозримый промежуток времени.

[2] Версия S-PLUS для UNIX стоит, например, $6500

[3] Scheme — это функциональный язык программирования. Один из наиболее популярных диалектов языка Lisp

1.3. Как скачать и установить R

Поскольку **R** — свободная система, то его можно скачать и установить совершенно свободно. Есть несколько способов, которые зависят от того, какая у Вас операционная система.

GNU/Linux Если у Вас какой-нибудь из распространённых дистрибутивов, то **R** наверняка входит в репозитарий «прилагающихся» к Вашей системе пакетов. Единственное, что нужно учесть — обновление пакетов часто отстаёт от выхода версий [4]. На момент написания статьи современной версией была 2.6.0.

Версия **R** нумеруется тремя числами, первые два — это главная версия, которая обновляется два раза в год. С каждой главной версией в **R** привносятся изменения, причём часто довольно значительные. Как правило, это множество новых команд, исправленные алгоритмы выполнения старых, и, разумеется, исправления ошибок. К недостаткам смены версии можно отнести возможные проблемы с обратной совместимостью. Естественно, разработчики стараются минимизировать такие изменения. С другой стороны, написанные на **R** программы пяти-семилетней давности, как правило, работают без проблем. В общем и целом, мораль такова: обновляйте **R** смело, но при этом всегда читайте список изменений.

На каждую главную версию выходит, как правило, две минорных версии (нулевая и первая). Первая минорная версия обычно ничего нового, кроме исправления ошибок, не содержит. Таким образом, если Вы хотите всегда иметь самую свежую версию, то репозитарий пакетов особенно в случае стабильных дистрибутивов не годится. В этом случае надо будет скачивать **R** из CRAN (`http://cran.r-project.org/`). У этого сайта довольно много зеркал, так что можно выбрать подходящее.

Компиляция **R** из исходного кода очень проста и не требует каких-то экзотических библиотек. Поэтому, если процесс компилирования не пугает, то собрать **R** из исходников не составит труда.

Mac OS X Для того чтобы начать работать с **R** в Mac OS X, надо сначала убедиться, что у Вас последняя версия этой операционной системы. Последние версии **R** выходят только под Mac OS 10.4.x («Tiger», версия под 10.5.x «Leopard» сейчас в разработке). Установочный пакет скачивается с CRAN. Имейте в виду, что графическая оболочка **R** для Mac («**R** Mac GUI») обновляется быстрее, чем сам **R**, поэтому разработчики предусмотрели возможность скачивания и установки оболочки отдельно. Установка происходит практически также, как установка любой программы под Mac.

[4] Цикл до выхода новой версии **R** длится примерно 3 месяца.

Ещё надо отметить, что **R** завоевал себе популярность не в последнюю очередь тем, что он запускался под Macintosh, в то время как S-PLUS под эту платформу был недоступен.

Windows Для того чтобы запускать **R**, можно иметь любые версии этой операционной системы, начиная с Windows 95. Как и в предыдущем случае, установочный пакет скачивается с CRAN. Опять-таки, установка напоминает обычную для среды Windows. Есть инсталлятор, который задаёт несколько вопросов, от ответов на которые ничего серьёзного не зависит. После инсталляции появляется ярлык, щёлкая на котором, можно запускать **R**.

Здесь интересно заметить, что Windows-инсталляция **R** является, как это принято сейчас говорить, «portable», и может запускаться, например, с USB-флешки или CD. **R** делает пару записей в реестр, но для работы они совершенно не критичны. Единственное, надо иметь в виду то, что рабочая папка должна быть открыта для записи, иначе некуда будет записывать результаты работы.

Есть одна, важная для всех операционных систем особенность: **R** (в отличие от того же S-PLUS) держит все свои вычисления в оперативной памяти, поэтому если в процессе работы, скажем, выключится питание, то результаты сессии, не записанные явным образом в файл, пропадут. Эта особенность, к сожалению, также не позволяет **R** работать с действительно большими объёмами (порядка сотен тысяч и более записей) данных, отдавая их на откуп гораздо менее удобной системе анализа ROOT[5] (`http://root.cern.ch`)

1.4. Как начать работать в R

Предполагается, что программная среда **R** уже установлена, поэтому приступим...

1.4.1. Запуск

Опять-таки, каждая операционная система имеет свои особенности работы. Но в целом можно сказать, что под все три упомянутые выше операционные системы существует так называемый «терминальный» способ запуска, а под Mac и Windows имеется и штатная GUI с некоторыми дополнительными возможностями (разными в разных ОС).

Терминальный способ прост: достаточно в командной строке набрать:

```
=> R
```

[5] В сентябрьском номер LInux Format за 2006 год была статья посвящённая ROOT. Сама статья доступна в открытом доступе по адресу `http://www.inp.nsk.su/~baldin/DataAnalysis/`

```
R version 2.4.0 Patched (2006-11-25 r39997)
Copyright (C) 2006 The R Foundation for Statistical
Computing ISBN 3-900051-07-0

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.

>
```

и появится приглашение в виде символа `>`. Теперь можно приступать к работе.
► Под Windows это немного сложнее — необходимо вызывать программу **R.exe**, причём для этого надо либо «находиться» в той же папке, где находится эта программа, либо путь к этой папке должен быть прописан в переменной PATH. Кроме того, для того чтобы в русскоязычной Windows вводный экран был читаем, надо сменить в окне терминала кодировку (`chcp 1251`) и поставить соответствующий шрифт (скажем, Lucida Console).

Если терминал запущен без графической среды, то все изображения будут «скидываться» в один многостраничный PostScript-файл `Rplots.ps`. Под Mac это будет происходить даже если X11 запущен, так что полноценно использовать **R** под Mac можно только в GUI-варианте. Терминальный запуск под Windows таких ограничений не имеет.

В дальнейшем договоримся, что под «сессией **R** » мы будем иметь в виду терминальный запуск под X11 в GNU/Linux и GUI-запуск в Windows и Mac. GUI под эти операционные системы построены так, что они всё равно запускают терминал-подобное окно. Общение с **R** возможно только в режиме диалога, «команда-ответ». Полноценного GUI[6] с **R** не поставляется, хотя существуют многочисленные попытки создать такую систему. Пока все эти попытки далеки от завершения. Возможно, это и к лучшему, так как система из меню-окошек-опций

[6]Скорее всего ближе всего подошла к рабочему состоянию программа **Rcommander**, о котором мы ещё поговорим.

не способна заменить полноценный интерфейс командной строки, особенно в случае таких сложных систем как **R**. Интересно, что S-PLUS имеет очень приличный GUI, но если открыть любой учебник по этой системе, то можно заметить, что автор настоятельно рекомендует пользоваться командной строкой.

► Для **R** GUI под Windows существует возможность запустить его в много- (MDI) и однооконном (SDI) режимах. Для использования настоятельно рекомендуется однооконный режим (SDI), тем более что все остальные реализации **R** только его и «умеют».

При запуске **R**, первым делом появляется вводный экран. Если выполнение программы производится в окружении с русской локалью, то стандартное введение будет выведено по русски. Если по какой-то причине требуется нелокализованный **R**, то в этом случае можно установить переменную LANGUAGE, а именно создать файл `Renviron.site`, в который внести строчку

```
LANGUAGE=en
```

Этот файл должен находиться в так называемой «домашней» директории **R**. Более подробно про это можно узнать, если прочитать помощь по команде `Startup`. Под Linux вызвать нелокализованный R другими способами, например, указав язык локали прямо в командной строке:

```
=> LANG=POSIX R
```

1.4.2. Первые шаги

Перед тем как начать работать, надо понять, как выйти. Для этого достаточно ввести одну команду и ответить на один вопрос:

```
> q()
> Save workspace image? [y/n/c]: n
```

Уже такой простой пример демонстрирует, что в **R** любая команда — это функция, которой можно передать аргумент. Даже если аргумент не указан, то скобки всё равно надо указать. Если этого не сделать, то вместо выхода из **R** на экран будет выведено определение функции:

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

Для того чтобы узнать как правильно вызывать функцию следует обучиться пользоваться встроенной справкой. Есть два пути. Первый — вызвать команду справки:

```
> help(q)
```

или

```
> ?q
```

Текст справки будет выведен в основном окне программы. Если внимательно прочитать текст, то становится ясно, что выйти из **R** можно и не отвечая на дополнительный вопрос, если ввести:

```
> q("no")
```

Зачем же нужен этот вопрос? Или, другими словами, что будет, если ответить положительно? В этом случае в рабочую папку **R** (ту, из которой он вызван), запишутся два файла: бинарный `.RData` и текстовый `.Rhistory`. Первый содержит все объекты, созданные за время сессии. Второй — полную историю введённых команд. **R** работает с историей команд стандартным образом: доступ к предыдущей команде осуществляется через клавишу-стрелку «вверх», а поиск в истории команд по комбинации `^r`. Если при выходе сохранить файл `.Rhistory`, то команды из этой сессии команды будут доступны и в следующей при условии, что **R** будет вызван из той же самой папки. И наоборот, если случайно сохранить рабочую среду (эти два файла), то при следующем старте они загрузятся автоматически. Иногда такое поведение **R** становится причиной различных недоумений, так что необходимо быть *внимательным*!

Итак, как войти и как выйти, уже понятно. Осталось ещё немного сказать про помощь. Её можно вызвать несколькими разными способами. Во-первых, с помощью команд `?` или `help()`, как написано выше. Во-вторых, можно вызвать команду `help.start()`. В этом случае откроется окно браузера, в котором будет демонстрироваться так называемая HTML-помощь. Основное её преимущество перед обычной текстовой помощью в том, что её разделы соединены гиперссылками (в Mac OS X такая помощь вызывается обычными командами). В третьих, вместе с **R** устанавливается несколько руководств в формате PDF. Их можно найти в папке, содержащей документацию Наконец, часто бывает нужна «обратная» помощь — Вы знаете, что Вы хотите, но не знаете, как это сделать (какую команду вызвать). В этом случае могут помочь две команды: `help.search()` и `apropos()`. Вот как их надо вызывать:

```
> help.search("vector")
```

Результатом исполнения команды будет список команд с кратким описанием их действия. Команда `apropos()` выдаст просто список команд, содержащих строку, которая была в кавычках. Кстати, обратите внимание на кавычки. Их надо обязательно использовать в этих командах. Кавычки можно использовать и в обычных командах помощи, например, `help("q")` или `?"q"`, причём иногда эти команды без кавычек просто не работают (например, нельзя получить справку по символу плюса, если ввести `?+`, надо вводить `?"+"`). Если ничего не помогает, и найти нужную функцию не удаётся, то приходится обращаться за справкой в Интернет или в список рассылки R-help.

1.4.3. Калькулятор-перекресток

Так назвал один из вводных разделов своей книги «Introduction to **R** » один из создателей системы, Peter Dalgaard. Это значит просто, что **R** можно использовать в том числе и как обычный калькулятор. Например:

```
> 3+2
[1] 5
> 16+3/5-11*8^2
[1] -687.4
> ((((16+3)/5)-11)*8)^2
[1] 3317.76
```

Для знакомых с интерактивными языками программирования типа Python здесь нет ничего особенно нового. Единственная любопытная деталь — это единичка в квадратных скобках. Она означает номер элемента вектора. Отсюда сразу два логических вывода:

1) **R** результат любой операции с числами трактует как вектор единичной длины. Скаляров в **R**, вообще говоря, просто нет;

2) Элементы векторов нумеруются с единицы, а не с нуля, как во многих языках программирования.

Для читателей, менее знакомых с программированием, отметим, что порядок арифметических действий в **R** стандартный, знакомый со школьной математики. Скобки (раскрывающиеся изнутри наружу) позволяют этот порядок действий менять:

```
> # Первый пример
> 3/7
[1] 0.4285714
> 3/7-0.4285714
[1] 2.857143e-08
> # Второй пример
> sqrt(2)*sqrt(2)
[1] 2
> (sqrt(2)*sqrt(2))-2
[1] 4.440892e-16
```

Эти примеры посложнее, кроме того, в них есть «подводные камни». Почему разность `3/7-0.4285714` не равна нулю, должно быть понятно всем знающим арифметику. **R** при выводе на консоль «невидимо» использует функцию `print()`, которая округляет бесконечную периодическую дробь 0,(428571). Второй пример интереснее. Произведение двух квадратных корней из двойки должно давать 2, что и происходит. Однако если вычесть из результата 2, получится какое-то

очень маленькое число ($4{,}440892 \times 10^{-16}$). Это происходит оттого, что вычисления выполняются на компьютере, который только притворяется, что работает с дробями, в то время как на самом деле оперирует только с целыми числами. Любая компьютерная система расчётов работает подобным образом, и с этим можно только смириться. Ещё один момент: приведённые примеры показывают, как можно пользоваться символом комментария (#).

И ещё немного о работе с аргументами на примере команды `round()` («округлить»). Она имеет два аргумента: число, которое нужно округлить, и значение `digits`, сообщающее, до какого знака округлять. Система аргументов работает разумно, так что все равно, что́ написать:

```
> round(1.5, digits=0)
[1] 2
> round(1.5, d=0)
[1] 2
> round(d=0, 1.5)
[1] 2
> round(1.5, 0)
[1] 2
> round(1.5,)
[1] 2
> round(1.5)
[1] 2
```

Это происходит благодаря тому, что есть значения аргументов по умолчанию. Например, в данном случае значение аргумента по умолчанию `digits` — «0». Об этом говорит и результат вывода команды `args()`:

```
> args(round)
> function (x, digits = 0)
```

Можно заметить также, что некоторые аргументы имеют имена. В результате аргументы можно вызывать не по порядку, а по именам. Имена же можно сокращать вплоть до одной буквы, но только если нет разных аргументов, которые от такого сокращения станут неразличимы. Можно вызывать аргументы по порядку, через запятую (не забудьте, что для десятичных дробей используется точка!), и тогда разрешается не использовать имён.

1.5. Скрипты

Просто открыть сессию **R** и вводить в окно программы команды, одну за другой — это лишь один из возможных способов работы. Гораздо более продуктивный метод, который является заодно и серьёзнейшим преимуществом **R** — это создание скриптов. Иначе говоря, программ, которые потом загружаются в **R** и

интерпретируются им. С самого начала работы следует создавать скрипты, даже для таких задач, которые кажутся пустяковыми — это в будущем значительно сэкономит Ваше бесценное время. Создание скриптов по любому поводу и даже без особого повода — одна из основ культуры работы в **R**.

Создать скрипт очень просто. Допустим, после открытия сессии была введена необходимая для получения искомого результата последовательность команд. Чтобы сделать свою работу воспроизводимой — надо просто сохранить историю команд. Лучше всего это сделать с помощью команды:

```
> savehistory(file="myscript.r")
```

После этого в текущей папке появляется файл `myscript.r`, который можно отредактировать любимом текстовом редакторе, а потом загрузить в **R** командой:

```
> source("myscript.r", echo=TRUE)
```

Опция `echo` добавлена для того чтобы можно было видеть сами команды, а не только результат их выполнения.

Есть ещё более эффективный способ работы: Вы открываете Ваш скрипт в текстовом редакторе, а потом посылаете отдельные его строки прямо в **R**. Есть несколько редакторов, которые умеют так делать. Во-первых, это **Emacs** с установленным пакетом **ESS** («Emacs Speaks Statistics»). Прелесть этой системы в том, что **R** запускается прямо в одном из окон редактора. Во-вторых, штатные **R** GUI под Windows и под Mac также имеют встроенные редакторы скриптов. К сожалению, Windows-редактор не подсвечивает синтаксис, и вообще довольно неудобен. Вместо него тем пользователям, которых пугает перспектива освоения **Emacs**, можно порекомендовать отличный редактор **Tinn-R**, специально «заточенный» под такую работу.

Скрипт **R** можно выполнить и не запуская интерактивную сессию. Для этого используются специальные опции командной строки. Вот так можно, например, выполнить наш скрипт-пример:

```
=> R --no-save < myscript.r > out
```

Опция `-no-save` говорит **R** не сохранять результаты сессии в файле истории `.RData/.Rhistory` (фактически, отвечает «no» на упомянутый выше заключительный вопрос).

1.6. Пакеты

Ещё одно важное преимущество **R** — наличие для него многочисленных расширений или пакетов буквально на все случаи жизни.

При установки **R** на компьютер, несколько пакетов уже в наличии: так называемые базовые пакеты, без которых система просто не работает (скажем, пакет,

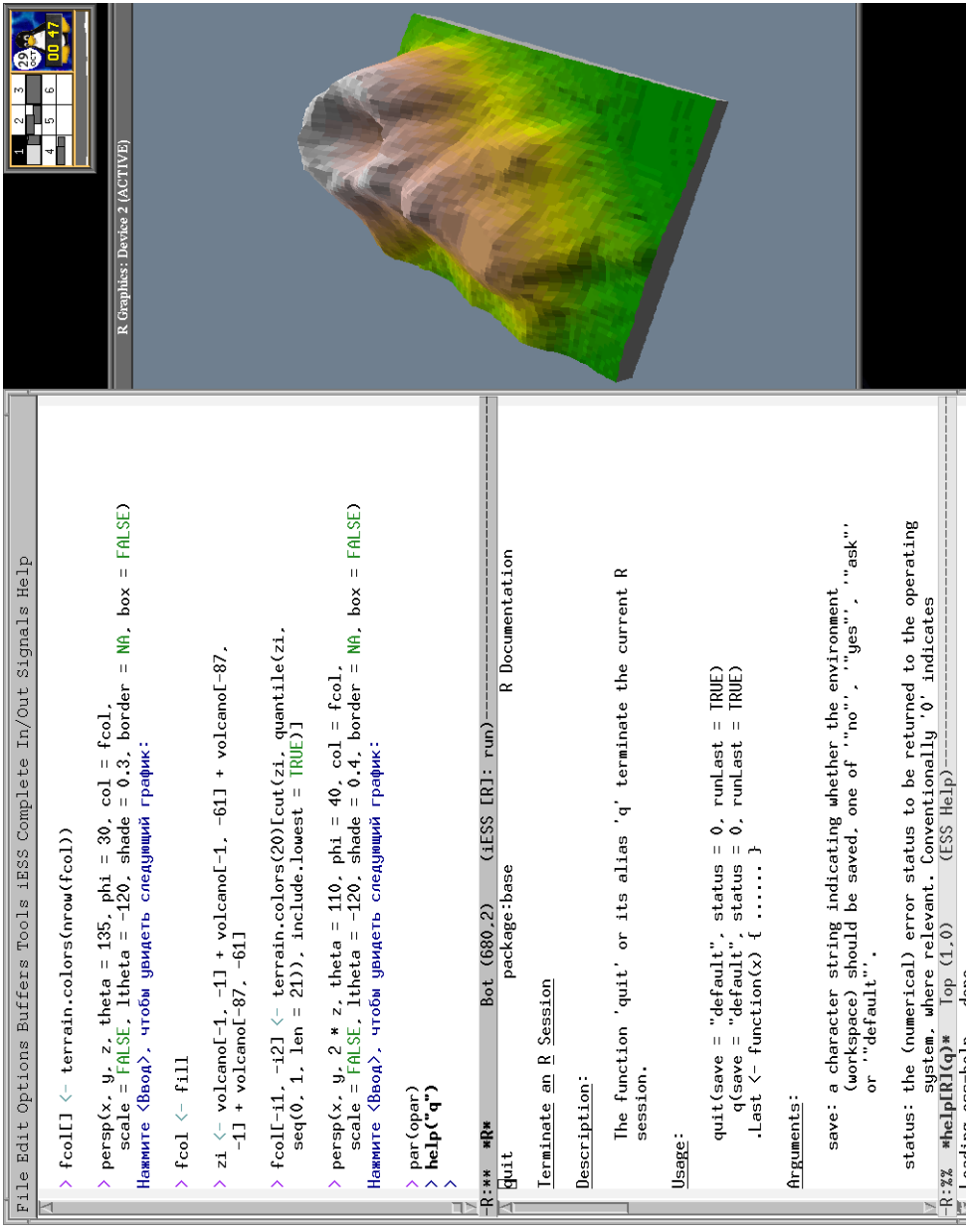

Рис. 1.1. emacs+ESS

который так и называется **base**, или пакет **grDevices**, который управляет выводом графиков), и ещё «рекомендованные» пакеты (пакет для специализированного кластерного анализа **cluster**, пакет для анализа нелинейных моделей **nlme** и пр.).

Кроме того можно поставить любой из почти полутора тысяч (!) доступных на CRAN пакетов. При доступном Интернете, это можно сделать прямо из **R** командой `install.packages()` (а под Mac и Windows есть соответствующие пункты в меню). Если соединение с сетью похуже, то можно скачать исходные тексты пакетов (под GNU/Linux) или скомпилированные пакеты (под Mac или Windows) и установить прямо с диска. После скачивания исходников пакета перед использованием сначала его нужно скомпилировать, потому что многие пакеты содержат код на FORTRAN или C. Для этого есть специальная форма вызова **R**:

```
=> R CMD INSTALL package.tar.gz
```

Естественно, это всё следует делать если **R** устанавливается не и из стандартного репозитария дистрибутива GNU/Linux. В случае стандартной установки можно поискать пакеты по сочетанию cran. В Debian GNU/Linux Etch таких пакетов ровно 85 — это не 1500 пакетов с CRAN, но скорее всего там уже есть многое из того, что необходимо.

Как только пакет установлен, то он сразу готов к работе. Нужно только инициализировать его перед употреблением. Для этого служит команда `library()`.

1.7. Полезные сайты, списки рассылки и документация

В заключение хотелось бы представить список самых полезных на наш взгляд сетевых ресурсов по **R**:

- `http://www.r-project.org/` — сайт проекта
- `http://cran.r-project.org/` — CRAN
- `https://stat.ethz.ch/pipermail/r-help/` — список рассылки R-help
- `http://finzi.psych.upenn.edu/nmz.html` — поиск в материалах по **R**
- `http://www.statmethods.net/index.html` — хороший справочный ресурс
- `http://zoonek2.free.fr/UNIX/48_R/all.html` — ещё один справочный ресурс
- `http://pj.freefaculty.org/R/Rtips.html` — советы по использованию **R**

Глава 2

Данные и графики

Анализ «хороших» данных — это просто. А вот чтобы сделать Ваши данные «хорошими», а затем и представить их — придётся попотеть.

2.1. R и работа с данными

Подготовка данных к работе — это одна из самых больших проблем для новичка в **R**. Сама по себе обработка данных подробно описана в разных руководствах и пособиях, а вот информация как добиться того, чтобы **R** прочитал приготовленные в другой программе данные, как правило, опускается. Почему это так очевидно: входные данные могут иметь слишком разный формат, чтобы написать по этому вопросу исчерпывающее и компактное руководство.

Данные можно представить в текстовом или в бинарном виде. Не вдаваясь в детали, примем, что текстовые данные — это данные, которые можно прочитать и отредактировать в текстовом редакторе (Emacs/Vi и прочее). Для того, чтобы отредактировать бинарные данные, как правило, нужна программа, которая эти данные произвела. Текстовые данные для статистической обработки — это текстовые таблицы, где каждая строка соответствует строчке таблицы, а колонки определяются при помощи разделителей. Обычно в качестве разделителей текстовых данных используются пробельные символы (пробел, табуляция и тому подобное), запятые или точки с запятой.

Первое что надо сделать перед чтением данных — это убедиться, что текущая директория в **R** и та директория, где находятся данные одно и то же. Для этого в запущенной сессии **R** надо ввести команду:

```
> getwd()
[1] "/home/username/"
```

Пусть это вовсе не та директория, в которой лежат данные. Поменять рабочую директорию можно командой:

```
> setwd("./workdir")
> getwd()
[1] "/home/username/workdir"
```

Как обычно, развёрнутую справку можно получить с помощью функции вызова справки `help(getwd)`. Далее следует проверить, а есть ли в текущей директории нужный файл:

```
> dir()
[1] "mydata.txt"
```

Вот теперь можно и загрузить данные. За чтение табличных текстовых данных отвечает команда `read.table()`:

```
> read.table("mydata.txt", sep=";", head=TRUE)
  a б в
1 1 2 3
2 4 5 6
3 7 8 9
```

Всё очень просто за исключением того, что перед чтением нужно знать в каком формате хранятся данные. То есть то, что у столбцов есть имена (`head=TRUE`) и разделителем является точка с запятой (`sep=";"`). Функция `read.table()` очень хороша, но не настолько умна, чтобы определять формат данных на лету. Чтобы посмотреть содержимое файла не выходя из **R**, можно воспользоваться функцией `file.show()`:

```
> file.show("mydata.txt")
a;б;в
1;2;3
4;5;6
7;8;9
```

В **R** многие команды, в том числе и `read.table()`, имеют для аргументов значения по умолчанию. Например, значение `sep` по умолчанию равно `""`. В данном случае это означает, что разделителем является любое количество пробелов или знаков табуляции, поэтому если данные вместо точек с запятыми разделены пробельными символами, то аргумент `sep` можно не указывать. Естественно, бывает безумное множество различных частных случаев, и сколько бы усилий не было приложено, всё не описать. Отметим, однако, ещё несколько важных моментов:

1) Файлы можно загружать и из других директорий, при этом можно использовать относительную адресацию:

```
> read.table("../workdir/mydata.txt")
```

2) Русский текст в файлах читается без проблем, если он набран в кодировке совпадающей с текущей локалью. Пусть локаль `ru_RU.UTF-8`, а сам файл закодирован в KOI8-R, тогда при его чтении следует воспользоваться функцией `file()`:

```
> read.table(
+  file("mydata-unicode.txt", encoding="KOI8-R"),
+                                sep=";", head=TRUE)
  а б в
1 1 2 3
2 4 5 6
3 7 8 9
```

3) Иногда нужно, чтобы **R** прочитал кроме имён столбцов ещё и имена строк. В этом случае в первой строке должно быть на одну колонку меньше, чем в теле таблицы (в данном примере три вместо четырёх):

```
> file.show("mydata2.txt")
а б в
раз 1 2 3
два 4 5 6
три 7 8 9
> read.table("mydata2.txt", head=TRUE)
    а б в
раз 1 2 3
два 4 5 6
три 7 8 9
```

4) По отечественным правилам в качестве десятичного разделителя нужно использовать запятую, а не точку. Если кто-то при подготовке исходных данных этим правилам последовал, то необходимо переопределить аргумент `dec`:

```
> read.table("mydata3.txt", dec=",", h=T)
      а   б   в
раз 1.1 2.2 3.3
два 4.4 5.0 6.0
три 7.0 8.0 9.0
```

Обратите внимание на сокращённое обозначение аргумента и его значения (`h=T`). Сокращать можно и нужно, но с осторожностью, поэтому в далее в этом тексте всегда будет именно `TRUE/FALSE`.

В целом, с текстовыми таблицами больших проблем не возникает. Разные экзотические текстовые форматы, как правило, можно преобразовать к «типичным» если не с помощью **R**, то с помощью каких-нибудь многочисленнейших текстовых утилит (вплоть до «тяжеловесов» типа языка Perl). А вот с «посторонними» бинарными форматами дело обстоит гораздо хуже. Здесь, прежде всего, возникают проблемы, связанные с полностью закрытыми форматами, например, такими как формат популярной в определённых кругах программы MS Excel. Вообще говоря, ответ на вопрос: «Как прочитать бинарный формат в **R**?» — часто сводится к совету по образцу известного анекдота: «выключим газ, выльем воду и вернёмся к условию предыдущей задачи». То есть надо найти способ, который позволит преобразовать бинарные данные в обычные текстовые таблицы. Проблем на этом пути возникает обычно не слишком много, но уж больно они разнообразные.

Второй путь — это найти способ прочитать данные в **R** без преобразования. Специально для этих целей в **R** есть пакет **foreign**, который может читать бинарные данные, выводимые пакетами Minitab, S, SAS, SPSS, Stata, Systat, а также формат DBF. Чтобы узнать подробнее об определённых в этом пакете командах, надо загрузить пакет и вызвать общую справку:

```
> library(foreign)
> help(package=foreign)
```

Что же касается всё того же пресловутого формата Excel, то здесь дело хуже. Есть не меньше пяти разных способов, как загружать в **R** эти файлы, но все они имеют ограничения. К тому же новый формат MS Excel 2007 пока вообще не поддерживается. Из всех способов наиболее привлекательным представляется обмен с **R** через буфер. Если открыть в OpenOffice Calc xls-файл , то можно скопировать в буфер любое количество ячеек, а потом загрузить их в **R**:

```
> read.table("clipboard")
```

Это очень просто, и главное, работает с любой Excel-подобной программой.

Тут следует отметить ещё одну вещь: *ни в коем случае не рекомендуется производить какой-либо статистический анализ в программах электронных таблиц.* Не говоря уже о том, что интернет просто забит статьями об ошибках в этих программах и/или в их статистических модулях, это ещё и крайне неверно идеологически. Иначе говоря:

Используйте R!

Добавим ещё несколько деталей:

1) **R** может загружать изображения. Для этого есть несколько пакетов. Наиболее продвинутый из них — это пакет **pixmap**. **R** также может загружать карты в формате ArcInfo и др. (пакеты **maps**, **maptools**) и вообще много чего ещё.

2) У **R** есть собственный бинарный формат. Он быстро записывается и быстро загружается, но его нельзя использовать для передачи данных.

```
> x <- "яблоко"
> save(x, file="x.rd")
> rm(x)
> x
Ошибка: объект "x" не найден
> dir()
[1] "x.rd"
> load("x.rd")
> x
[1] "яблоко"
```

Для сохранения и загрузки бинарных файлов служат команды `save()` и `load()`, для создания объекта — `<-`, а для удаления — `rm()`.

3) Для **R** написано множество интерфейсов к базам данных, в частности, для MySQL, PostgresSQL и SQLite (последний может вызываться прямо из **R**, см. пакеты **RSQLite** и **sqldf**).

4) Наконец, **R** сам может записывать таблицы и другие результаты обработки данных, и, разумеется, графики. Об этом мы поговорим ниже.

2.2. Графики

Несмотря на то, что «настоящие» статистики часто относятся к графикам почти с презрением, для «широких масс» одним из основных достоинств **R** служит именно удивительное разнообразие типов графиков, которые он может построить. **R** в этом смысле — один из рекордсменов. В базовом наборе есть несколько десятков типов графиков, ещё больше в рекомендуемом пакете **lattice**, и, естественно, намного больше в пакетах с CRAN. По оценочным прикидкам получается, что разнообразных типов графиков в **R** никак не меньше тысячи. При этом они все ещё достаточно хорошо настраиваются, то есть пользователь при желании достаточно легко может разнообразить эту исходную тысячу на свой вкус.

2.2.1. Два типа графических команд

Для правильного отображения кириллицы в X-окне (если это действительно необходимо) для начала следует правильно указать шрифты, например так:

```
> X11(fonts = c(
+   "-rfx-helvetica-%s-%s-*-*-%d-*-*-*-*-*-koi8-r",
```

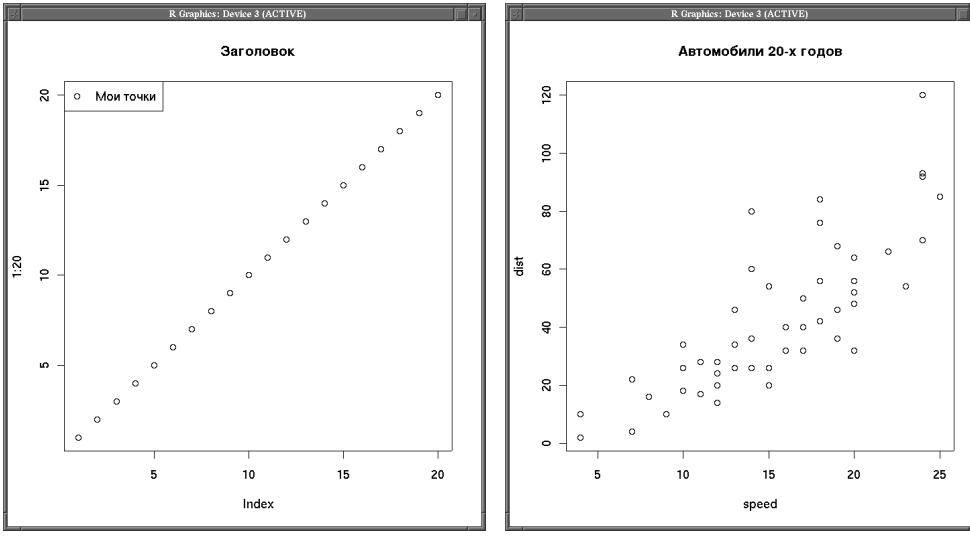

Рис. 2.1. Окна X11 с простейшим графиком

```
+      "-adobe-symbol-medium-r-*-*-%d-*-*-*-*-*-*-*-*"))
```

Опция `fonts` команды `X11` принимает вектор из двух элементов, который формируется с помощью команды `c()`. Первый элемент соответствует шрифту, используемому для обычных текстовых меток, а второй для отображения стандартных спецсимволов, например, греческого алфавита. С помощью программы **xfontsel** можно подобрать себе шрифт по вкусу. Подробности в разделе Fonts в документации, выдаваемой по `help(X11)`.

Рассмотрим пример:

```
> plot(1:20, main="Заголовок")
> legend("topleft", pch=1, legend="Мои␣точки")
```

Тут много такого, о чём пока речи не шло. Но самое главное это то, что первая команда рисует график «с нуля», тогда как вторая только добавляет к уже нарисованному графику детали. Это и есть два типа графических команд, используемых в базовом графическом наборе **R**. Теперь немного подробнее: `plot()` — основная графическая команда, причём команда «умная» (правильнее сказать «generic», или общая). Это значит, что она распознаёт тип объекта, который подлежит рисованию, и строит соответствующий график. Например, в приведённом примере `1:20` — это последовательность чисел от 1 до 20, то есть вектор, а для одиночного вектора предусмотрен график, где по оси абсцисс — индексы (номера каждого элемента вектора по порядку), а по оси ординат — сами эти элементы.

Если в аргументе команды будет что-то другое, то будет построен иной, более подходящий для этого объекта, график. Вот пример:

```
> plot(cars)
> title(main="Автомобили␣20-х␣годов")
```

Здесь тоже команды обоих типов, оформленные немного иначе. Не беда, что мы забыли дать заголовок в команде `plot()`, так как его всегда можно добавить потом, командой `title()`. «cars»[1] — это встроенная в **R** таблица данных, которая использована здесь по прямому назначению, то есть для демонстрации возможностей программы. Для нас сейчас важно, что это — не вектор, а таблица из двух колонок: speed и distance (скорость и тормозная дистанция). Функция `plot()` автоматически нарисовала, так называемый, scatterplot, когда по оси X откладывается значение одной переменной (колонки), а по оси Y — другой, и ещё присвоила осям имена этих колонок. Любопытным советуем проверить, что нарисует `plot()`, если ему «подложить» таблицу с тремя колонками, скажем, встроенную таблицу «trees». Кстати говоря, узнать, какие ещё есть встроенные таблицы, можно с помощью команды `data()` (именно так, без аргументов).

2.2.2. Графические устройства

Когда вводится команда `plot()`, **R** открывает, так называемое, экранное графическое устройство[2] и начинает вывод на него. Если следующая команда того же типа, то есть не добавляющая, то **R** «сотрёт» старое изображение и начнёт выводить новое в этом же окне. Если ввести команду:

```
> dev.off()
```

то **R** закроет графическое окно, что, впрочем, можно сделать, просто щёлкнув по кнопке закрытия окна. Экранных устройств в **R** предусмотрено несколько, в каждой операционной системе своё (а в Mac OS X даже два). Но всё это не так важно, пока не захочется строить графики и сохранять их в виде графических файлов. В этом случае придётся познакомиться с другими графическими устройствами. Их несколько (количество опять-таки зависит от операционной системы), а пакеты предоставляют ещё около десятка. Работают они примерно так:

```
> png(file="1-20.png", bg="transparent")
> plot(1:20)
> dev.off()
```

Команда `png()` открывает одноимённое графическое устройство, причём задаётся параметр, включающий прозрачность базового фона (удобно, например,

[1] Прочитать, что такое «cars», можно, вызвав справку обычным образом ("?cars").
[2] В случае использования X Window — это стандартное окно X11.

для Web). Такого параметра у экранных устройств нет. Как только вводится команда `dev.off()`, устройство закрывается и на диске появляется файл `1-20.png`. `png()` — одно из самых распространённых устройств при записи файлов. Недостатком его является, разумеется, растровая природа этого формата. Аналогичным по своей функциональности является и устройство `jpeg()`, которое производит jpeg-файлы.

R поддерживает и векторные форматы, например, PDF. Здесь, однако, могут возникнуть специфические для русскоязычного пользователя трудности со шрифтами. Остановимся на этом чуть подробнее. Вот как надо «правильно» создавать PDF-файл, содержащий русский текст:

```
> pdf("1-20.pdf",family="NimbusSan",encoding="KOI8-R.enc")
> plot(1:20, main="Заголовок")
> dev.off()
> embedFonts("1-20.pdf")
```

Как видим, требуется указать, какой шрифт мы будем использовать, а также кодировку, с которой работаем. Другие доступные однобайтные кириллические кодировки, идущие с **R**: `CP1251.enc` и `KOI8-U.enc`. Затем нужно закрыть графическое устройство и *встроить в полученный файл шрифты* с помощью команды `embedFonts()`. Следует отметить, что шрифт NimbusSan и возможность встраивания шрифтов командой обеспечивается взаимодействием **R** со сторонней программой Ghostscript, в поставку которой входят шрифты, содержащие русские буквы. Кроме PDF, **R** «знает» и другие векторные форматы, например, PostScript, xfig и picTeX. Есть отдельный пакет **RSvgDevice**, который поддерживает популярный векторный формат SVG. График в этом формате можно, например, открыть и видоизменить в свободном векторном редакторе Inkscape.

2.2.3. Графические опции

Как уже говорилось, графика в **R** настраивается в очень широких пределах. Один из способов настройки — это видоизменение графических опций, встроенных в **R**. Вот, к примеру, распространённая задача: нарисовать две гистограммы одну под другой на одном рисунке. Чтобы это сделать, надо изменить исходные опции, а именно разделить пространство рисунка на две части, примерно так:

```
> # Создаётся eps-файл размером 6 на 6 дюймов
> postscript("2hist.eps",width=6.0,height=6.0,
+    horizontal=FALSE,onefile=FALSE,paper="special")
> # Изменяется одно из значений по умолчанию
> old.par <- par(mfrow=c(2,1))
> hist(cars$speed)
> hist(cars$dist)
> # Восстанавливаем старое значение по умолчанию
```

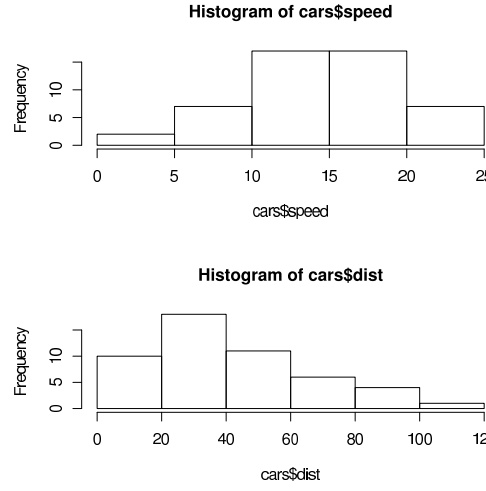

Рис. 2.2. Две гистограммы друг под другом

```
> par(old.par)
> dev.off()
```

Ключевая команда здесь `par()` — изменяется один из её параметров, `mfrow`, который регулирует сколько изображений и как будет размещено на «листе». Значение `mfrow` по умолчанию — `c(1,1)`, то есть один график по вертикали и один по горизонтали. Чтобы не печатать каждый раз команду `par()` без аргументов (для того чтобы выяснить умалчиваемые значения каждого из 71 параметра), мы «запомнили» старое значение в объекте `old.par`, а в конце вернули состояние к запомненному. То, что команда `hist()` строит гистограмму, очевидно из названия.

2.2.4. Идеологически верная графика

Несмотря на своё разнообразие, графическая система в **R** построена на основе строгих правил. Выбор типа графика, основных цветов и символов для изображения точек, расположение подписей и т. д. был тщательно продумано создателями. Одним из ключевых для **R** исследований является книга Уильяма Кливленда «Элементы графической обработки данных». Многие его идеи были осуществлены именно в S-PLUS, а затем и в **R**. Например, Кливленд нашёл, что традиционные «столбчатые» графики очень плохо воспринимаются, особенно когда речь идёт о близких значениях данных, и предложил им на замену: «точечные диаграммы». Вот так они реализованы в **R**:

```
> dotchart(Titanic[,,"Adult","No"],
```

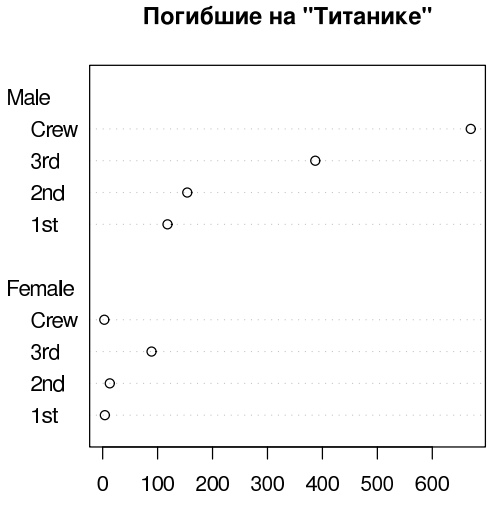

Рис. 2.3. Точечная диаграмма

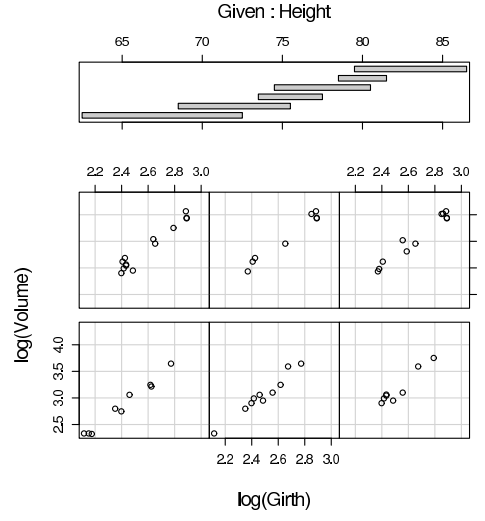

Рис. 2.4. График-решётка

```
+                    main='Погибшие на "Титанике"')
```

Встроенная таблица данных Titanic — это четырёхмерная матрица, которая отражает статистику по возрастным группам, типу билета и полу.

Особенно активно Кливленд (и далеко не только он) возражал против использования трёхмерных графиков и так называемых «пирогов». Поначалу «пирожных» графиков в **R** вовсе не было[3], причём по принципиальным соображениям. Трёхмерных графиков в **R** и сейчас немного (правда, есть особый пакет **rgl**, который позволяет строить такие графики на базе OpenGL), а если Вы хотите узнать, как меняется поведение двух переменных по отношению к третьей, **R** предлагает так называемые «Trellis graphs» или графики-решётки:

```
> coplot(log(Volume) ~ log(Girth) | Height, data = trees)
```

При выполнении этой команды на рисунке отображается как зависит объём древесины от объёма кроны (в логарифмической шкале) у деревьев различной высоты. Действительно, такое представление гораздо эффективнее трёхмерного. Странно, что распространённые пакеты статобработки почти не используют графики-решётки, хотя их наличие неоднократно называлось одной из главных причин коммерческого успеха S-PLUS.

[3]Сейчас они есть, но если Вы откроете страницу помощи, то узнаете, что «Pie charts are a very bad way of displaying information».

2.2.5. Интерактивность

Интерактивная графика позволяет выяснить, где именно на графике расположены нужные Вам точки и разместить объект (скажем, подпись) в нужное место, а также проследить «судьбу» одних и тех же точек на разных графиках. Кроме того, если данные многомерные, то можно вращать облако точек в плоскости разных переменных с тем чтобы выяснить структуру данных.

Ещё несколько лет назад пришлось бы написать, что здесь вместо **R** следует воспользоваться другими аналитическими инструментами, но **R** развивается так быстро, что все эти методы теперь доступны, причём в нескольких вариантах. Например, так можно добавлять подписи в указанную мышкой область графика:

```
> plot(1:20)
> text(locator(), "Моя␣любимая␣точка", pos=4)
```

После того как введена вторая команда, надо щёлкнуть левой кнопкой мыши на выбранной точке в графике, а затем уже без разницы где щёлкнуть правой кнопкой мыши.

Интерактивная графика других типов реализована командой `identify()`, а также пакетами **rggobi**, **TeachingDemos** и **iplot**.

2.3. Как сохранять результаты

Начинающие работу с **R** обычно просто копируют результаты работы (скажем, данные тестов) из консоли **R** в текстовый файл. И действительно, на первых порах этого может оказаться достаточно. Однако рано или поздно возникает необходимость сохранять объёмные объекты (например, таблицы данных), созданные в течении работы. Можно использовать уже упомянутый в начале статьи внутренний бинарный формат, но это не всегда удобно. Лучше всего сохранять таблицы данных в виде текстовых таблиц, которые потом можно будет открывать другими приложениями или текстовыми редакторами. Для этого служит команда `write.table()`:

```
> write.table(file="trees.csv", trees,
+             row.names=F, sep=";", quote=F)
```

В текущую рабочую директорию будет записан файл `trees.csv`, образованный из встроенной в **R** таблицы данных trees. А что, если надо записать во внешний файл результаты выполнения команд? В этом случае используется команда `sink()`:

```
> sink("1.txt", split=T)
> 2+2
[1] 4
> sink()
```

В этом случае во внешний файл запишется строчка «[1] 4», то есть результат выполнения команды. Сама команда записана не будет, а если хочется, чтобы она была записана, то придётся ввести что-то вроде:

```
> print("2+2")
[1] "2+2"
> 2+2
[1] 4
```

то есть повторять каждую команду два раза. Для сохранения истории команд служит команда `savehistory()`, а для сохранения всех созданных объектов — `save.image()`. Последняя может оказаться также полезной для сохранения промежуточных результатов работы, если не уверены в стабильности работы компьютера.

2.4. Мастера отчётов

Таблицы, созданные в **R**, можно сохранять и в более «приличном» виде, например, в форматах LaTeX ([1]) или HTML, при помощи пакета **xtable**. Естественно, хочется пойти дальше, и сохранять в каком-нибудь из этих форматов вообще всю **R** -сессию. Для HTML такое возможно, если использовать пакет **R2HTML** с CRAN:

```
> library(R2HTML)
> dir.create("example")
> HTMLStart("example")
HTML> 2+2
HTML> plot(1:20)
HTML> HTMLplot()
HTML> HTMLStop()
>
```

В рабочей директории будет создана поддиректория `example` и туда будут записаны HTML-файлы, содержащие полный отчёт о текущей сессии, в том числе и созданный график.

Можно пойти и ещё дальше. Что, если создать файл, который будет содержать код **R**, перемешанный с текстовыми комментариями, и потом «скормить» этот файл **R** так, чтобы фрагменты кода заменились на результат их исполнения? Идея эта называется «literate programming» (грамотное программирование) и принадлежит Дональду Кнуту, создателю TeX. В случае **R** такая система используется для автоматической генерации отчётов — «фичи», которая фактически отсутствует в остальных статистических пакетах и делает **R** поистине незаменимым. Для создания подобного отчёта, для начала, надо набрать простой файл с LaTeX-подобной структурой и назвать его, например, `test-Sweave.Rnw`:

```
\documentclass[a4paper,12pt]{scrartcl}
% Стандартная шапка для \LaTeX-документа
\usepackage[T2A]{fontenc}
% В зависимости от используемой локали вместо utf8 нужно
%поставить cp1251 или koi8-r
\usepackage[utf8]{inputenc}
\usepackage[english,russian]{babel}
\usepackage{indentfirst}

\title{Тест Sweave}
\author{A.B.\,Top}
\begin{document} % Тело документа
\maketitle

\textsf{R} как калькулятор:
<<echo=TRUE,print=TRUE>>=
1 + 1
1 + pi
sin(pi/2)
@

Картинка:
<<fig=TRUE>>=
plot(1:20)
@

\end{document}
```

Затем этот файл необходимо обработать в **R**:

```
> Sweave("test-Sweave.Rnw")
Writing to file test-Sweave.tex
Processing code chunks ...
 1 : echo print term verbatim
 2 : echo term verbatim eps pdf

You can now run LaTeX on 'test-Sweave.tex'
```

При этом создаётся готовый LaTeX-файл `test-Sweave.tex`. И, наконец, при помощи **latex**/**dvips** или **pdflatex** получить результирующий файл:

```
=> latex test-Sweave.tex
=> dvips test-Sweave.dvi
=> gv test-Sweave.ps
```

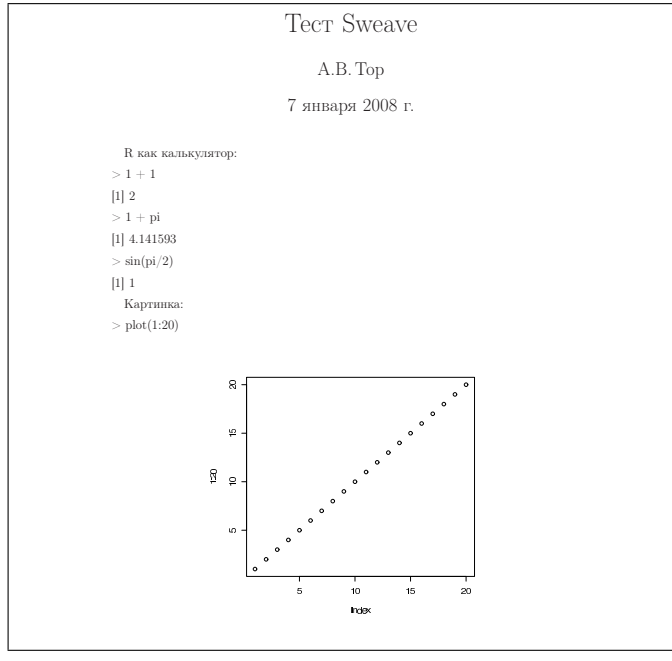

Рис. 2.5. Пример отчёта полученного с помощью команды Sweave

```
# или
=> pdflatex test-Sweave.tex
=> acroread test-Sweave.pdf
```

Такой отчёт можно расширять, шлифовать, изменять исходные данные, и при этом усилия по оформлению практически сводятся на нет. Если есть желание, чтобы код **R** набирался моноширинным шрифтом, то в LaTeX-преамбуле Rnw-файла следует добавить строчку:

```
\usepackage[noae]{Sweave}
```

Исходный код и авторскую документацию профессора Фридриха Лайша (Friedrich Leisch) можно найти здесь: `http://www.ci.tuwien.ac.at/~leisch/Sweave/`.

Есть и другие системы генерации отчётов, например, уже упомянутый пакет **R2HTML** умеет производить похожие отчёты в HTML. Есть пакет **brew**, который позволяет создавать автоматические отчёты в текстовой форме (разумеется, без графиков), и совсем новый пакет **odfWeave**, который может работать с ODF (формат OpenOffice.org).

Глава 3

Типы данных в R и принципы работы с ними

Теперь, когда читатель знает достаточно для того чтобы суметь загрузить данные в **R**, рассмотрим, что же происходит с ними внутри системы, и как, собственно, работать с этими в данными.

3.1. Форматы данных

С точки зрения статистики, данные принято делить на типы в зависимости от того, насколько близко их можно представить при помощи известной метафоры числовой прямой. Например, возраст человека легко представить таким образом, за тем исключением, что он не может быть отрицательным. Размер ботинок представить так уже сложнее, поскольку между двумя соседним размерами, как правило, не бывает промежуточного значения. В то время как между двумя любыми числами на числовой прямой всегда можно найти нечто промежуточное. Зато размеры можно хотя бы расположить по возрастающей или по убывающей. А вот пол человека так представить уже совсем не получится: есть только два значения, и «промежуточного» просто не бывает. Мы, конечно, можем обозначить женский пол единицей, а мужской — нулём (или двойкой), но никакой числовой информации эти обозначения нести не будут — их даже нельзя отсортировать. Есть ещё и другие специальные виды данных, например, углы, географические координаты, даты и т. п., но все они так или иначе могут быть представлены с помощью чисел. Таким образом, наиболее принципиальное различие между типами данных — это можно или нельзя их представить при помощи «обычных» чисел. Если нельзя, то такие данные принято называть *категориальными*. Статистические законы, а, значит, и статистические программы, работают

с такими данными, только если заранее указан их тип. Остальные типы данных в разных книгах называют по разному: числовые, счётные, порядковые или некатегориальные. Примем название «числовые» как самое простое.

3.2. Числовые векторы

Допустим, у нас есть данные о росте семи сотрудников небольшой компании. Вот так можно создать из этих данных простейший вектор:

```
> x <- c(174, 162, 188, 192, 165, 168, 172)
```

x — это имя объекта **R**, `<-` — функция присвоения, `c()` — функция создания вектора (от англ. «concatenate», собрать). Собственно, **R** и работает в основном с объектами и функциями. У объекта может быть своя структура:

```
> str(x)
 num [1:7] 174 162 188 192 165 168 172
```

то есть x — это числовой (num, numerical) вектор. В языках программирования бывают ещё скаляры, но в **R** скаляров нет. «Одиночные» объекты трактуются как векторы из одного элемента.

Вот так можно проверить, вектор ли перед нами:

```
> is.vector(x)
[1] TRUE
```

Вообще говоря, в **R** множество функций вида `is.что-то()` для подобной проверки, а ещё есть функции вида `as.что-то()`, которые будут использованы чуть далее по тексту. Называть объекты можно в принципе как угодно, но лучше придерживаться некоторых простых правил:

1) Использовать для названий только латинские буквы, цифры и точку (имена объектов не должны начинаться с точки или цифры);

2) Помнить, что **R** чувствителен к регистру, X и x — это разные имена;

3) Не давать объектам имена, уже занятые распространёнными функциями (типа `c()`), а также ключевыми словами (особенно `T`, `F`, `NA`, `NaN`, `Inf`).

Для создания «искусственных» векторов очень полезен оператор «`:`», а также функции `seq()` и `rep()`.

3.3. Факторы

Для обозначения категориальных данных в **R** есть несколько способов, разной степени «правильности». Во-первых, можно создать текстовый (character) вектор:

```
> sex <- c("male", "female", "male", "male",
+                          "female", "male", "male")
> is.character(sex)
[1] TRUE
> is.vector(sex)
[1] TRUE
> str(sex)
 chr [1:7] "male" "female" "male" "male" "female" "male" ...
```

Предположим, что sex — это описание пола сотрудников небольшой организации. Вот как **R** выводит содержимое этого вектора:

```
> sex
[1] "male" "female" "male" "male" "female" "male" "male"
```

Кстати, пора раскрыть загадку единицы в квадратных скобках — это просто номер элемента вектора. Вот как его можно использовать[1]:

```
> sex[1]
[1] "male"
```

«Умные», то есть объект-ориентированные команды **R** кое-что понимают про объект «sex», например, команда `table()`:

```
> table(sex)
sex
female   male
     2      5
```

А вот команда `plot()`, увы, не умеет ничего хорошего сделать с таким вектором. И это, в общем-то, правильно, потому что программа ничего не знает про свойства пола человека. В таких случаях пользователь сам должен проинформировать **R**, что его надо рассматривать как категориальный тип данных. Делается это так:

```
> sex.f <- factor(sex)
> sex.f
[1] male   female male   male   female male   male
Levels: female male
```

И теперь команда `plot()` уже понимает, что ей надо делать:

```
> plot(sex.f)
```

[1] Да-да, квадратные скобки — это тоже команда. Можно это проверить, набрав помощь `?"["`.

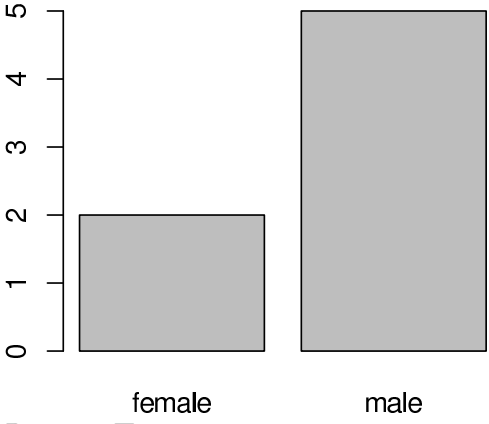

Рис. 3.1. Пример представления категориальных типов данных

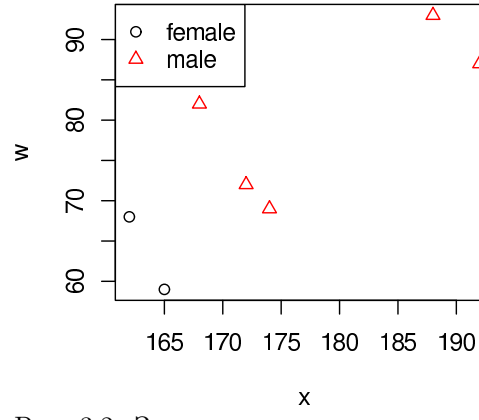

Рис. 3.2. Зависимость веса от роста с указанием пола.

Потому что перед нами специальный тип объекта, предназначенный для категориальных данных — фактор с двумя уровнями (levels):

```
> is.factor(sex.f)
[1] TRUE
> is.character(sex.f)
[1] FALSE
> str(sex.f)
 Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
```

Очень многие функции **R** (скажем, тот же самый `plot()`) предпочитают факторы текстовым векторам. При этом некоторые умеют конвертировать текстовые векторы в факторы, а некоторые — нет, поэтому надо быть внимательным.

Есть ещё несколько важных свойств факторов, которые надо знать заранее. Во-первых, подмножество фактора — это фактор с тем же количеством уровней, даже если их в подмножестве не осталось:

```
> sex.f[5:6]
[1] female male
Levels: female male
> sex.f[6:7]
[1] male male
Levels: female male
```

«Избавиться» от лишнего уровня можно, только применив специальный аргумент или выполнив преобразование данных «туда и обратно»:

```
> sex.f[6:7, drop=TRUE]
```

```
[1] male male
Levels: male
> factor(as.character(sex.f[6:7]))
[1] male male
Levels: male
```

Во-вторых, факторы в отличие от текстовых векторов можно легко преобразовать в числовые значения:

```
> as.numeric(sex.f)
[1] 2 1 2 2 1 2 2
```

Зачем это нужно, становится понятным, если рассмотреть вот такой пример: положим, кроме роста, у нас есть ещё и данные по весу сотрудников и мы хотим построить такой график, на котором были бы видны одновременно рост, вес и пол. Вот как это можно сделать:

```
> # Вектор веса
> w <- c(69, 68, 93, 87, 59, 82, 72)
> # Построение графика
> plot(x, w, pch=as.numeric(sex.f), col=as.numeric(sex.f))
> legend("topleft", pch=1:2, col=1:2, legend=levels(sex.f))
```

Тут, разумеется, нужно кое-что объяснить. `pch` и `col` — эти параметры предназначены для определения соответственно типа значков и их цвета на графике. Таким образом, в зависимости от того, какому полу принадлежит данная точка, она будет изображена кружком или треугольником и чёрным или красным цветом, соответственно. При условии, разумеется, что все три вектора соответствуют друг другу. Ещё надо отметить, что изображение пола при помощи значка и цвета избыточно, для «нормального» графика хватит и одного из этих способов.

В-третьих, факторы можно упорядочивать, превращая их некое подобие числовых данных. Введём четвёртую переменную: размер маек для тех же самых гипотетических восьмерых сотрудников:

```
> m <- c("L", "S", "XL", "XXL", "S", "M", "L")
> m.f <- factor(m)
> m.f
[1] L   S   XL  XXL S   M   L
Levels: L M S XL XXL
```

Как видно из примера, уровни расположены просто по алфавиту, а нам надо, чтобы "S"(small) шёл первым. Кроме того, надо как-то сообщить **R**, что перед нами не просто категориальные, а упорядочиваемые категориальные данные. Делается это так:

```
> m.o <- ordered(m.f, levels=c("S", "M", "L", "XL", "XXL"))
```

```
> m.o
[1] L   S   XL  XXL S   M   L
Levels: S < M < L < XL < XXL
```

Теперь **R** «знает», какой размер больше. Это может сыграть критическую роль, например, при вычислениях коэффициентов корреляции.

3.4. Пропущенные данные

В дополнение к векторам из чисел и текстовым векторам, **R** поддерживает ещё и логические вектора, а также специальные типы данных, которые бывают очень важны для статистических расчётов. Прежде всего это пропущенные или отсутствующие данные, которые обозначаются как `NA`. Такие данные очень часто возникают в реальных полевых и лабораторных исследованиях, опросах, тестированиях и т. д. При этом следует осознавать, что наличие пропущенных данных вовсе не означает, что данные в целом некачественны. С другой стороны, статистические программы должны как-то работать и с такими данными. Разберём следующие пример: предположим, что у нас имеется результат опроса тех же самых семи сотрудников. Их спрашивали: сколько в среднем часов они спят, при этом один из опрашиваемых отвечать отказался, другой ответил «не знаю», а третьего в момент опроса просто не было в офисе. Так возникли пропущенные данные:

```
> h <- c(8, 10, NA, NA, 8, NA, 8)
> h
[1]  8 10 NA NA  8 NA  8
```

Из примера видно, что NA надо вводить без кавычек, а **R** нимало не смущается, что среди цифр находится «вроде бы» текст. Отметим, что пропущенные данные очень часто столь же разнородны, как и в нашем примере. Однако кодируются они одинаково, и об этом не нужно забывать. Теперь о том, как надо работать с полученным вектором h. Если мы просто попробуем посчитать среднее значение (функция `mean()`), то получим:

```
> mean(h)
[1] NA
```

И это «идеологически правильно», поскольку функция может по-разному обрабатывать `NA`, и по умолчанию она просто сигнализирует о том, что с данными что-то не так. Чтобы высчитать среднее от «не пропущенной» части вектора, можно поступить одним из двух способов:

```
> mean(h, na.rm=TRUE)
[1] 8.5
> mean(na.omit(h))
```

```
[1] 8.5
```

Какой из способов лучше, зависит от ситуации. Часто возникает ещё одна проблема: как сделать подстановку пропущенных данных, скажем, заменить все NA на среднюю по выборке. Распространённое решение примерно следующее:

```
> h[is.na(h)] <- mean(h, na.rm=TRUE)
> h
[1]  8.0 10.0  8.5  8.5  8.0  8.5  8.0
```

В левой части первого выражения осуществляется индексирование, то есть выбор нужных значений h таких, которые являются пропущенными (is.na()). После того, как выражение выполнено, «старые» значения исчезают навсегда.

3.5. Матрицы

Матрицы — чрезвычайно распространённая форма представления данных, организованных в форме таблицы. Про матрицы в **R**, в общем, нужно знать две важные вещи: во-первых, что они могут быть разной размерности, и во-вторых, что матриц как таковых в **R**, по сути, нет.

Начнём с последнего. Матрица в **R** — это просто специальный тип вектора, обладающий некоторыми добавочными свойствами («атрибутами»), позволяющими интерпретировать его как совокупность строк и столбцов. Предположим, мы хотим создать простейшую матрицу 2×2. Для начала создадим её из числового вектора:

```
> m <- 1:4
> m
[1] 1 2 3 4
> ma <- matrix(m, ncol=2, byrow=TRUE)
> ma
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> str(ma)
 int [1:2, 1:2] 1 3 2 4
> str(m)
 int [1:4] 1 2 3 4
```

Из примера видно, что структура объектов m и ma не слишком различается. Различается, по сути, лишь их вывод на экран. Ещё очевиднее единство между векторами и матрицами прослеживается, если создать матрицу несколько иным способом:

```
> mb <- m
> mb
[1] 1 2 3 4
> attr(mb, "dim") <- c(2,2)
> mb
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Выглядит как некий фокус. Однако все просто: мы присваивает вектору mb атрибут dim (от слова dimensions, т. е. размерность) и устанавливаем значение этого атрибута в `c(2,2)`, то есть две строки и два столбца. Читателю предоставляется догадаться, почему матрица mb отличается от матрицы ma (ответ в конце статьи).

Мы указали лишь два способа создания матриц, а в действительности их гораздо больше. Очень популярно, например, «делать» матрицы из векторов-колонок или строк при помощи команд `cbind()` или `rbind()`. Если результат нужно «повернуть» на 90 градусов, используется команда `t()`.

Наиболее распространены матрицы, имеющие два измерения, однако никто не препятствует сделать многомерную матрицу:

```
> m3 <- 1:8
> dim(m3) <- c(2,2,2)
> m3
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

m3 — это трёхмерная матрица. Естественно, показать в виде таблицы её нельзя, поэтому **R** выводит её на экран в виде серии таблиц. Аналогично можно создать и четырёхмерную матрицу (как встроенные данные Titanic из предыдущей статьи). Многомерные матрицы в **R** принято называть arrays.

3.6. Списки

Списки — ещё один очень важный тип представления данных. Создавать их, особенно на первых порах, скорее всего, не придётся, но знать их особенности необходимо. Это нужно, прежде всего потому, что очень многие функции в **R** возвращают именно списки. В начале знакомства создадим список для тренировки:

```
> l <- list("R", 1:3, TRUE, NA, list("r", 4))
> l
[[1]]
[1] "R"

[[2]]
[1] 1 2 3

[[3]]
[1] TRUE

[[4]]
[1] NA

[[5]]
[[5]][[1]]
[1] "r"

[[5]][[2]]
[1] 4
```

Видно, что список — это своего рода ассорти. Вектор и, соответственно, матрица могут состоять из элементов только одного и того же типа, а вот список — из чего угодно. В том числе, как это видно из примера, и из других списков. Теперь поговорим про индексирование, или выборе элементов списка. Элементы вектора выбираются, при помощи функции-квадратной скобки:

```
> h[3]
[1] 8.5
```

Элементы матрицы выбираются так же, только используется несколько аргументов (для двумерных матриц это номер строки и номер столбца — именно в такой последовательности):

```
>  ma[2, 1]
[1] 3
```

А вот элементы списка выбираются тремя различными методами. Во первых можно использовать квадратные скобки:

```
> l[1]
[[1]]
[1] "R"
> str(l[1])
List of 1
 $ : chr "R"
```

Здесь очень важно, что полученный объект *тоже будет списком*. Во вторых можно использовать двойные квадратные скобки:

```
> l[[1]]
[1] "R"
> str(l[[1]])
 chr "R"
```

В этом случае полученный объект будет того типа, какого он был бы до объединения в список (поэтому первый объект будет текстовым вектором, а пятый — списком). В третьих можно использовать имена элементов списка. Но для этого сначала надо их присвоить:

```
> names(l) <- c("first", "second", "third",
+                                  "fourth", "fifth")
> l$first
[1] "R"
> str(l$first)
 chr "R"
```

Для выбора по имени используется знак доллара, а полученный объект будет таким же, как при использовании двойной квадратной скобки. На самом деле имена в **R** могут иметь и элементы вектора, и строки и стобцы матрицы:

```
> names(w) <- c("Коля", "Женя", "Петя", "Саша",
+                               "Катя", "Вася", "Жора")
> w
Коля Женя Петя Саша Катя Вася Жора
  69   68   93   87   59   82   72
> rownames(ma) <- c("a1","a2")
> colnames(ma) <- c("b1","b2")
> ma
   b1 b2
a1  1  2
a2  3  4
```

Единственное условие состоит в том, что все имена должны быть разными. Однако, знак доллара ($) можно использовать только со списками. Элементы вектора по имени можно отбирать так:

```
> w["Женя"]
Женя
  68
```

3.7. Таблицы данных

Наконец подошли к самому важному типу данных — к таблицам данных (data frames). Именно таблицы данных больше всего похожи на электронные таблицы Excel и аналогов, и поэтому с ними работают чаще всего. Особенно это касается начинающих пользователей **R**. Таблицы данных — это гибридный тип представления, *одномерный список из векторов одинаковой длины*. Таким образом, каждая таблица данных — это список колонок, причём внутри одной колонки все данные должны быть одного типа. Проиллюстрируем это на примере созданных в этой статье векторов:

```
> d <- data.frame(weight=w, height=x, size=m.o, sex=sex.f)
> d
     weight height size    sex
Коля     69    174    L   male
Женя     68    162    S female
Петя     93    188   XL   male
Саша     87    192  XXL   male
Катя     59    165    S female
Вася     82    168    M   male
Жора     72    172    L   male

> str(d)
'data.frame':   7 obs. of  4 variables:
 $ weight: num  69 68 93 87 59 82 72
 $ height: num  174 162 188 192 165 168 172
 $ size  : Ord.factor w/ 5 levels "S"<"M"<"L"<"XL"<"XXL":
3 1 4 5 1 2 3
 $ sex : Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
```

Поскольку таблица данных является списком, к ней применимы методы индексации списков. Кроме того, таблицы данных можно индексировать и как двумерные матрицы. Вот несколько примеров:

```
> d$weight
```

```
[1] 69 68 93 87 59 82 72
> d[[1]]
[1] 69 68 93 87 59 82 72
> d[,1]
[1] 69 68 93 87 59 82 72
> d[,"weight"]
[1] 69 68 93 87 59 82 72
```

Очень часто бывает нужно отобрать несколько конкретных колонок. Это можно сделать разными способами (исключаем столбец weight):

```
> d[,2:4]
     height size    sex
Коля    174    L   male
Женя    162    S female
Петя    188   XL   male
Саша    192  XXL   male
Катя    165    S female
Вася    168    M   male
Жора    172    L   male
> d[,-1]
     height size    sex
Коля    174    L   male
Женя    162    S female
Петя    188   XL   male
Саша    192  XXL   male
Катя    165    S female
Вася    168    M   male
Жора    172    L   male
```

Второй способ (отрицательная индексация) бывает в некоторых случаях незаменим. К индексации имеет прямое отношение ещё один тип данных в **R** — логические векторы. Как, например, отобрать из нашей таблицы только данные, относящиеся к женщинам? Вот один из способов:

```
> d[d$sex=="female",]
     weight height size    sex
Женя     68    162    S female
Катя     59    165    S female
```

Чтобы отобрать нужные строки, мы поместили перед запятой логическое выражение, `d$sex==female`. Его значением является логический вектор:

```
> d$sex=="female"
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

Таким образом, после того, как «отработала» селекция, в таблице данных остались только те строки, которые соответствуют "TRUE то есть строки 2 и 5.

Более сложным случаем селекции является сортировка таблиц данных. Для сортировки вектора достаточно применить команду `sort()`, а вот если нужно, скажем, отсортировать наши данные сначала по полу, а потом по росту, приходится применить операцию посложнее:

```
> d[order(d$sex, d$height), ]
     weight height size    sex
Женя     68    162    S female
Катя     59    165    S female
Вася     82    168    M   male
Жора     72    172    L   male
Коля     69    174    L   male
Петя     93    188   XL   male
Саша     87    192  XXL   male
```

Команда `order()` создаёт не логический, а числовой вектор, который соответствует будущему порядку расположения строк. Подумайте, как применить команду `order()` для того чтобы отсортировать колонки по алфавиту (ответ можно отыскать в конце статьи).

3.8. Векторизованные вычисления

Несмотря на то, что **R** похож на многие современные скриптовые языки программирования, например, такие, как Perl и Python, в нём есть много своеобразного. Одна из интересных и очень полезных особенностей **R** — это, так называемые, векторизованные вычисления. Использовать их очень просто. Допустим, мы хотим перевести вес из килограммов в граммы:

```
> w*1000
 Коля  Женя  Петя  Саша  Катя  Вася  Жора
69000 68000 93000 87000 59000 82000 72000
```

Для такой операции часто требуется использовать циклические конструкции (loops), а здесь всё получается в одно действие. Конечно, в R циклы тоже будут работать:

```
> for (i in seq_along(w)) {
+ w[i] <- w[i] * 1000
+ }
> w
 Коля  Женя  Петя  Саша  Катя  Вася  Жора
69000 68000 93000 87000 59000 82000 72000
```

Но это уж слишком громоздко. Векторизованы и операции между векторами и матрицами:

```
> ma + mb
   b1 b2
a1  2  5
a2  5  8

> 1:8 + 1:2
[1]  2  4  4  6  6  8  8 10
```

В последнем примере второй вектор гораздо короче первого, поэтому при вычислении результата он был несколько раз повторен. Так будет и в том случае, если длина меньшего вектора (матрицы) не кратна длине большего, но тогда **R** выдаст предупреждение.

Помимо простых арифметических операций, векторизованы и более сложные преобразования. Есть целое семейство функций, которые специализируются на векторизованных вычислениях: `apply()`, `by()`, `lapply()`, `sapply()`, `tapply()` и другие. Вот как работает, например, функция `apply()`:

```
> apply(trees, 2, mean)
   Girth   Height   Volume
13.24839 76.00000 30.17097
```

Двойка во втором аргументе означает, что среднее (`mean()`) вычисляется для каждой *колонки* данных. Для строк надо поставить единицу, но в данном случае это лишено смысла, потому что разные колонки относятся к измерениям разной природы. А вот так при помощи `sapply()` можно преобразовать наши данные в «кодированный», цифровой вид:

```
> sapply(d, as.numeric)
     weight height size sex
[1,]     69    174    3   2
[2,]     68    162    1   1
[3,]     93    188    4   2
[4,]     87    192    5   2
[5,]     59    165    1   1
[6,]     82    168    2   2
[7,]     72    172    3   2
```

`tapply()` и `by()` позволяют сделать ещё хитрее:

```
> by(d[,1:2], d$sex, mean)
d$sex: female
weight height
  63.5  163.5
```

```
------------------------------------------------------------
d$sex: male
weight height
  80.6  178.8
```

Мы вычислили средний рост и вес для мужчин и женщин за одно действие!

Наконец, `lapply()` позволяет применить некую команду к каждому элементу списка:

```
> lapply(d, is.factor)
$weight
[1] FALSE

$height
[1] FALSE

$size
[1] TRUE

$sex
[1] TRUE
```

Ответы на вопросы

Ответ на вопрос про матрицы. Когда мы создавали матрицу ma, мы использовали параметр `byrow=TRUE`. Значение его по умолчанию — `FALSE`, и если не задавать его так, как сделали мы, получится точно такая же матрица, как mb.

Ответ на вопрос про сортировку колонок.

```
> d[,order(colnames(d))]
```

В этом случае вместо `order()` можно было использовать и `sort()`.

Глава 4

Статистическая обработка данных

Данные собраны и упакованы, а теперь *анализируй*! Анализ данных — это то, ради чего данные и собираются.

4.1. Приёмы элементарного анализа данных

Начнём с самых элементарных приёмов анализа — вычисления общих характеристик выборки[1]. Можно сказать, что таких характеристик всего две: центр и разброс. В качестве центральной характеристики чаще всего используются среднее и медиана, а в качестве разброса — стандартное отклонение и квартили. Среднее отличается от медианы прежде всего тем, что оно хорошо работает в случае если распределение данных близко к нормальному. Медиана не так зависит от характеристики распределения, то есть, как говорят статистики, она более робастна или устойчива. Понять разницу легче всего на реальном примере. Возьмём опять наших гипотетических сотрудников. Вот их зарплаты (в тыс. руб.):

```
> salary <- c(21, 19, 27, 11, 102, 25, 21)
> names(salary) <- c("Коля", "Женя", "Петя", "Саша",
+                                  "Катя", "Вася", "Жора")
> salary
Коля Женя Петя Саша Катя Вася Жора
  21   19   27   11  102   25   21
```

Разница в зарплатах обусловлена, в частности, тем, что, скажем, Саша — экспедитор, а Катя — глава фирмы. Посмотрим чему равен центр:

```
> mean(salary); median(salary)
```

[1] Выборка — набор значений, полученных в результате ряда измерений.

```
[1] 32.28571
[1] 21
```

Получается, что из-за высокой Катиной зарплаты среднее гораздо хуже отражает «типичную», центральную зарплату, чем медиана.

Примечание: Мы не будем здесь объяснять, что такое среднее и как именно вычисляется медиана. Желающие это выяснить могут обратиться за формулами к любому учебнику по статистике.

Часто стоит задача посчитать среднее (или медиану) для целой таблицы данных. Есть несколько облегчающих жизнь приёмов, покажем их на примере встроенных данных trees:

```
> attach(trees)
> mean(Girth)
[1] 13.24839
> mean(Height)
[1] 76
> mean(Volume/Height)
[1] 0.3890012
> detach(trees)
```

Команда `attach` позволяет присоединить колонки таблицы данных к списку текущих переменных. После этого к переменным можно обращаться по именам, не упоминая имени таблицы. Важно не забыть сделать в конце `detach()`, потому что велика опасность запутаться в том, что Вы присоединили, а что — нет. Кроме того, если присоединённые переменные были как-то модифицированы, на самой таблице это не скажется. Это же можно сделать чуть по другому:

```
> with(trees, mean(Volume/Height))
[1] 0.3890012
```

Этот способ, в сущности, аналогичен первому, только присоединение происходит внутри круглых скобок. Можно так же воспользоваться тем фактом, что таблицы данных — это списки колонок:

```
> lapply(trees, mean)
$Girth
[1] 13.24839
$Height
[1] 76
$Volume
[1] 30.17097
```

Для строк такой прием не сработает, то есть надо будет запустить `apply()`. Не следует забывать, что циклические конструкции типа `for` в **R** использовать не рекомендуется.

Вот так определяются стандартное отклонение, дисперия (его квадрат) и так называемый межквартильный размах:

```
> sd(salary); var(salary); IQR(salary)
[1] 31.15934
[1] 970.9048
[1] 6
```

Опять-таки, `IQR` (межквартильный размах) лучше подходит для примера с зарплатой, чем стандартное отклонение, из-за высокой зарплаты у главы фирмы.

```
> attach(trees)
> mean(Height)
[1] 76
> median(Height)
[1] 76
> sd(Height)
[1] 6.371813
> IQR(Height)
[1] 8
> detach(trees)
```

А вот для деревьев эти характеристики куда ближе друг к другу. Разумно предположить, что распределение высоты деревьев близко к нормальному.

В наших данных по зарплате — всего 7 цифр. Их можно просмотреть глазами и всё понять. А как понять за разумный промежуток времени, есть ли какие-то «выдающиеся» цифры, типа Катиной директорской зарплаты, в данных тысячного размера? Для этого есть графические функции. Самая простая — это, так называемый, «ящик-с-усами», или боксплот. Для начала добавим к нашим данным ещё тысячу гипотетических работников с зарплатой, случайно взятой из межквартильного размаха исходных данных:

```
> new.1000 <- sample((median(salary)-IQR(salary)):
+  (median(salary)+IQR(salary)), 1000, replace=TRUE)
> salary2 <- c(salary, new.1000)
> boxplot(salary2)
```

Это пример интересен ещё и потому, что в нём впервые представлена техника получения случайных значений. Функция `sample()` способна выбирать случайным образом данные из выборки. В данном случае мы использовали `replace=TRUE`, поскольку нам нужно было выбрать много чисел из гораздо меньшей выборки. Если писать на **R** имитацию карточных игр (а такие программы *уже* написаны!), то надо использовать `replace=FALSE`, потому что из колоды нельзя достать опять ту же самую карту. Кстати говоря, из того что значения случайные, следует, что результаты последующих вычислений могут отличаться, если их воспроизвести ещё раз.

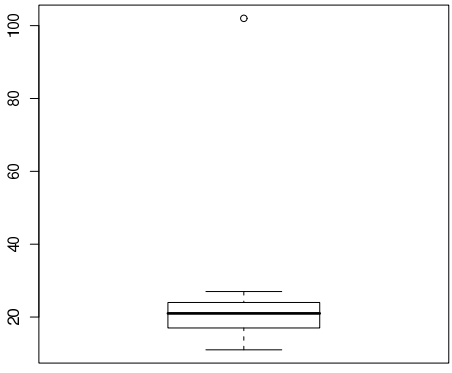

Рис. 4.1. boxplot или «ящик с усами»

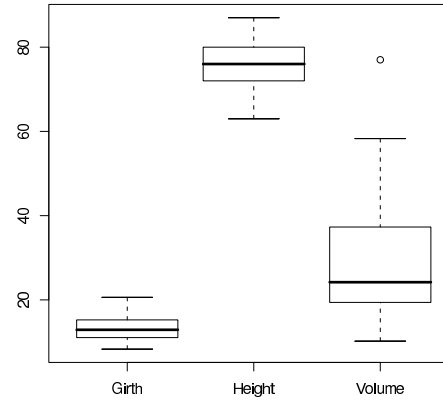

Рис. 4.2. Действие команды `boxplot` на встроенный объект trees

Но вернемся к боксплоту. Как видно из рис. 4.1, Катина зарплата представлена высоко расположенной точкой. Сам бокс, то есть главный прямоугольник, ограничен сверху и снизу квартилями, так что высота прямоугольника — это IQR. Так называемые «усы» по умолчанию обозначают точки, удалённые на полтора IQR. Линия посередине прямоугольника — это, как легко догадаться, медиана. Точки лежащие вне усов, рассматриваются как выбросы, и поэтому рисуются отдельно. Боксплоты были специально придуманы известным статистиком Дж. Тьюки (John W. Tukey[2]) для того, чтобы быстро, эффективно и устойчиво отражать основные характеристики выборки. **R** использует оригинальные боксплоты Тьюки, а кроме того, может рисовать несколько боксплотов сразу, то есть эта команда векторизована:

```
> boxplot(trees)
```

Есть ещё две функции, которые связаны с боксплотами: функция `quantile()` по умолчанию выдает все пять квартилей, а функция `fivenum()` предоставляет все основные характеристики распределения по Тьюки.

Другой способ графического изображения выборки тоже очень популярен (см. рис. 4.3). Это гистограммы, то есть столбики, высота которых соответствует встречаемости данных, попавших в определенный диапазон:

```
> hist(salary2, breaks=20)
```

По умолчанию команда `hist` разбивает переменную на 10 интервалов, но их количество можно указать и вручную, как в предложенном примере. Числен-

[2]Интересно, что именно Джон Тьюки первый в 1958 г. применил слово software по отношению к программному обеспечению.

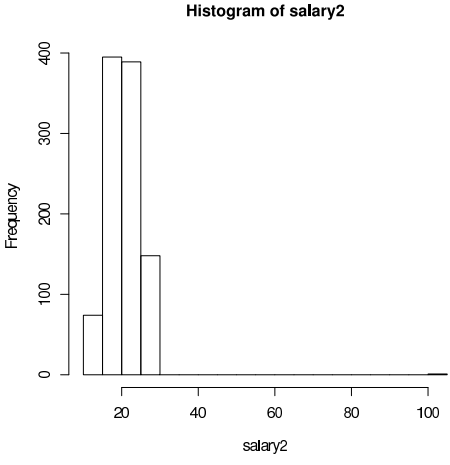

Рис. 4.3. Гистограмма (команда `hist`)

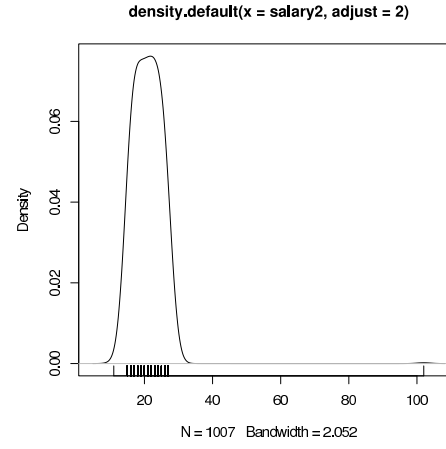

Рис. 4.4. Сглаженная гистограмма (результат действий команд `density` и `rug`)

ным аналогом гистограммы является функция `cut()`. При помощи этой функции можно выяснить, сколько данных какого типа у нас имеется:

```
> table(cut(salary2, 20))

(10.9,15.5]   (15.5,20]   (20,24.6] (24.6,29.1] (29.1,33.7]
         74         395         318         219           0
(33.7,38.3] (38.3,42.8] (42.8,47.4] (47.4,51.9] (51.9,56.5]
          0           0           0           0           0
(56.5,61.1] (61.1,65.6] (65.6,70.2] (70.2,74.7] (74.7,79.3]
          0           0           0           0           0
(79.3,83.9] (83.9,88.4]   (88.4,93]   (93,97.5]  (97.5,102]
          0           0           0           0           1
```

Есть ещё две графические функции, близкие по своей идеологии к гистограмме. Во-первых, это `stem()` — псевдографическая (текстовая) гистограмма:

```
> stem(salary, scale=2)
  The decimal point is 1 digit(s) to the right of the 1  19
   2  11573
   4 5
   6 7
   8 9
  10  2
```

Логика отображения не сложна: значения данных изображаются не точками, а цифрами, соответствующими самим этим значениям. Таким образом, видно, что в интервале от 10 до 20 есть две зарплаты (11 и 19), в интервале от 20 до 30 — четыре (21, 21, 25, 27) и т. д.

Другая функция тоже близка к гистограмме (см. рис. 4.4), но требует гораздо более изощрённых вычислений. Это график плотности распределения:

```
> plot(density(salary2, adjust=2))
> rug(salary2)
```

В этом примере была использована «добавляющая» графическая функция `rug()`, чтобы акцентировать места с наиболее высокой плотностью. По сути, перед нами сглаживание гистограммы, иными словами, попытка превратить её в непрерывную гладкую функцию. Насколько гладкой она будет, зависит от параметра `adjust` (по умолчанию он равен единице).

Ну и, наконец, самая главная функция для описания базовой статистики:

```
> summary(trees)
     Girth           Height       Volume
 Min.   : 8.30   Min.   :63   Min.   :10.20
 1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
 Median :12.90   Median :76   Median :24.20
 Mean   :13.25   Mean   :76   Mean   :30.17
 3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
 Max.   :20.60   Max.   :87   Max.   :77.00

> lapply(list(salary, salary2), summary)
[[1]]
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  11.00   20.00   21.00   32.29   26.00  102.00

[[2]]
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  11.00   17.00   21.00   20.97   24.00  102.00
```

Заметьте, что у обоих «зарплат» медианы одинаковы, тогда как средние существенно отличаются. Это ещё один пример неустойчивости средних значений, ведь с добавлением случайно взятых «зарплат» вид распределения не должен был существенно поменяться.

Фактически команда `summary()`, возвращает те же самые данные, что и `fivenum()` с добавлением среднего значения (Mean). Однако, устроена эта функция более «хитро». Во-первых, она общая, и по законам объект-ориентированного подхода возвращает разные значения для объектов разного типа. В последнем примере видно как она работает с числовыми векторами, матрицами и таблицами данных.

Для списков она работает по-другому. Вывод может быть таким (на примере встроенных данных о 23 землетрясениях в Калифорнии):

```
> summary(attenu)
    event           mag          station        dist
Min.   : 1.00  Min.   :5.000  117    :  5  Min.   :  0.50
1st Qu.: 9.00  1st Qu.:5.300  1028   :  4  1st Qu.: 11.32
Median :18.00  Median :6.100  113    :  4  Median : 23.40
Mean   :14.74  Mean   :6.084  112    :  3  Mean   : 45.60
3rd Qu.:20.00  3rd Qu.:6.600  135    :  3  3rd Qu.: 47.55
Max.   :23.00  Max.   :7.700  (Other):147  Max.   :370.00
                              NA     : 16
```

Переменная station (номер станции наблюдений, третья колонка) — фактор, и к тому же с пропущенными данными. Чтобы не ошибиться, полезно узнать, какие методы у общей функции существуют:

```
> methods(summary)
 [1] summary.Date            summary.POSIXct
 [3] summary.POSIXlt         summary.aov
 [5] summary.aovlist         summary.connection
 [7] summary.data.frame      summary.default
 [9] summary.ecdf*           summary.factor
[11] summary.glm             summary.infl
[13] summary.lm              summary.loess*
[15] summary.manova          summary.matrix
[17] summary.mlm             summary.nls*
[19] summary.packageStatus*  summary.ppr*
[21] summary.prcomp*         summary.princomp*
[23] summary.stepfun         summary.stl*
[25] summary.table           summary.tukeysmooth*
   Non-visible functions are asterisked
```

А когда Вам нужна помощь, надо указать, какая конкретно версия `summary()` имеется в виду:

```
> ?summary.data.frame
```

4.2. Одномерные статистические тесты

Закончив разбираться с описательными статистиками, перейдём к простейшим статистическим тестам. Начнём «одномерных» тестов, которые позволяют проверить утверждения относительно того, как распределены исходные данные.

Предположим, мы знаем, что средняя зарплата в нашем первом примере этой главы около 32 тыс. руб. Проверим теперь, насколько эта наша информация достоверна:

```
> t.test(salary, mu=32)

        One Sample t-test

data:  salary
t = 0.0243, df = 6, p-value = 0.9814
alternative hypothesis: true mean is not equal to 32
95 percent confidence interval:
  3.468127 61.103302
sample estimates:
mean of x
 32.28571
```

Это был вариант теста Стьюдента[3] для одномерных данных.

Статистические тесты пытаются высчитать, так называемую, тестовую статистику. Затем на основании этой статистики рассчитывается p-величина или p-value, отражающая вероятность ошибки первого рода. Ошибкой первого рода (её ещё называют «ложной тревогой»), в свою очередь, называется ситуация, когда мы принимаем альтернативную гипотезу, в то время как на самом деле верна нулевая. Принято считать, что нулевой гипотезе соответствует ситуация «по умолчанию». Наконец, вычисленная p-величина используется для сравнения с заранее заданным порогом (уровнем) значимости. Если p-величина ниже порога, то нулевая гипотеза отвергается, а если выше, то принимается.

Перейдём к анализу вывода функции. Вычисляемая статистика в нашем случае t (критерий Стьюдента). При шести степенях свободы (df=6, поскольку у нас всего 7 значений) это даёт p-значение очень близкое к единице ($0.9814 \simeq 1$). Какой бы распространённый порог мы не приняли (0.1%, 1% или 5%), это значение всё равно больше. Следовательно, мы принимаем нулевую гипотезу (наша информация о средней зарплате скорее верна чем нет). Поскольку альтернативная гипотеза в нашем случае — это то, что предполагаемое среднее не равно вычисленному среднему, то получается, что «на самом деле» эти цифры статистически не отличаются. Кроме всего этого, функция выдаёт ещё и доверительный интервал (confidence interval), в котором, по её «мнению», может находиться «истинное» среднее. В данном случае он очень широк — от трёх с половиной тысяч до 61 тысячи рублей (это всё из-за высокой Катиной зарплаты).

[3]Данный критерий был разработан Уильямом Госсеттом (William Sealy Gosset) для оценки качества пива в компании Гиннесс. Статья Госсетта вышла в журнале «Биометрика» под псевдонимом «Student» (Студент).

Существует так же непараметрический аналог этого теста, то есть теста не связанного предположениями о распределении. Это ранговый тест Уилкоксона (Wilcoxon signed-rank test):

```
> wilcox.test(salary2, mu=median(salary2), conf.int=TRUE)

     Wilcoxon signed rank test with continuity correction

data:  salary2
V = 206882.5, p-value = 0.3704
alternative hypothesis: true location is not equal to 21
95 percent confidence interval:
 20.50008 21.00003
sample estimates:
(pseudo)median
      20.99995
```

Эта функция и выводит практически то же самое. Обратите внимание, что тест связан не со средним, а с медианой. Соответственно, вычисляется (если задать `conf.int=TRUE`) доверительный интервал. Здесь он значительно у́же.

Некоторые статистические методы (например, ANOVA или дисперсионный анализ) основаны на том, что данные имеют нормальное распределение. Поэтому вопрос соответствует ли распределение данных нормальному или хотя бы напоминает оно нормальное хоть как-то является очень и очень важным. В **R** реализовано несколько разных техник, отвечающих на вопрос о нормальности. Во-первых, это статистические тесты. Самый простой из них — тест Шапиро-Уилкса (Shapiro-Wilk test):

```
> shapiro.test(salary)
        Shapiro-Wilk normality test

data:  salary
W = 0.6116, p-value = 0.0003726

> shapiro.test(salary2)
        Shapiro-Wilk normality test

data:  salary2
W = 0.7407, p-value < 2.2e-16
```

Но что же он показывает? Здесь функция выводит гораздо меньше, чем в предыдущих случаях. Более того, даже встроенная справка не содержит объяснений того, какая здесь, например, альтернативная гипотеза. Что, собственно, показывает p-значение? Разумеется, можно обратиться к литературе, благо

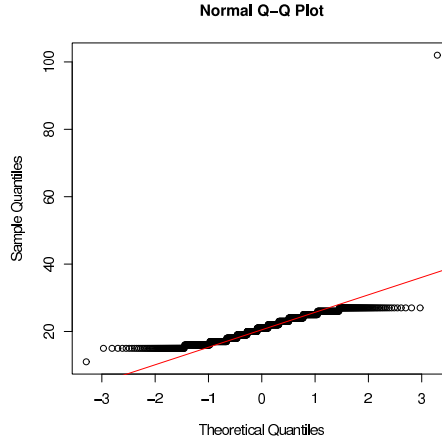

Рис. 4.5. Графическая проверка выборки на нормальность

справка даёт ссылки на статьи. А можно просто поставить модельный эксперимент:

```
> set.seed(1638)
> shapiro.test(rnorm(100))

        Shapiro-Wilk normality test

data:  rnorm(100)
W = 0.9934, p-value = 0.9094
```

`rnorm()` генерирует столько случайных чисел, распределённых по нормальному закону, сколько указано в его аргументе. Это аналог функции `sample()`. Раз мы получили высокое p-значение, то это свидетельствует о том, что альтернативная гипотеза в данном случае: «распределение не соответствует нормальному». Обратите внимание, что для того чтобы результаты при вторичном воспроизведении были теми же, использована функция `set.seed()`. Эта функция подстраивает встроенный в **R** генератор случайных чисел так, чтобы числа в следующей команде были сгенерированы по одному и тому же «закону». Кстати говоря, генераторов случайных чисел в **R** целых шесть (см. `help(set.seed)`)!

Таким образом на основании теста Шапиро-Уилкса можно заключить, что распределение данных в salary и salary2 существенно отличается от нормального.

Другой популярный способ проверить, насколько распределение похожее на нормальное — графический (см. рис. 4.5). Это делается примерно так:

```
> qqnorm(salary2); qqline(salary2, col=2)
```

Для каждого элемента вычисляется, какое место он должен занять в сортированных данных (выборочный квантиль), и какое место он должен был бы занять, если распределение нормальное (теоретический квантиль). Прямая проводится через квартили. Если точки лежат на прямой, то распределение нормальное. В нашем случае точки лежат достаточно далеко от красной прямой, а значит, не соответствуют «нормальным».

4.3. Как создавать свои функции

Тест Шапиро-Уилкса всем хорош, но он не векторизован, как и многие другие тесты в **R**. Поэтому применить его сразу к нескольким колонкам таблицы данных не получится. Можно, конечно, аккуратно повторить его для каждой колонки, но более глобальный подход — это создать пользовательскую функцию. Вот пример такой функции:

```
> normality <- function(data.f)
+ {
+ result <- data.frame(var=names(data.f),
+           p.value=rep(0, ncol(data.f)),
+           normality=is.numeric(names(data.f)))
+ for (i in 1:ncol(data.f))
+   {
+   data.sh <- shapiro.test(data.f[, i])$p.value
+   result[i, 2] <- round(data.sh, 5)
+   result[i, 3] <- (data.sh > .05)
+   }
+ return(result)
+ }
```

Чтобы функция заработала, надо скопировать эти строчки в окно консоли, или записать их в отдельный файл (желательно с расширением `.r`), а потом загрузить командой `source()`. После этого её можно вызвать:

```
> normality(trees)
     var p.value normality
1  Girth 0.08893      TRUE
2 Height 0.40342      TRUE
3 Volume 0.00358     FALSE
```

Функция не только запускает тест Шапиро-Уилкса несколько раз, но ещё и разборчиво оформляет результат выполнения. Разберём функцию чуть подробнее. В первой строчке указан ее аргумент «data.f». Дальше, в окружении фигурных скобок, находится само тело функции. На третьей строчке формируется пустая таблица данных такой размерности, какая потребуется нам в конце. После этого

начинается цикл: для каждой колонки выполняется тест, а потом (*это важно!*) из теста извлекается p-значение. Эта процедура основана на знании структуры вывода теста — это список, где элемент «p-value» содержит p-значение. Проверить это можно, заглянув в справку, а можно и экспериментально (как? см. ответ в конце главы). Все p-значения извлекаются, округляются, сравниваются с пороговым уровнем значимости (в данном случае 0.05), и записываются в таблицу. Затем таблица выдаётся «наружу». Предложенная функция совершенно не оптимизирована. Её легко можно сделать чуть короче, и к тому же несколько «смышлёнее», скажем, так:

```
> normality2 <- function(data.f, p=.05)
+ {
+ nn <- ncol(data.f)
+ result<-data.frame(var=names(data.f),p.value=numeric(nn),
+ normality=logical(nn))
+ for (i in 1:nn)
+   {
+   data.sh <- shapiro.test(data.f[, i])$p.value
+   result[i, 2:3] <- list(round(data.sh, 5), data.sh > p)
+   }
+ return(result)
+ }

> normality2(trees)
     var p.value normality
1  Girth 0.08893      TRUE
2 Height 0.40342      TRUE
3 Volume 0.00358     FALSE
```

Результаты, разумеется, не отличаются. Зато теперь видно, как можно добавить аргумент, причём сразу со значением по умолчанию. Теперь можно вызвать функцию и так:

```
> normality2(trees, 0.1)
     var p.value normality
1  Girth 0.08893     FALSE
2 Height 0.40341      TRUE
3 Volume 0.00358     FALSE
```

То есть если вместо 5% взять порог в 10%, то уже и для первой колонки можно отвергнуть нормальное распределение.

Не раз говорилось, что циклов в **R** следует избегать. Можно ли сделать это в нашем случае? Оказывается, да:

```
> lapply(trees, shapiro.test)
```

```
$Girth

        Shapiro-Wilk normality test

data:  X[[1]]
W = 0.9412, p-value = 0.08893


$Height

        Shapiro-Wilk normality test

data:  X[[2]]
W = 0.9655, p-value = 0.4034


$Volume

        Shapiro-Wilk normality test

data:  X[[3]]
W = 0.8876, p-value = 0.003579
```

Как видите, всё ещё проще! Если мы хотим улучшить зрительный эффект для вывода, то можно сделать так:

```
> lapply(trees, function(.x)
+   ifelse(shapiro.test(.x)$p.value > .05,
+                     "NORMAL", "NOT␣NORMAL"))
$Girth
[1] "NORMAL"

$Height
[1] "NORMAL"

$Volume
[1] "NOT␣NORMAL"
```

Здесь применена так называемая анонимная функция или функция без названия, обычно употребляемая в качестве последнего аргумента команд типа `apply()`. Кроме того, используется логическая конструкция `ifelse()`. И, наконец, на этой основе можно сделать третью пользовательскую функцию:

```
> normality3 <- function(df, p=.05)
+ {
```

```
+ lapply(df, function(.x)
+       ifelse(shapiro.test(.x)$p.value > p,
+                       "NORMAL","NOT␣NORMAL"))
+ }
> normality3(list(salary, salary2))
[[1]]
[1] "NOT␣NORMAL"

[[2]]
[1] "NOT␣NORMAL"

> normality3(log(trees))
$Girth
[1] "NORMAL"

$Height
[1] "NORMAL"

$Volume
[1] "NORMAL"
```

Эти примеры тоже интересны. Во-первых, нашу третью функцию можно применять не только к таблицам данных, но и к «настоящим» спискам, с неравной длиной элементов. Во-вторых, простейшее логарифмическое преобразование сразу же изменило «нормальность» колонок. Это следует запомнить, как и то, насколько просто такие преобразования делаются в **R**.

Ответ на вопрос

```
> str(shapiro.test(rnorm(100)))
```