Учебно-методическое пособие

Зарядов Иван Сергеевич

Введение в статистический пакет **R**: типы переменных, структуры данных, чтение и запись информации, графика.

Москва

Издательство Российского университета дружбы народов

2010

Оглавление

Глава 1

Введение

1.1 Цель этой книги

Данное пособие предназначено для студентов 3–4 курсов специальности «Прикладная математика и информатика», изучающих курс «Практикум по статистике», а также для всех желающих разобраться в прикладных аспектах математической статистики на примере работы в пакете **R**.

1.2 Что из себя представляет пакет R

R — это одновременно и свободно распространяемая программная среда с открытым кодом, развиваемая в рамках проекта **GNU**, и язык программирования для статистической обработки данных и работы с графикой.

R можно применять везде, где нужна работа с данными. Это и сама математическая статистика во всех её приложениях, и первичный анализ данных, и математическое моделирование. Основная мощь **R** лучше всего проявляется именно при статистическом анализе данных: от вычисления средних величин до серьёзных операций с временными рядами. С помощью **R** можно подготовить данные для исследования, которое может быть осуществлено с помощью реализованных в различных функциях статистических методов, а затем вывести полученные результаты для дальнейшего анализа.

Сейчас практически во всех западноевропейских и американских университетах изучают и используют **R**, ежегодно издаются многостраничные учебники и монографии относительно как работы с самим пакетом **R**, так и его применения при исследовании и обработке данных в статистике, медицине, экологии, финансовом анализе, актуарной математике и пр. Многие компании также применяют **R**, например Boeing.

1.3 История создания

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

Язык **S** был разработан в 1976 году в компании AT&T Labs. Первая реализация **S** была написана на FORTRAN и работала под управлением операционной системы GCOS. В 1980 году реализация была переписана под UNIX. Именно тогда в научной среде и стал распространяться **S**. В 1988 году вышла третья версия, коммерческая реализация которой стала называться **S-PLUS**. Довольно высокая стоимость предлагаемого коммерческого статистического пакета и привела к возникновению **R**.

В августе 1993 двое новозеландских учёных (Robert Gentleman и Ross Ihaka, Statistics Department, Auckland University) анонсировали свою разработку под названием **R**. Это была новая реализация языка **S**, отличающаяся от **S-PLUS** рядом деталей, например, работой с памятью, обращением с глобальными и локальными переменными.

Поначалу проект развивался довольно медленно, но с появлением возможности довольно лёгкого написания дополнений (пакетов) всё большее количество людей стало переходить с **S-PLUS** на **R**. После устранения проблем, связанных с памятью, среди пользователей **R** стали появляться и «поклонники» других пакетов (**SAS**, **Stata**, **SYSTAT**, **SPSS**). Количество книг по **R** за последние годы значительно выросло, а число дополнительных к базовой версии пакетов стало больше двух тысяч (на начало января 2010 года число пакетов составило 2142).

Дополнительную популярность **R** принесло создание центральной системы хранения и распространения пакетов — **CRAN** (Comprehensive R Archive Network — http://cran.r-project.org).

1.4 «Плюсы» и «минусы» R

Перечислим достоинства и недостатки пакета **R**.
«Плюсы»:

- **R** является свободно распространяемым программным обеспечением (ПО), каждый может его бесплатно скачать с сайта http://www.r-project.org;
- достаточно просто устанавливается под Windows, MacOS X, Linux;
- базовая комплектация **R** занимает немного места на жёстком диске и включает в себя все функции, необходимые для статистического анализа;

– для более серьёзной работы всегда можно дополнительно установить вспомогательные пакеты с необходимыми функциями;

– на данный момент разработаны пакеты, применимые практически во в всех областях знания, где используется статистика;

– можно работать с большими массивами данных (несколько сотен тысяч наблюдений);

– встроенная система помощи и подсказок;

– хорошие графические возможности представления результатов исследований;

– возможность самостоятельного написания необходимых функций;

– много свободной литературы по **R**.

Недостатки:

– в отличие от большинства коммерческих программ **R** имеет не графический интерфейс, а интерфейс командной строки, таким образом нужно знать необходимые для работы функции и синтаксис языка программирования;

– нет коммерческой поддержки (но есть международная система рассылки сообщений об обновлениях);

– довольно мало литературы по **R** на русском языке (в основном литература на английском), но при желании можно найти в Интернете.

1.5 Полезные ссылки

Приведём ряд полезных ссылок:

– http://www.r-project.org/ — сайт проекта **R**;

– http://cran.gis-lab.info/ и http://www.cran.r-project.org/ — **CRAN**;

– http://journal.r-project.org/ — сайт журнала по **R**;

– https://stat.ethz.ch/mailman/r-help/ — список рассылки **R**-help;

– http://wiki.linuxformat.ru/index.php/LXF100-101:R — небольшая книга по **R** на русском языке;

– http://herba.msu.ru/shipunov/software/r/rplus1.htm — небольшая методичка на русском языке;

– http://www.sciviews.org/Tinn-R/ — графический интерфейс для работы с **R**;

– http://ru.wikipedia.org/wiki/R_Commande — графический интерфейс для работы с **R**;

– http://cran.r-project.org/doc/contrib — учебники по **R** на английском и других языках;

– http://finzi.psych.upenn.edu/nmz.html — поиск в материалах по **R**;

– http://www.statmethods.net/index.html — справочный ресурс;

– http://pj.freefaculty.org/R/Rtips.html — советы по использованию **R**;

– http://www.uic.unn.ru/ zny/ml//Labs/ — ещё один русскоязычный сайт по **R**.

1.6 Благодарности

Автор благодарен студентам 3, 4 и 5 курсов специальности «Прикладная математика и информатика» факультета физико-математических и естественных наук РУДН за помощь в разработке данного пособия. Особая благодарность Дубининой Ю. и Стальченко И.

Глава 2

Начальные этапы работы в R

2.1 Установка и запуск статистического пакета

2.1.1 Установка пакета

Скачать и установить **R** довольно просто. Надо зайти на сайт http://cran.gis-lab.info/ и выбрать ту версию пакета, которая подходит под Вашу операционную систему. Последняя (на момент создания данного пособия) версия **R** — R-2.12.1 от 16 декабря 2010 года[1]. На примере вышеуказанной версии и рассмотрим установку. Этапы установки:

- Выбираем нужную версию пакета (в нашем случае R-2.12.1), лучше всего непосредственно exe-файл (exe-файл в базовой комплектации **base** займёт 37 мегабайт).
- Установка **R** под Windows[2] довольно типична. Щёлкаем по скачанному exe-файлу. По ходу установки задаётся ряд вопросов.
  - Первый вопрос — язык во время установки. Выбираем русский (в принципе, он и предлагается).
  - Далее предлагается закрыть все рабочие приложения перед началом установки. Можно так и сделать (а можно и не делать). Нажимаем кнопку «Далее».

[1]Как правило, в новой версии устраняются недостатки предыдущих. Но, желательно установить несколько версий (текущую и предыдущую), чтобы проверить, какая из них по функциональности больше подходит. К примеру, в версии R-2.9.2 по сравнению с R-2.7.2 есть недостаток — для полноценной работы в подокнах необходимо их разворачивать на весь экран

[2]В дальнейшем речь пойдёт о работе именно в этой операционной системе

– Затем выводится лицензионное соглашение, если с ним согласны, то снова жмём на кнопку «Далее».

– Выбираем папку, где будет установлен пакет. (По умолчанию создаётся директория «R» с поддиректорией «R-2.10.1» в «Program Files»)

– Далее предлагаются варианты установки. Выбираем все pdf-файлы с руководством пользователя (Manual). Полный вариант установки займёт примерно 65 мегабайт на жёстком диске.

– Создаём ярлыки в предлагаемой папке (или выбираем сами) меню «Пуск».

– На следующем шаге выбираем предлагаемые варианты (создание ярлыка на «Рабочем столе» и привязывание к **R** файлов с расширением .Rdata).

- Сама установка займёт буквально пару минут. По её окончанию на рабочем столе появится ярлык с указанием версии пакета.

2.1.2 Запуск программы и начало работы

После установки на рабочем столе имеется ярлык **R**, щёлкая по которому, запускаем программу. Появится основное окно программы, содержащее подокно — R Console. Каждая сессия в **R** начинается со следующих строк

```
R version 2.12.1 (2010-12-16)
Copyright (C) 2010 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.
```

```
[Загружено ранее сохраненное рабочее пространство]

>
```

Это командное окно (консоль), в котором пользователь вводит команды, а программа печатает результаты. В ходе работы в основном графическом окне могут появиться и другие окна — редактор скриптов, окна с графическим результатом выполнения команд (графики).

Команды вводятся пользователем в консоли (командном окне) после приглашения, имеющего вид

```
>
```

После нажатия кнопки Enter, введённая команда поступает на обработку. В одной строке можно ввести несколько команд, разделяя их **;**. Одну команду можно расположить и на двух (и более) строках. Для этого достаточно нажать Enter, тогда на новой строке вместо

```
>
```

появится приглашение

```
+
```

Символ # означает начало комментария. Всё, что находится после этого знака в рамках одной строки, игнорируется программой.

Кнопки ↑ и ↓ на клавиатуре позволяют осуществлять навигацию среди ранее введённых команд (можно выбрать одну из предыдущих команд).

С помощью кнопок → и ← можно перемещаться в уже введённой команде, в частности, редактируя её.

В **R** можно создавать имена для различных объектов (переменных) как на латинице, так и на кириллице[3], но следует учесть, что **а** (кириллица) и **a** (латиница) — это два разных объекта. Кроме того, **R** чувствителен и к регистру, т.е. строчные и заглавные буквы в нём различаются. Имена переменных (идентификаторов) в **R** состоят из букв, цифр и знаков точки (.) и подчёркивания (_). Имя объекта не может начинаться с цифры, и если первый символ — это точка, то цифра не может быть вторым символом[4]. При помощи **?имя** можно проверить, есть ли такая переменная или функция с тем же именем.

Операторами присваивания в **R** выступают либо =, либо <- (состоящий из двух символов: < и -) — присваивание справа, либо -> — присваивание слева, как поодиночке, так и в последовательности.

[3]Предпочтительнее в латинице

[4]Возможны имена переменных, состоящие только из точек, причём любого их количества, кроме трёх

Пример 1. *Рассмотрим несколько вариантов.*

```
> aa<-5;aa
[1] 5
> 6->bb;bb
[1] 6
> c=7;c
[1] 7
> cc<-dd<-8;cc
[1] 8
> dd
[1] 8
>
```

Сначала переменной **aa** справа присваивается значение 5, повторное обращение к этой переменной вызывает вывод на экран результата операции. Затем переменной **bb** слева присваивается значение 6 и результат выводится на экран. Переменной **c** при помощи знака «равно» присваивается значение 7. И, наконец, переменной **cc** присваивается значение переменной **dd**, которой в свою очередь было присвоено значение 8.

Заметим, что выводимые результаты начинаются с [1]. Это объясняется тем, что **R** рассматривает любые вводимые данные (если не указано иное) как массив. Выводимое число — это вектор длины 1, первый и единственный элемент которого и обозначается [1].

```
> bb->cc<-7
```

В данном случае получим сообщение об ошибке, так как переменной **cc** слева присваивается значение переменной **bb**, а справа — 7.

```
> cc=7
> cc<-5;cc
[1] 5
```

Присваивание нового значения переменной уничтожает присвоенное ранее.

Чтобы для вывода результатов повторно не обращаться к переменной, достаточно взять операцию в круглые скобки.

```
> (z=6)
[1] 6
```

Если нужно, чтобы выводились результаты только последнего выражения, нужно весь блок команд взять в фигурные скобки.

```
> {z=5;
+ c=4;z
+ c}
[1] 4
```

Выражения в {} воспринимаются как единое целое.

В **R** много функций, чьё имя состоит только из одной буквы[5] (например, **c()** или **t()**), поэтому создание переменной со схожим названием может привести к ряду проблем. Чтобы проверить, есть ли в **R** функция, примеру, **t()**, нужно сделать следующее:

```
> t
function (x)
UseMethod("t")
<environment: namespace:base>
```

В результате выведено сообщение о том. что в пакете **base** есть функция с таким именем.

Однако, работать в режиме командной строки не очень удобно, особенно если пишется большая программа. *После того, как нажата кнопка Enter, исправить набранную команду уже невозможно.* И, если была ошибка, то придётся начать заново с того места в вводимой программе, где и была сделана ошибка. Поэтому, перейдём к следующему разделу.

2.1.3 Работа со скриптами

Скрипт — это самостоятельно написанная программа с использованием всех возможностей **R**. В командном меню выбираем **Файл**, а затем нажимаем на **Новый скрипт**. Откроется новое окно, представляющее из себя редактор, в котором можно писать и править текст создаваемой программы.

Стоит отметить, что для полноценной работы в этом редакторе не обязательно его раскрывать на весь экран монитора. При переходе от редактора к консоли и наоборот будет меняться командное меню.

Разберём командное меню редактора.

- **Файл**:
  - **Новый скрипт** — создаётся новое «безымянное» окно текстового редактора, открытые ранее окна редактора не закрываются (можно также воспользоваться сочетанием клавиш **Ctrl**+**N**).
  - **Открыть скрипт** — открываем ранее созданный скрипт (**Ctrl**+**O**)

[5] всё следующее верно и для функций, имя которых состоит более чем из одной буквы

– **Сохранить** — созданный скрипт сохраняется в рабочей директории (**Ctrl**+**S**). Если скрипт сохраняется в первый раз, то появится окно, в котором будет предложено ввести имя сохраняемого файла, место хранения (папку, куда записать; по умолчанию — это рабочая директория, которой является папка «Документы» Рабочего стола), расширение сохраняемого файла (предлагаемое расширение **.R** — расширение для **R**, можно сохранить с расширениями для **S** — **.q**, **.ssc** и **.S**, наконец, можно сохранить с любым иным расширением, например **.txt**). В дальнейшем, при нажатии **Сохранить** скрипт будет сохраняться в ранее выбранном месте с выбранным именем и расширением.

– **Сохранить как** — появится окно, в котором будет предложено ввести имя сохраняемого файла, место его хранения и расширение сохраняемого файла.

• **Правка** — здесь остановимся только на двух пунктах:

– **Запустить строку или блок** — запуск на исполнение отдельной выделенной строки или выделенного блока программы (**Ctrl**+**R**).

– **Запустить всё** — запуск на исполнение набранной в скрипте программы.

2.1.4 Выход из программы, сохранение данных

Выйти из программы можно несколькими способами:

1. Первый способ — набрать команду

```
> q()
```

Появится диалоговое окно, предлагающее сохранить рабочее пространство. **Да** — сохраняются все команды, набранные за рабочую сессию, а также результаты их исполнения. При следующем запуске **R** они будут доступны. **Нет** — выход из **R** без сохранения информации.

2. Выход через командное меню. Заходим в **Файл** и выбираем **Выйти**. Далее всё аналогично предыдущему случаю. Также зайдя в **Файл** и выбирая **Сохранить рабочее пространство** или **Сохранить историю команд**, можно либо полностью сохранить всё набранное за время работы, либо только команды.

Если при выходе из **R** на вопрос о сохранении рабочего пространства был дан положительный ответ, то в рабочей директории будут созданы два файла: бинарный .RData и текстовый .Rhistory. В первом содержатся все объекты, созданные за время сессии. Во втором — полная история введённых команд.

Всю текущую рабочую сессию можно сохранить и при помощи пункта **Сохранить в файл** командного меню **Файл**. Для сохранения предлагается файл с расширением .txt в рабочей директории.

2.2 Полезные команды

Перечислим самые главные и полезные команды — команды вызова помощи **help()** и **help.search()**.

Если точно известно название функции и нужно получить дополнительно информацию о её аргументах, о результатах её исполнения, то можно (и нужно) набрать в командной строке **help(имя_функции)**. В результате — окно с полной справкой о нужной функции. Варианты написания: **help("имя_функции")** или **help('имя_функции')**. Также можно в командной строке написать **?имя_функции**. В результате всё равно получим справочную информацию о нужной функции.

Пример 2. *Возъмём команду* **ls()** *и на её примере разберём структуру выводимой справочной информации.*

```
> help(ls)
```

В результате будет получено:

```
ls {base}                                                    R Documentation
                                List Objects
Description

ls and objects return a vector of character strings giving the names of the
 objects in the specified environment. When invoked with no argument at the
top level prompt, ls shows what data sets and functions a user has defined.
When invoked with no argument inside a function, ls returns the names of the
functions local variables. This is useful in conjunction with browser.

Usage

ls(name, pos = -1, envir = as.environment(pos),   all.names = FALSE, pattern)
objects(name, pos= -1, envir = as.environment(pos), all.names = FALSE, pattern)

Arguments

name     which environment to use in listing the available objects. Defaults to
```

```
         the current environment. Although called name for back compatibility,
         in fact  this argument can specify the environment in any form; see
         the details section.

pos      an alternative argument to name for specifying the environment as a
         position in the search list. Mostly there for back compatibility.

envir    an alternative argument to name for specifying the environment
         evaluation environment. Mostly there for back compatibility.

all.names   a logical value. If TRUE, all object names are returned. If FALSE,
            names which begin with a . are omitted.
pattern     an optional regular expression. Only names matching pattern are
            returned. glob2rx can be used to convert wildcard patterns to
            regular expressions.

Details

The name argument can specify the environment from which object names are taken
 in one of several forms: as an integer (the position in the search list); as
 the character string name of an element in the search list; or as an explicit
 environment (including using sys.frame to access the currently active function
 calls). By default, the environment of the call to ls or objects is used. The
 pos and envir arguments are an alternative way to specify an environment, but
 are primarily there for back compatibility.

Note that the order of the resulting strings is locale dependent, see Sys.getlocale.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language.
 Wadsworth & Brooks/Cole.

See Also

glob2rx for converting wildcard patterns to regular expressions.

ls.str for a long listing based on str. apropos (or find) for finding objects in
 the whole search path; grep for more details on 'regular expressions'; class,
 methods, etc., for object-oriented programming.

Examples

.Ob <- 1
ls(pattern = "O")
ls(pattern= " O", all.names = TRUE)    # also shows ".[foo]"

# shows an empty list because inside myfunc no variables are defined
myfunc <- function() {ls()}
```

```
myfunc()

# define a local variable inside myfunc
myfunc <- function() {y <- 1; ls()}
myfunc()                # shows "y"
```

Структура справочной информации:

- на первой строке приведены название функции и пакет, в которой она находится;
- **Description** — описание функции, для чего она используется, что она делает;
- **Usage** — правильное написание функции со всеми её аргументами;
- **Arguments** — подробное описание аргументов функции,какие значения они могут принимать;
- **Details** — различные детали;
- **References** — ссылка на разработчиков;
- **See Also** — иные рекомендуемые для просмотра функции;
- **Examples** — примеры использования.

Если же название функции известно не полностью (или не уверены в названии, или известен аргумент функции, или результат реализации функции), то можно воспользоваться **help.search("имя")**[6]. Результатом исполнения команды откроется окно справки в котором приведён список пакетов и функций из этих пакетов, содержащих указанное в кавычках имя.

Пример 3. `> help.search("norm")`

В результате получаем окно со следующей информацией:

```
Help files with alias or concept or title matching ‘norm’ using
regular expression matching:


boot::norm.ci         Normal Approximation Confidence Intervals
MASS::bandwidth.nrd
                      Bandwidth for density() via Normal
```

[6]Кавычки обязательны

```
                    Reference Distribution
MASS::mvrnorm       Simulate from a Multivariate Normal
                    Distribution
Matrix::ddenseMatrix-class
                    Virtual Class "ddenseMatrix" of Numeric
                    Dense Matrices
Matrix::dgeMatrix-class
                    Class "dgeMatrix" of Dense Numeric (S4
                    Class) Matrices
Matrix::diagonalMatrix-class
                    Class "diagonalMatrix" of Diagonal
                    Matrices
Matrix::dspMatrix-class
                    Symmetric Dense Numeric Matrices
Matrix::dtpMatrix-class
                    Packed triangular dense matrices
Matrix::%*%,dtrMatrix,dtrMatrix-method
                    Triangular, dense, numeric matrices
Matrix::ldenseMatrix-class
                    Virtual Class "ldenseMatrix" of Dense
                    Logical Matrices
Matrix::ndenseMatrix-class
                    Virtual Class "ndenseMatrix" of Dense
                    Logical Matrices
Matrix::norm        Matrix Norms
Matrix::sparseMatrix-class
                    Virtual Class "sparseMatrix" -- Mother of
                    Sparse Matrices
nlme::qqnorm.gls    Normal Plot of Residuals from a gls Object
nlme::qqnorm.lm     Normal Plot of Residuals or Random Effects
                    from an lme Object
nnet::nnet          Fit Neural Networks
stats::Lognormal    The Log Normal Distribution
stats::Normal       The Normal Distribution
stats::qqnorm       Quantile-Quantile Plots
stats::shapiro.test
                    Shapiro-Wilk Normality Test
utils::normalizePath
                    Express File Paths in Canonical Form
```

Схожая с **help.search()** команда **apropos("имя")** возвращает список

функций, содержащих указанное в кавычках имя.

Пример 4. *Команда* **apropos("имя")**.

```
> apropos("norm")
 [1] "dlnorm"         "dnorm"          "normalizePath"  "plnorm"
 [5] "pnorm"          "qlnorm"         "qnorm"          "qqnorm"
 [9] "qqnorm.default" "rlnorm"         "rnorm"
```

Получили список из 11 функций, в название которых входит «norm».

Ещё одна команда — **help.start()** открывает HTML-помощь в окне браузера.

Кроме того, необходимую информацию можно получить и из **Справки** командного меню:

- **Консоль** — справка по работе в консоли. Включает в себя информацию по «горячим клавишам», полезным при работе в режиме командной строки
- **ЧаВО по R** — ответы на наиболее часто встречаемые вопросы по **R**.
- **ЧаВО по R для Windows** — аналогично предыдущему пункту, только уже с учётом взаимодействия **R** и Windows.
- **Руководства (в PDF)** — руководства[7] по работе в **R**: краткое введение (100 страниц) в работу со статистическим пакетом; полное руководство по всем функциям, входящим в базовую установку (1719 страниц); руководство по чтению/записи данных (34 страницы); введение в **R**-язык программирования; руководство по написанию приложений (новых пакетов) на 138 страницах; руководство по внутренней структуре **R** (51 страница); руководство по установке и администрированию **R**.
- **Функции R (текст)** — выводится окно, где нужно написать функцию, по которой нужна справка (аналог команды **help(имя_функции)**).
- **Гипертекстовая справка** — справка в режиме гипертекста.
- **Искать помощь** — аналог команды **help.search()**.
- **search.r-project.org** — поиск нужного слова в архивах списков рассылки и в документации на сайте r-project.org.

[7] к сожалению, только на английском языке

- **Подсказка команд** — аналог команды **apropos("имя")**.
- **Домашняя страница проекта R** — открытие домашней интернет-страницы проекта **R**.
- **Домашняя страница архива CRAN** — открытие домашней интернет-страницы проекта **CRAN**.

Другие полезные команды:

- **ls()** и **objects()** выводят на экран все созданные в течение сессии объекты (очень полезно, если программа большая);
- **rm(имя_объекта)** — удаляет объект с указанным именем.
- **example(имя_функции)** — выводит пример (если есть) для выбранной функции.
- **history(n)** — будет выведено новое окно, в котором перечислены n последних команд.
- **getwd()** — вывод текущей (рабочей директории)

```
> getwd()
[1] "C:/Users/1/Documents"
```

- **setwd("имя_новой_рабочей_директории")** — смена рабочей директории (при помощи **Изменить папку** в **Файле** также это можно сделать).
- **dir()** — выводит список файлов в рабочей директории.
- **source('имя_файла.R')** — сохранение **R**-кода в текущей директории в файле с указанным именем и расширением .R
- **sink('имя_файла.расширение')** — перенаправляет вывод с экрана в указанный файл; повторный вызов команды **sink()** закрывает файл.

2.3 Пакеты

Новые пакеты в **R** можно установить двумя способами:

1. Можно зайти в **Пакеты** командного меню и выбрать **Установить пакет(ы)**. Сначала предложат выбрать зеркало **CRAN** — страну, откуда будут скачен пакет. В случае удачи появится список доступных пакетов, откуда нужно выбрать нужный (нужные) и нажать кнопку **OK**. Далее предложат создать персональную библиотеку для скаченных пакетов. После этого нужный пакет установлен. Для его инициализации нужна команда **library(имя_пакета)**.

2. Если по каким-либо причинам **R** самостоятельно не может выйти на зеркала **CRAN**, то можно непосредственно зайти на сайт http://cran.gis-lab.info/web/packages/ и выбрать нужный пакет для установки (Преимущество — можно предварительно посмотреть информацию по пакету, так к каждому пакету прилагается описание входящих в него функций). Все пакеты заархивированы. Скачиваем нужный в выбранную папку. После этого заходим в командном меню в **Пакеты** и выбираем **Установить пакеты из локальных zip-файлов**. В открывшемся окне выбираем папку с нужным пакетом, выбираем пакет — и всё, он загружен. Для его инициализации перед непосредственным использованием снова нужна команда **library(имя_пакета)**.

Как видно, установка новых пакетов в **R** довольно проста.

2.4 Классы объектов, типы данных и структуры объектов в R

2.4.1 Классы объектов в R

Все данные (а следовательно и переменные) в **R** можно поделить на следующие классы:

1. **numeric** — название класса, а также типа объектов. К нему относятся действительные числа. Объекты данного класса делятся на целочисленные (**integer**) и собственно действительные (**double** или **real**).

2. **complex** — объекты комплексного типа.

3. **logical** — логические объекты, принимают только два значения: FALSE (F)[8] и TRUE (T) .

4. **character** — символьные объекты (символьные переменные задаются либо в двойных кавычках (), либо в одинарных (")).

[8] **R** позволяет использовать сокращённое имя объектов (переменных)

5. **raw** — объекты потокового типа (разбираться не будут, но желающие сами могут подробно с ними ознакомиться)

Иерархия типов: **raw** < **logical** < **integer** < **real** < **complex** < **character**.

Numeric

Объект класса **numeric** создаётся при помощи команды **numeric(n)**, где n — количество элементов данного типа. Создаётся нулевой вектор длины n.

```
> x=numeric(5)
> x
[1] 0 0 0 0 0
```

В результате создан нулевой вектор типа **numeric** длины 5.

Объекты типов **integer** и **double** создаются, соответственно, при помощи команд **integer(n)** и **double(n)**, а при помощи функций **is.numeric(имя_объекта)** (**is.double(имя_объекта)**, **is.integer(имя_объекта)**) можно проверить объект на принадлежность к классу **numeric** (**double** и **integer**). Десятичным разделителем для чисел является точка.

```
> x=3.7
```

Пример 5. *Создадим переменные x и y типов* **double** *и* **integer** *соответственно.*

```
> x=double(1)
> x=5
> y=integer(1)
> y=7
> is.integer(x)
[1] FALSE
> is.double(x)
[1] TRUE
> is.integer(y)
[1] FALSE
> is.double(y)
[1] TRUE
> is.numeric(x)
[1] TRUE
> is.numeric(y)
[1] TRUE
```

и

```
> y=integer(1)
> is.integer(y)
[1] TRUE
```

Поясним этот пример. Сначала создали объект x типа **double** и присвоили ему целое число 5. При помощи **is.integer(x)** проверили на принадлежность к типу **integer**. Результат — FALSE. Проверили на принадлежность к **double** и **numeric**, в обоих случаях — TRUE. Т.е., несмотря на присвоение целого числа, переменная x не стала целочисленной. Переменная y была создана целочисленной (y=integer(1)) и ей присвоили значение 7. Но после проверки она оказалась не **integer**, а **double**.

Почему? Ответ прост. По умолчанию, все числа в **R** являются вещественными. Чтобы сделать их целочисленными, надо воспользоваться командой **as.integer(имя_объекта)**.

Пример 6. *Создадим переменную x класса* **double***, затем присвоим ей значение 3 и воспользуемся командой* **as.integer()**.

```
> x=double(1)
> x=5
> is.double(x)
[1] TRUE
> x=as.integer(x)
> is.double(x)
[1] FALSE
> is.integer(x)
[1] TRUE
```

Как видно, переменная x стала целочисленной.

Complex

Теперь перейдём к объектам комплексного типа. Они создаются при помощи функции **complex(length.out = 0, real = numeric(), imaginary = numeric(), modulus = 1, argument = 0)**. Обращение

```
> z=complex(5)
> z
[1] 0+0i 0+0i 0+0i 0+0i 0+0i
```

создаёт нулевой комплексный вектор[9]. В качестве аргумента взято значение **length.out** = **5**. Параметр **length.out** задаёт длину выводимого вектора (числовой параметр), **real** и **imaginary** задают действительную и мнимую части комплексного вектора (оба эти аргумента —- числовые вектора).

```
> z=complex(5,c(1:5),c(6:10));z
[1] 1+ 6i 2+ 7i 3+ 8i 4+ 9i 5+10i
```

Названия аргументов функций в **R** можно опускать и писать только их значения, но это только в том случае, если аргументы идут по порядку.

Параметры **modulus** и **argument** — альтернативный способ задания комплексного вектора через его модуль ρ и аргумент φ[10].

```
> z=complex(5,modulus=c(1:5),argument=c(6:10));z
[1]  0.9601703-0.2794155i  1.5078045+1.3139732i -0.4365001+2.9680747i
[4] -3.6445210+1.6484739i -4.1953576-2.7201056i
```

Наконец, есть ещё один довольно простой способ задания комплексного числа:

```
> x=4
> y=5
> z=x+1i*y;z
[1] 4+5i
```

Выражение 1i соответствует мнимой единице $\mathrm{i} = \sqrt{-1}$.

Проверка принадлежности объекта комплексному типу производится при помощи функции **is.complex(имя_объекта)**.

```
> x=5+1i+10
> is.complex(x)
[1] TRUE
> is.numeric(x)
[1] FALSE
```

Объекты класса **numeric** не являются комплексными, комплексные объекты не являются объектами типа **numeric**. Перевод некоторого объекта в класс **complex** можно сделать при помощи команды **as.complex(имя_объекта)**, при этом мнимая часть комплексного вектора будет нулевой.

```
> x=3.7
> y=as.complex(x);y
[1] 3.7+0i
```

[9]Более подробно о векторах будет изложено в разделе 5.2 главы 5.

[10]Напомним представление комплексного числа z: $z = x + iy$, где $x = \rho\cos(\varphi)$, $y = \rho\sin(\varphi)$

Перевод комплексных объектов в действительные можно сделать при помощи функции **as.numeric(имя_объекта)**, либо при помощи **as.double(имя_объекта)**.

```
> z=2.3-1i*6
> x=as.numeric(z)
Warning message:
мнимые части убраны при преобразовании
> y=as.double(z)
Warning message:
мнимые части убраны при преобразовании
> x
[1] 2.3
> y
[1] 2.3
```

Как видно из примера, мнимые части при таком преобразовании просто отбрасываются.

Примеры операций над комплексными объектами будут приведены в разделе 3.2.4 главы 3.

Logical

Перейдём теперь к логическим объектам, т.е. объектам типа **logical**. Объекты этого класса принимают два возможных значения: TRUE (истина) и FALSE (ложь)[11], и создаются при помощи команды **logical(n)**, где n — это длина создаваемого вектора.

```
> x=logical(4)
> x
[1] FALSE FALSE FALSE FALSE
```

Как видно из примера, при обращении к команде **logical(5)** был создан логический вектор, состоящий только из FALSE.

Проверка объекта на принадлежность к логическому типу осуществляется при помощи **is.logical(имя_объекта)**.

```
> x=1;y=F
> is.logical(x)
[1] FALSE
> is.logical(y)
[1] TRUE
```

[11] В дальнейшем будем писать просто T и F

Перевести объект в логический можно при помощи функции **as.logical(имя_объекта)**. Используем предыдущий пример.

```
> z=as.logical(x)
> z
[1] TRUE
```

Переменной x было присвоено значение 1, и потом эта переменная была переведена в логический тип. При таком переходе 1 становится TRUE, а 0 — FALSE. Если перевести логические объекты в числовые (при помощи **as.numeric()**, **as.double()** или **as.integer()**), то все логические значения F перейдут в 0, а T — в 1.

```
> x=F;y=T
> z1=as.numeric(x);z1
[1] 0
> (z2=as.double(y))
[1] 1
> (z3=as.integer(x))
[1] 0
```

В разделе 3.2.1 главы 3 будут рассмотрены различные операции над логическими объектами, а в разделе 5.2.4 главы 5 будут описаны логические вектора.

Character

Последний класс объектов, который будет рассмотрен в данном разделе — это объекты типа **character**, т.е символьные объекты. Создаются при помощи команды **character(n)**, результат — пустой символьный вектор размерности n.

```
> x=character(1);x
[1] ""
```

Проверка на принадлежность к данному типу осуществляется при помощи **is.character()**.

```
> (x='a')
[1] "a"
> (y="a")
[1] "a"
> is.character(x)
```

```
[1] TRUE
> is.character(y)
[1] TRUE
```

Символьные объекты обязательно задаются в кавычках (одинарных или двойных). Символьным объектом может быть как просто символ, так и строка.

```
> x='a';x
[1] "a"
> y='символьный объект';y
[1] "символьный объект"
```

Объекты любого типа можно перевести в символьные. Для этого нужно воспользоваться командой **as.character(имя_объекта)**.

```
> x=F;y=2.4
> (z1=as.character(x))
[1] "FALSE"
> (z2=as.character(y))
[1] "2.4"
```

Символьный же объект перевести в иной тип сложнее.

```
> x='1'
> y=as.numeric(x);y
[1] 1
> x='b'
> y=as.numeric(x);y
Предупреждение
в результате преобразования созданы NA
[1] NA
```

Символьный объект можно перевести в числовой, если он представляет из себя число, окружённое кавычками. Если же в кавычках стоял непосредственно символ (или набор символов), то такой перевод приведёт к появлению **NA** (смотри следующий раздел).

Другой пример.

```
> x="F"
> (y1=as.logical(x))
[1] FALSE
> x='T'
> (y1=as.logical(x))
[1] TRUE
```

Символьные переменные **'T'** и **'F'** можно перевести в логические TRUE и FALSE (Это верно и относительно символьных переменных **'TRUE'** и **'FALSE'**).

Операции над символьными объектами будут рассмотрены в разделе 5.2.2 главы 5.

Тип любого объекта можно проверить (и изменить) при помощи функции **mode(имя_объекта)**.

```
> x='TRUE'
> (y1=as.logical(x))
[1] TRUE
> mode(x)
[1] "character"
> mode(y1)
[1] "logical"
```

Чтобы изменить тип объекта нужно сделать следующее:

```
> x=F
> mode(x)='numeric';x
[1] 0
> mode(x)='character';x
[1] "0"
> mode(x)='complex';x
[1] 0+0i
> mode(x)='logical';x
[1] FALSE
> mode(x)='double';x
[1] 0
```

2.4.2 Специальные переменные в R

В **R** существует ряд особых объектов:

Inf — бесконечность: положительная ($+\infty$ — **Inf**) и отрицательная ($-\infty$ — **-Inf**);

NA — «отсутствующее значение» (Not Available);

NaN — «не число» (Not a Number);

NULL — «ничто».

Все эти объекты можно использовать в любых выражениях. Рассмотрим их более подробно.

Inf появляется при переполнении и в результате операций вида $\frac{a}{0}$, где $a <> 0$.

```
> x=5/0;x
[1] Inf
> y=log(0);y
[1] -Inf
```

Проверить объект на конечность (бесконечность) можно при помощи команд **is.finite()** (**is.infinite()**):

```
> x=5;y=log(0)
> is.finite(x)
[1] TRUE
> is.infinite(y)
[1] TRUE
```

Объект **NaN** — «не число», появляется при операциях над числами, результат которых не определён (не является числом):

```
> x=0/0;x
[1] NaN
> y=Inf-Inf;y
[1] NaN
> y1=log(-2);y1
[1] NaN
> y2=Inf/Inf
> y2=Inf/Inf;y2
[1] NaN
> y3=cos(Inf);y3
[1] NaN
```

При помощи **is.nan(имя_объекта)** можно проверить, является ли объект **NaN**:

```
> x=Inf-Inf
> is.nan(x)
[1] TRUE
```

«Отсутствующее значение» — **NA** — возникает, если значение некоторого объекта не доступно (не задано). Включает в себя и **NaN**. Проверка,относится ли объект к **NA**, делается при помощи **is.na(имя_объекта)**.

```
> x=NaN
> y=NA
> is.na(y)
[1] TRUE
> is.na(x)
[1] TRUE
```

«Ничто» — **NULL** — нулевой (пустой) объект. Возникает как результат выражений (функций), чьи значения не определены. Обнулить объект можно при помощи команды **as.null(имя_объекта)**, проверить объект на принадлежность к **NULL** можно при помощи функции **is.null(имя_объекта)**.

```
> x=7;y=as.null(x)
> is.null(x)
[1] FALSE
> is.null(y)
[1] TRUE
```

При задании аргументов различных функций **R** используется аналогично **NA**.

2.4.3 Создание числовых последовательностей в R

В этом разделе разберём различные варианты задания числовых последовательностей[12].

Если элементы числовой последовательности отличаются друг от друга на единицу (возрастающая последовательность), или на -1 (убывающая последовательность), то можно задать только начальное **a** и конечное **b** значения искомой последовательности — **a:b**.

```
> x=2:10;x
[1]  2  3  4  5  6  7  8  9 10
> y=10:2;y
[1] 10  9  8  7  6  5  4  3  2
```

Второй вариант — при помощи команды **seq()**. Варианты использования:

- **seq(from, to)** — аналогично варианту **a:b**.

[12]Задание числовых векторов рассмотрено в 5.2.3 главы 5

```
> x=seq(from=-5,to=5);x
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
> x=seq(-5,5);x
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5
```

- **seq(from, to, by=)** — задаются начальное, конечное значения и шаг последовательности.

```
> x=seq(-5,5,by=0.5);x
 [1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
[16]  2.5  3.0  3.5  4.0  4.5  5.0
> x=seq(-5,5,0.5);x
 [1] -5.0 -4.5 -4.0 -3.5 -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
[16]  2.5  3.0  3.5  4.0  4.5  5.0
```

(Как видно из примера, не обязательно указывать имя аргумента **by**)

- **seq(from, to, length.out=)** — задаются начальное, конечное значения последовательности, а также общее число элементов последовательности. Шаг последовательности определяется самостоятельно by = ((to − from)/(length.out − 1)).

```
x=seq(-5,5,length.out=10);x
 [1] -5.0000000 -3.8888889 -2.7777778 -1.6666667 -0.5555556  0.5555556
 [7]  1.6666667  2.7777778  3.8888889  5.0000000
```

- **seq(c)** — последовательность строится от 1 до c.

```
> x=seq(6);x
[1] 1 2 3 4 5 6
> y=seq(-4);y
[1]  1  0 -1 -2 -3 -4
```

Другие варианты задания числовых последовательностей (точнее говоря, числовых векторов) будут представлены в разделе 5.2.3 главы 5.

2.4.4 Классы данных в R

Последний раздел этой главы посвящён структурам данных в **R**. Ограничимся только перечислением, так как подробно они рассматриваются в главе 5.

В **R** можно выделить:

- **Вектора** (**vector**) — одномерные массивы, состоящие из элементов одного типа данных. Можно выделить числовые, логические и символьные вектора.
- **Матрицы** (**matrix**) — двумерные массивы, как правило числовые.
- **Многомерные массивы** (**array**) — массивы, чья размерность больше двух.
- **Факторы** (**factor**)— структура, полезная при работе с категориальными данными.
- **Список** (**list**) — это коллекция объектов, доступ к которым можно осуществить по номеру или имени.
- **Таблица данных** (**data.frame**) — наиболее общая структура, используемая при работе в **R**.

Глава 3

Операции над различными переменными. Математика в R

3.1 Предисловие

В этой главе будут рассмотрены операции над различного типа переменными: логическими и комплексными; простейшие математические операции; тригонометрические операции.

3.2 Простейшие операции

3.2.1 Логические операции

В этом разделе будут рассмотрены не только операции над логическими объектами, но и операции, приводящие к возникновению логических объектов.

Таблица 3.1: Логические операции

Запись в R	**Описание**
$!\, x$	логическое отрицание
$x \,\&\, y$	поэлементное логическое **И**
$x \,\&\&\, y$	логическое **И** только для первых элементов векторов x и y
$x \,\vert\, y$	поэлементное логическое **ИЛИ**
$x \,\vert\vert\, y$	логическое **ИЛИ** только для первых элементов x и y
$\mathrm{xor}(x, y)$	если x и y различны, то получим TRUE, иначе FALSE

В следующей таблице приводятся операторы сравнения, результатом работы которых являются логические объекты.

Таблица 3.2: Операторы сравнения

Оператор	**Описание**
$>$	больше
$>=$	больше или равно
$<$	меньше
$<= y$	меньше или равно
$==$	тождество
$!=$	не тождественны

Эти операторы сравнения применимы не только к числовым переменным, но и к символьным. Тогда сравнение происходит в лексикографическом порядке.

```
> 'a'<'b'
[1] TRUE
> 'иэ'>='ия'
[1] FALSE
> 'иэ'!='ия'
[1] TRUE
> 'и'>'э'
[1] FALSE
> 'и'<'э'
[1] TRUE
> 'вг'>='вв'
[1] TRUE
```

Если сравниваются символьный объект и числовой, то символьный будет больше.

```
> x=10000; y='a'; x>y
[1] FALSE
> x=10000;y='a';y>x
[1] TRUE
```

Также можно сравнивать и логические переменные:

```
> x=F;y=T
> x<y
[1] TRUE
```

Логические переменные можно сравнивать как с числовыми, так и с символьными. При этом логическая переменная переводится либо в числовой тип, либо в символьный.

При сравнении любого объекта с **NA** или **NaN** результатом будет **NA**.

```
> x=10;y=NA
> x>y
[1] NA
> x=NA;y=NaN
> x>=y
[1] NA
```

3.2.2 Математические функции

Простейшие математические операции

Числа в **R** являются одним из основных типов данных. Дробная часть числа от целой отделяется точкой (**только точкой**). Показатель отделяется от мантиссы либо символом **e**, либо **E**. В примере

```
> x=0.235;x
[1] 0.235
> x=.235;x
[1] 0.235
> x=23.5e-2;x
[1] 0.235
> x=235E-3;x
[1] 0.235
> x=0.00235e2;x
[1] 0.235
```

представлены различные способы записи одного и того же числа.

В **R** над числами можно выполнять обычные арифметические операции: + (сложение), − (вычитание),∗ (умножение), / (деление), ∧ (возведение в степень), %/% (целочисленное деление), %% (остаток от деления). Операции имеют обычный приоритет, т.е. сначала выполняется возведение в степень, затем умножение или деление, потом уже сложение или вычитание. В выражениях также используются круглые скобки и операции в них имеют приоритет.

Набранное в командной строке арифметическое выражение после нажатия на Enter вычисляется и результат сразу же отображается:

```
> 2*(7-9)^8+6/3
[1] 514
```

Ответ — 514.

Стоит заметить, что все арифметические операции, приведённые выше, являются на самом деле функциями:

```
> '+'(2,3)
[1] 5
> '/'(9,2)
[1] 4.5
```

Рассмотрим элементарные математические функции.

Логарифмические и экспоненциальные функции

Экспоненты:

- **exp(x)** — вычисление e^x;
- **exp1m(x)** — для $|x| \ll 1$ нахождение $e^x - 1$.

В **R** реализовано несколько логарифмических функций:

- **log(x)** и **logb(x)** — натуральный логарифм числа x;
- **log10(x)** — логарифм по десятичному основанию;
- **log2(x)** — логарифм по основанию 2;
- **log1p(x)** — вычисление логарифма $\ln(1+x)$ для $|x| \ll 1$ (при $x \approx -1$ точность вычислений снижается);
- **log(x, base=)** и **logb(x, base=)** — вычисление логарифмов по произвольному основанию.

```
> x=exp(2);x
[1] 7.389056
> log(x)
[1] 2
> logb(x)
[1] 2
> log10(100)
[1] 2
> log(8)
[1] 2.079442
> log1p(0.01)
[1] 0.00995033
> log1p(-0.99999)
[1] -11.51293
> log(125,base=5)
[1] 3
> logb(256,base=16)
[1] 2
```

Функции округления

Доступны следующие функции округления.

- **ceiling(x)** — находит минимальное целое, не меньшее x (округление в сторону $+1$);
- **floor(x)** —возвращает максимальное целое, не превосходящее x (округление в сторону -1);
- **trunc(x)** — отбрасывает дробную часть (округление в сторону 0);
- **round(x)** — округляет к ближайшему целому;
- **round(x, dig)** — общий вид функции **round(x)**, дополнительный параметр dig указывает на число знаков после запятой, до которого нужно произвести округление;
- **signif(x, digits = 6)** — округляет до заданного (digits) числа значащих знаков (т.е. чисел, отличных от нуля), по умолчанию параметр digits равен 6.

```
> x=1.5555
> ceiling(x)
[1] 2
> floor(x)
[1] 1
> trunc(x)
[1] 1
> y=-1.5
> ceiling(y)
[1] -1
> floor(y)
[1] -2
> trunc(y)
[1] -1
> x=1/33;x
[1] 0.03030303
> round(x)
[1] 0
> round(x,4)
[1] 0.0303
> signif(x,5)
[1] 0.030303
```

Модуль и квадратный корень

- **abs(x)** — модуль аргумента x, который может быть как числовым (действительный или комплексный), так и логическим аргументом.

```
> x=-5; y=F;
> abs(x)
[1] 5
> abs(y)
[1] 0
> z=T;abs(z)
[1] 1
> z=5-1i*2
> abs(z)
[1] 5.385165
```

- **sqrt(x)** — корень квадратный переменной x.

```
> x=4;sqrt(x)
[1] 2
> y=-4;sqrt(y)
[1] NaN
Предупреждение
In sqrt(y) : созданы NaN
```

Чтобы найти квадратный корень отрицательного числа x, нужно x определить как комплексное число:

```
> y=as.complex(y); sqrt(y)
[1] 0+2i
```

Специальные функции

К специальным функциям относятся: бета-функция и её логарифм, гамма-функция, логарифм модуля гамма-функции, производные гамма-функции, факториал и число сочетаний.

- **gamma(x)** — гамма-функция $\Gamma(x)$, $\Gamma(x) = \int\limits_0^\infty t^{x-1}e^{-t}\,dt$, где x — действительное число, не равное нулю и не являющееся целым отрицательным.
- **lgamma(x)** — натуральный логарифм абсолютного значения гамма-функции.

- **digamma(x)** — первая производная натурального логарифма гамма-функции.
- **trigamma(x)** — вторая производная натурального логарифма гамма-функции.
- **psigamma(x, deriv)** — вычисляет deriv-производную от ψ-функции (первой производной гамма-функции) (по умолчанию deriv=0, т.е. **psigamma(x)**=**digamma(x)**).

```
> x=-8.9
> gamma(x)
[1] -3.507258e-05
> x=2.15
> gamma(x)
[1] 1.072997
> digamma(x)
[1] 0.5152385
> psigamma(x)
[1] 0.5152385
> trigamma(x)
[1] 0.5894157
```

- **beta(a, b)** — вычисляется значение бета-функции $B(a,b)$ с параметрами a и b, где
$$B(a,b) = \int_0^1 t^{a-1}(1-t)^{b-1}\,dt = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)},$$
параметры a и b больше нуля.
- **lbeta(a, b)** — натуральный логарифм бета-функции $B(a,b)$.

```
> a=2;b=3
> beta(a,b)
[1] 0.08333333
> lbeta(a,b)
[1] -2.484907
```

- **choose(n, k)** — число сочетаний C_n^k из n по k, где n — действительное число, k — целое положительное[1].

[1]При $k < 0$ выдаётся ноль, нецелое k округляется до целого числа

- **lchoose(n, k)** — натуральный логарифм числа сочетаний C_n^k.
- **factorial(x)** — факториал x, где $x \geq 0$.
- **lfactorial(x)** — натуральный логарифм факториала x.

```
> n=5.4;k=3;x=4.6;x1=4
> choose(n,k)
[1] 13.464
> choose(round(n),k)
[1] 10
> factorial(x)
[1] 61.55392
> factorial(x1)
[1] 24
```

3.2.3 Тригонометрические функции

В **R** реализованы следующие тригонометрические функции:

- **sin(x)** — $\sin(x)$, где x задан в радианах (например $\pi/6$);
- **cos(x)** — $\cos(x)$;
- **tan(x)** — $\mathrm{tg}(x)$.

Обратные тригонометрические функции:

- **asin(x)** — $\arcsin(x)$;
- **acos(x)** — $\arccos(x)$;
- **atan(x)** — $\mathrm{arctg}(x)$.

```
> sin(pi/2)
[1] 1
```

где **pi** — это константа π. Если у функции несколько аргументов, то они отделяются знаком , (запятая). Например, **atan2(y,x)** возвращает угол между осью Ox и вектором (x; y).

Гиперболические функции:

- **cosh(x)** — гиперболический косинус аргумента x;
- **acosh(x)** — обратный гиперболический косинус x;

- **sinh(x)** — гиперболический синус x;
- **asinh(x)** — обратный гиперболический синус x;
- **tanh(x)** — гиперболический тангенс x;
- **atanh(x)** — обратный гиперболический тангенс x.

3.2.4 Операции над комплексными переменными

Над комплексными числами в **R** можно производить операции сложения, вычитания, умножения, деления, возведения в степень:

```
> x=5+1i*5
> y=4-1i*5
> x+y
[1] 9+0i
> x-y
[1] 1+10i
> x*y
[1] 45-5i
> x/y
[1] -0.121951+1.097561i
> x^y
[1] 118961-44137.3i
```

Специальные функции для комплексных переменных:

- **Re(z)** — нахождение действительной части комплексного числа z;
- **Im(z)** — мнимая часть комплексного числа z;
- **Mod(z)** — модуль $\rho = \sqrt{x^2 + y^2}$ комплексного числа $z = x + \mathrm{i}y$;
- **Arg(z)** — аргумент φ комплексного числа $z = x + \mathrm{i}y$, где $x = \rho\cos\varphi$, $y = \rho\sin\varphi$;
- **Conj(z)** — комплексно сопряжённое к z число.

К комплексным переменным можно применять и тригонометрические

```
> x=2+1i*2;y=2-1i*3
> Re(x)
[1] 2
> Im(y)
[1] -3
> Mod(x)
[1] 2.828427
> Arg(x)
[1] 0.7853982
> Conj(x)
[1] 2-2i
> cos(x)
[1] -1.565626-3.297895i
```

и кумулятивные функции[2].

[2]Смотри раздел 5.2.3 главы 5

Глава 4

Операторы цикла и условия. Создание собственных функций в R

Несмотря на то, что в рамках проекта **R** написано очень большое число пакетов с разнообразными функциями, каждый пользователь может создавать свои собственные функции, используя в них операторы управления — циклы и условные операторы.

4.1 Операторы цикла и условия в R

В этой части рассмотрим структуры,позволяющие управлять обработкой данных, так называемые операторы управления — циклы и условные операторы.

4.1.1 Оператор if

Начнём с условного оператора **if**, который в **R** имеет две формы записи: краткую (без реализации альтернативы) и полную (возможность реализации альтернативных операций).

Краткий вариант оператора if

```
if(условие) выражение
```

условие — любой оператор условия (<, >, >=, <=, ==, ! =), результатом выполнения которого является логический вектор единичный длины. Если значение вектора **TRUE**, то выполняется **выражение**.

выражение — одно или несколько выражений, выполняемых в случае верности условия. Если задано несколько выражений, то они должны быть заключены в фигурные скобки {} и разделяться точкой с запятой (если на одной строке)[1].

Оператор возвращает значение **выражения** в случае верности условия или ничего не возвращает (**NULL**).

```
> x=5;y=4
> if(x>y) {z=x+y;z}
[1] 9
```

Условие выполнено и в результате получено значение z.

```
> x=5;y=4
> if(x<y) w=x+y
> w
Ошибка: объект 'w' не найден
```

В этом примере условие не выполняется, следовательно, заданное выражение не будет посчитано и объект w не создаётся.

Пример 7. *Рассмотрим ещё один пример, в котором x и y являются векторами одинаковой длины (10). Зададим условие: если x не равен y, то берётся отношение x/y. В результате должны получить вектор, чья размерность совпадает с размерностью исходных векторов. Но из каких элементов он состоит?*

```
> x=1:10; y=10;1
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> y
 [1] 10  9  8  7  6  5  4  3  2  1
> if (x<y) {x/y}
 [1]  0.1000000  0.2222222  0.3750000  0.5714286  0.8333333  1.2000000
 [7]  1.7500000  2.6666667  4.5000000 10.0000000
```

Получается, что взято отношение только первых элементов векторов. Кроме того, выводится предупреждение

```
Предупреждение
In if (x != y) { :
  длина условия > 1, будет использован только первый элемент
```

Таким образом, длина условия должна быть равной единице.

[1] напомним, что в случае блока выражений выводится результат последнего из них

Полный вариант оператора if

```
if(условие) выражение1 else выражение2
```

Если условие выполняется,то вычисляется выражение1 (или блок выражений), в противном случае выражение2 (или блок выражений)[2].

```
> x=5;y=4
> if(x<y) {x+y} else {x-y}
[1] 1
```

4.1.2 Оператор ifelse

Функция

```
ifelse(условие, yes, no)
```

позволяет переменной в зависимости от выполнения (невыполнения) некоторого условия принимать различные значения. Отличие от оператора **if** состоит в том, что здесь условие является логическим вектором любой заданной размерности (зависит от размерности сравниваемых объектов). Порядковый номер элемента логического вектора и его значение определяет какой элемент вектора **yes** или **no** берётся новой переменной.

Пример 8. *Создадим два вектора x и y одинаковой длины и присвоим переменной z либо со знаком $+$ номер совпадающих элементов, либо со знаком $-$ номер несовпадающих элементов.*

```
> x=c(1,3,1,5,1,7,1,9)
> y=c(2,3,4,5,2,7,1,8)
> z=ifelse(x==y,1:10,(-1):(-10))
> z
[1] -1  2 -3  4 -5  6  7 -8
```

Другой пример. Создадим последовательность чисел от 6 *до* -4 *и вычислим корень квадратный.*

```
> x <- c(6:-4)
> sqrt(x)
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000
 [8]      NaN      NaN      NaN NaN
Предупреждение
In sqrt(x) : созданы NaN
```

[2]см.также пример 9

При извлечении квадратного корня из отрицательных чисел были созданы **NaN** *и выведено предупреждение. Попробуем вычислить корень так, чтобы не было предупреждения.*

```
> sqrt(ifelse(x >= 0, x, NA))
 [1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000
 [8]       NA       NA       NA NA
```

Ещё один пример:

```
> x=rep(c(1,2,3),3)
> x=matrix(x,3,3)
> z=cos(ifelse(1<x,x*pi,x*pi/2))
> z
              [,1]          [,2]          [,3]
[1,]  6.123032e-17  6.123032e-17  6.123032e-17
[2,]  1.000000e+00  1.000000e+00  1.000000e+00
[3,] -1.000000e+00 -1.000000e+00 -1.000000e+00
```

4.1.3 Оператор for

```
for(переменная in последовательность) выражение
```

Оператор цикла. Пока **переменная** находится в рамках заданной числовой последовательности, выполняется выражение (или блок выражений).

Пример 9. *Обратимся к примеру 7.*

```
> x=1:10; y=10:1
> x
 [1]  1  2  3  4  5  6  7  8  9 10
> y
 [1] 10  9  8  7  6  5  4  3  2  1
> w=vector(length=10,mode='numeric')
> for(i in 1:10)
+ { if (x[i]<y[i]) {w[i]=x[i]/y[i]}
+ else {w[i]=x[i]*y[i]}
+ }
> w
 [1]  0.1000000  0.2222222  0.3750000  0.5714286  0.8333333 30.0000000
 [7] 28.0000000 24.0000000 18.0000000 10.0000000
```

В цикле мы задали условный оператор, выполнение которого означает, что элемент вектора w равен отношению x/y, невыполнение условия приводит к появлению элемента $w = xy$. Отметим, что до цикла нужно задать вектор w, иначе будет выдано сообщение об ошибке.

Ещё один пример. Снова используем вектора x и y из примера 7, но теперь w — это просто число.

```
> for(i in 1:10)
+ { if (x[i]<y[i]) {w=x[i]/y[i]}
+ else {w=x[i]*y[i]}
+ }
> w
[1] 10
```

В результате переменной w будет присвоено значение, созданное на последнем витке цикла.

Если по каким-либо причинам последовательность, задающая число витков в цикле, имеет нулевую длину, то цикл выполняться не будет.

```
c=integer(0)
for (i in c) {m=i}
m
Ошибка: Объект 'm' не найден
```

4.1.4 Оператор while

Ещё один оператор цикла.

```
while(условие) выражение
```

Пока выполняется условие вычисляется выражение, как только результатом условия становится **FALSE**, выходим из цикла.

```
> x=-10
> while (x<0) {z=x; x=x+1}
> z
[1] -1
```

4.1.5 Операторы repeat, break и next

Оператор **repeat**

```
repeat выражение
```

создаёт бесконечный цикл, в котором вычисляется выражение (или блок выражений). Для выхода из этого цикла нужно использовать **break**, а также условный оператор в качестве одного из выражений.

Пример 10. *Построим цикл при помощи оператора* **repeat**, *при помощи условного оператора* **if** *и* **break** *зададим выход. Результатом работы оператора* **repeat** *будет вычисление натурального логарифма абсолютного значения переменной* t.

```
> t=-10
> repeat{
+ if (t>0) break
+ f=log(abs(t))
+ t=t+1}
> f
[1] -Inf
```

Как только t *стало больше* 0 *выходим из цикла с помощью оператора* **break** *и получаем последнее вычисленное значение* $f = \ln|t|$, *равное* $f = \ln 0 = -\infty$.

Оператор

```
next
```

как и **break** может применяться только внутри циклов. Отличие от **break** состоит в том, что происходит не прерывание цикла, а переход на следующий виток.

4.1.6 Оператор switch

Оператор

```
switch(управляющее выражение, альтернативные действия)
```

позволяет выполнять одну из нескольких операций в зависимости от результатов управляющего выражения.

Управляющее выражение возвращает:

- либо целое число (от 1 до числа альтернатив), которое является номером выполняемого действия;
- либо символьную переменную (строку), соответствующую имени выполняемой операции.

Если возвращаемое контрольным выражением значение не соответствует ни номеру выполняемой операции, ни имени, то результатом оператора **switch** будет **NULL**.

Оператор **switch** не является самостоятельным, используется либо внутри функций, либо внутри других управляющих конструкций.

Пример 11. *Создадим числовой вектор x длины 5, элементы которого принимают значения в зависимости от значения i — либо* $\cos\pi$*, либо* e*, либо* $\log_2(4)$*, либо* $\lg_{10}(0.01)$*, либо, наконец, число, соответствующее логическому значению* **TRUE**.

```
> x=numeric(5)
> for (i in 1:5)
+ { x[i]=switch(i,cos(pi),exp(1),log2(4),log10(0.01),TRUE)}
> x
[1] -1.000000  2.718282  2.000000 -2.000000  1.000000
```

4.2 Написание функций в R

4.2.1 Стандартная форма задания функции в R

Создать собственную функцию в **R** довольно просто. Общий вид:

```
function(аргументы) { выражение}
```

Здесь:

- **function** — ключевое слово, сообщающее **R** о том, что будет создана функция;
- **аргументы функции** — список формальных аргументов (может иметь произвольную длину), от которых зависит **выражение**Б аргументы разделяются запятой. Формальным аргументом может быть: символ (т.е. имя переменной, например, x или y), выражение вида **символ=выражение** (например, **x=TRUE**), специальный формальный аргумент — троеточие **...**.
- **выражение** (тело функции) — представляет собой команду или блок команд (заключённых в фигурные скобки {}), как правило, зависящих от определённых ранее аргументов функции. Отдельные команды в блоке пишутся с новой строки (но можно и на одной строке через **;**).
- Функция в качестве своего значения возвращает результат последнего в фигурных скобках выражения.

Обращение к функции имеет вид **имя(арг1, арг2, ...)**. Здесь значения выражений **арг1, арг2, ...** являются фактическими аргументами, которые подставляются вместо соответствующих формальных аргументов, определённых при задании функции.

Пример 12. *Создадим функцию, вычисляющую норму — корень квадратный скалярного произведения векторов x и y.*

```
> norm = function(x,y) sqrt(x%*%y)
> norm(1:4,2:5)
         [,1]
[1,] 6.324555
```

Если фактические аргументы заданы в именованном виде **имя = выражение**, то они могут быть перечислены в произвольном порядке. Более того, список фактических аргументов может начинаться с аргументов, представленных в обычной позиционной форме, после чего могут идти аргументы в именованной форме. Например, имеется функция fun, заданная следующим образом:

```
> fun = function(arg1, arg2, arg3, arg4)
{
# тело функции опущено
}
```

Тогда fun может быть вызвана многочисленными способами, например:

```
> ans = fun(d, f, 20, TRUE)
> ans = fun(d, f, arg4 = TRUE, arg3 = 20)
> ans = fun(arg1 = d, arg3 = 20, arg2 = f, arg4 = TRUE)
```

Все эти способы эквивалентны.

Во многих случаях при определении функции некоторым аргументам могут быть присвоены значения по умолчанию. Тогда при вызове функции эти аргументы могут быть опущены. Предположим, что функция fun определена как

```
> fun <- function(arg1, arg2, arg3 = 20, arg4 = TRUE)
{
# тело функции опущено
}
```

Тогда обращение к этой функции вида

```
> fun(d, f)
```

эквивалентно трём, приведённым выше. Следующее обращение

```
> fun(d, f, arg4 = FALSE)
```

меняет одно из значений по умолчанию.

Заметим, что в значениях по умолчанию можно использовать любые выражения, в том числе включающие другие аргументы функции.

При вызове функции фактические аргументы заменяют формальные, прописанные при создании функции. Это может происходить разными способами:

- **точное соответствие** — если фактические аргументы заданы в именованном виде, то при вызове функции происходит поиск по полному имени соответствующих формальных аргументов (т.е. ищутся те формальные аргументы, чьё имя полностью соответствует имени фактического аргумента);

- **частичное соответствие** — при вызове функции именованные фактические аргументы могут заменить формальные и в том случае, если имя фактического аргумента совпадает с частью имени формального;

- **позиционное соответствие** — неименованные формальные аргументы заменяются неименованными фактическими согласно порядку их расположения (первый фактический аргумент заменяет первый формальный и т.д.).

- Если останутся несовпавшие фактические аргументы, то будет выведено сообщение об ошибке.

4.2.2 Аргумент ...

При разработке приложений иногда бывает необходимо создать функции для произвольного числа формальных аргументов. Это можно сделать при помощи формального аргумента `...`, означающего, что потом функции будет передано произвольное число фактических аргументов.

Пример 13. *В качестве примера рассмотрим функцию, находящую для произвольного числа векторов их минимальные, максимальные и средние значения.*

```
fun= function (...) {
data = list(...)
n =length(data)
maxs = numeric(n)
```

```
mins = numeric(n)
means<- numeric(n)
for (i in 1:n) {
maxs[i] = max(data[[i]])
mins[i]<-min(data[[i]])
means[i]<-mean(data[[i]])
}
print(maxs)
print(mins)
print(means)
invisible(NULL)
}
```

Зададим теперь три числовых вектора — случайные выборки, подчиняющиеся стандартному нормальному закону

```
x=rnorm(100)
y=rnorm(200)
z=rnorm(300)
```

и вызовем функцию, фактическими аргументами которой являются заданные вектора. В результате получим таблицу

```
>fun(x,y,z)
[1] 2.220520 2.429910 3.541140
[1] -2.185287 -2.436616 -3.001431
[1] -0.13654894 -0.03310537  0.07264300
```

где первая строка — это максимальные значения векторов, вторая строка — минимальные, а последняя — средние значения.

4.2.3 Формальные аргументы, локальные переменные и свободные переменные

Все встречающиеся в теле функции символы делятся на три группы: это формальные аргументы, локальные переменные и свободные переменные.

- Формальные аргументы (формальные параметры) — это аргументы, перечисленные в заголовке функции (в круглых скобках после ключевого слова **function**).
- Локальные переменные — это переменные, не являющиеся формальными аргументами, значения которых определяются во время выполнения функции.

- Переменные, не являющиеся формальными аргументами и локальными переменными, являются свободными переменными.

Например, в функции

```
> f = function(x)
{
y = 2*x
print(x)
print(y)
print(z)
}
```

x — формальный аргумент, y — локальная переменная, z — свободная переменная. Область видимости формальных аргументов и локальных переменных — только сама эта функция. Это означает, что изменения этих переменных внутри функции никак не отражаются на переменных с такими же именами во внешней функции. Область видимости свободных переменных распространяется до внешней функции, в которой они были определены. Изменение таких переменных в теле функции влияет также на соответствующие переменные в этой внешней функции.

4.2.4 Полная форма задания функции в R

Полная форма записи от краткой отличается только дополнительной записью в теле функции:

```
> имя = function(аргументы) {выражение
return(значение)
}
```

return(значение) — возвращаемое функцией значение(я) (возвращаемое значение — вектор единичной длины, в противном случае будет выведено предупреждение).

Пример 14. *Приведём два примера — с* **return(значение)** *и без.*

```
ff=function(x){
y=sum(x)
z=cumprod(x)
}
y=ff(1:10); y
```

Результатом функции является значение переменной z, *так как она последней вычисляется в функции.*

```
 [1]       1       2       6      24     120     720    5040   40320
[9] 362880 3628800
```

Вводим команду возвращения значения.

```
ff1=function(x){
y=sum(x); z=cumprod(x)
return(y)}
ff1(1:10)
```

Получим

```
[1] 55
```

При наличии **return(значение)** для вывода результатов работы функции не нужно присваивать дополнительной переменной результат исполнения функции **y=ff(1:10); y**, а затем вызывать эту переменную. Достаточно обратиться к функции и будет выведено значение той переменной, что указана в **return()**.

Что нужно сделать, чтобы в результате работы функции выводилось два и более значения? Для этого нужно, чтобы переменная в **return** была списком, элементами которого были бы нужные нам значения. Создадим функцию, возвращающую минимальное и максимальное значения числовых векторов x и y.

Пример 15. *Создадим сначала сами вектора* x *и* y.

```
x<-c(1,9,2,8,3,7)
y<-c(9,2,8,3,7,2)
```

Функция:

```
minmax=function (a,b) {
c1=max(a)
c2=max(b)
d1=min(a,b)
d2=min(b)
answer<-list(c1,c2,d1,d2)
names(answer)[[1]]<-"максимальное значение вектора x"
names(answer)[[2]]<-"максимальное значение вектора y"
names(answer)[[3]]<-"минимальное значение вектора x"
names(answer)[[4]]<-"минимальное значение вектора y"
return(answer) }
```

Выводимый результат

```
 minmax(x,y)
$‘максимальное значение вектора x‘
[1] 9
$‘максимальное значение вектора y‘
[1] 9
$‘минимальное значение вектора x‘
[1] 1
$‘минимальное значение вектора y‘
[1] 2
```

4.2.5 Сильное присваивание в R

Действуя обычным образом, меняя значения формальных аргументов внутри функции, вы не меняете соответствующих фактических аргументов. Единственный способ сделать это — использовать в теле функции сильное присваивание $<< -$:

```
arg1 <<- выражение
```

Пример 16. *Рассмотрим пример с сильным присваиванием*[3].

```
y=5;z=6
ff=function(x){
y=sum(x); z<<- sqrt(y)
return(y)}
ff(1:10)
y
z
```

Найденное значение функции

```
> ff(1:10)
[1] 55
```

Значения переменных y и z:

```
> y
[1] 5
> z
[1] 7.416198
```

[3]Если нет $>$, то все выражения написаны в редакторе скриптов

В функции ищется значение переменной y, и именно это значение и выводится как результат выполнения функции. Но найденное внутри функции значение y не влияет на определённое вне функции. Благодаря сильному присваиванию значение z, найденное внутри функции, «перекрывает» значение, присвоенное вне функции.

4.2.6 Команды apply(), sapply() и lapply()

Анонимные функции

С **apply()**, **sapply()** и **lapply()** часто используют **анонимные** функции, пример одной из которых приведён ниже.

```
(function(x,y){ z <- 2*x^2 + y^2; x+y+z })(0:7, 1)
[1]   2   5  12  23  38  57  80 107
```

Создаётся числовой вектор, но сама функция не обладает именем.

Команда apply()

Команда **apply()** используется тогда, когда нужно применить какую-нибудь функцию к строке или столбцу матрицы (таблицы данных). Полная форма записи:

```
apply(X, указатель, функция, ...)
```

Здесь:

- **X** — имя матрицы или таблицы данных;
- **указатель** — указываем, к чему применяем функцию: **1** — к строкам, **2** — к столбцам, **c(1,2)** — к строкам и столбцам одновременно (если функция применяется ко всем элементам матрицы и результат — матрица, то определяется порядок вывода элементов);
- **функция** — имя применяемой функции, Если нужно применить простые операции вида $+$, **-** и т.д., то их необходимо задать в кавычках.

Покажем на примере.

Пример 17. *Сначала создадим матрицу X.*

```
> (X=matrix(1:25,nrow=5))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Найдём сумму элементов по строкам

```
apply(X,1,sum)
[1] 55 60 65 70 75
```

и по столбцам

```
> apply(X,2,sum)
[1]  15  40  65  90 115
```

В обоих случаях получили вектора, чьи длины соответствуют числу столбцов и строк соответственно.

Найдём корень квадратный по всем элементам матрицы.

```
apply(X,1,sqrt)
         [,1]     [,2]     [,3]     [,4]     [,5]
[1,] 1.000000 1.414214 1.732051 2.000000 2.236068
[2,] 2.449490 2.645751 2.828427 3.000000 3.162278
[3,] 3.316625 3.464102 3.605551 3.741657 3.872983
[4,] 4.000000 4.123106 4.242641 4.358899 4.472136
[5,] 4.582576 4.690416 4.795832 4.898979 5.000000
```

Здесь извлекается корень квадратный из всех элементов матрицы и вывод происходит последовательно по строкам.

Применим собственную функцию:

```
> apply(X,1,function(x) x^2-x)
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    2    6   12   20
[2,]   30   42   56   72   90
[3,]  110  132  156  182  210
[4,]  240  272  306  342  380
[5,]  420  462  506  552  600
```

Элементы выводятся по строкам: сначала по первой строке, затем по второй и т.д.

Команда sapply()

Функция **sapply()** применяется аналогично **apply()**, но только к векторам и спискам. Она полезна при сложных итерационных вычислениях, так как позволяет избежать создания циклов (смотри вычисление факториалов в 4.2.7).

```
sapply(X, функция, ..., simplify = TRUE, USE.NAMES = TRUE)
```

- **X** — объект, к которому применяется команда;
- **функция** — применяемая к объекту **X** функция;
- **simplify** — логический аргумент: нужно ли представлять выводимый результат в виде вектора или матрицы (значение **TRUE** — по умолчанию);
- **USE.NAMES** — логический аргумент: если данный аргумент принимает значение **TRUE** (по умолчанию) и **X** символьного типа, то в качестве названий для выводимых результатов используется **X**.

В результате применения **sapply()** создаётся список той же размерности, что и объект **X**.

Пример 18. *Создадим список, элементами которого являются три вектора — два числовых и один логический.*

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
x
$a
 [1]  1  2  3  4  5  6  7  8  9 10
$beta
[1]  0.04978707  0.13533528  0.36787944  1.00000000  2.71828183
[6] 7.38905610 20.08553692
$logic
[1]  TRUE FALSE FALSE  TRUE
```

Теперь с помощью **sapply()** *находим квантили для вышеуказанных векторов.*

```
> sapply(x, quantile)
         a        beta logic
0\%    1.00  0.04978707   0.0
25\%   3.25  0.25160736   0.0
50\%   5.50  1.00000000   0.5
75\%   7.75  5.05366896   1.0
100\% 10.00 20.08553692   1.0
```

Команда lapply()

Функция

```
lapply(X, функция, ...)
```

является версией функции **sapply()**, а именно:

```
sapply(X, функция, ..., simplify = FALSE, USE.NAMES = FALSE)
```

lapply() является полезной при работе со списками, позволяет применять различные функции к элементам списка (числовые функции можно применять только в том случае, если все элементы списка **X** принадлежат классу **numeric**).

Пример 19. *Создадим список из трёх компонент — символьного, числового и логического векторов.*

```
a=c("a","b","c","d")
b=c(1,2,3,4,4,3,2,1)
c=c(T,T,F)
X=list(a,b,c)
```

Проверим, к какому классу принадлежит созданный объект и выведем его элементы на экран.

```
 class(X)
[1] "list"
 X
[[1]]
[1] "a" "b" "c" "d"
[[2]]
[1] 1 2 3 4 4 3 2 1
[[3]]
[1]  TRUE  TRUE FALSE
```

Найдём при помощи **lapply()** *длины элементов списка*

```
> lapply(X,length)
[[1]]
[1] 4
[[2]]
[1] 8
[[3]]
[1] 3
```

и средние значения

```
> lapply(X,mean)
[[1]]
[1] NA
[[2]]
[1] 2.5
[[3]]
[1] 0.6666667
```

Кроме того выведено сообщение

```
Предупреждение
In mean.default(X[[1L]], ...) :
  аргумент не является числовым или логическим: возвращаю NA
```

Это логично, так как первый элемент списка — символьный вектор и к нему не применимы числовые функции.

4.2.7 Примеры написания функций в R с использованием управляющих конструкций

В этой части разберём несколько вариантов функций, позволяющих вычислить факториал x ($x!$), используя циклы и условные операторы.

С помощью операторов **if** и **for**:

```
fac1=function(x){
f=1
if (x<2) return(1)
for (i in 2:x) f = f*i
f}
```

Вызов функции и результаты:

```
> sapply(0:5,fac1)
[1]   1   1   2   6  24 120
```

С помощью **while**:

```
fac2=function(x) {
f = 1
t = x
while(t>1) {
f = f*t
t = t-1 }
return(f) }
```

Используя оператор **while**, важно не забыть про контрольную переменную (t в нашем случае) и про изменение её значений на каждой итерации ($t = t - 1$). Применяя созданную функцию для последовательности чисел от 0 до 5, получим

```
> sapply(0:5,fac2)
[1]   1   1   2   6  24 120
```

Наконец, при помощи оператора «бесконечного» цикла **repeat**:

```
fac3=function(x) {
f = 1
t = x
repeat {
if (t<2) break
f = f*t
t = t-1 }
return(f) }
```

При использовании **repeat** важно не забыть про команду прерывания цикла **break**.

Результаты исполнения функции аналогичны предыдущим:

```
> sapply(0:5,fac3)
[1]   1   1   2   6  24 120
```

Построения циклов можно избежать, если использовать встроенные функции. Для нашего примера такой полезной функцией является **cumprod()**[4].

```
fac4=function(x) max(cumprod(1:x))
```

Функция **max()** нужна для того, чтобы получить факториал нуля, равный единице ($0! = 1$). Вызов функции **fac4** снова приводит к уже знакомым результатам.

```
sapply(0:5,fac4)
[1] 1 1 2 6 24 120
```

[4]См. раздел 5.2.3 главы 5

Глава 5

Классы данных в R

5.1 Предисловие

В этой главе рассмотрим следующие классы (форматы, структуры) данных :

- **Векторы** (**vector**) — одномерные массивы, состоящие из элементов одного типа данных. Можно выделить числовые, логические и символьные вектора.
- **Матрицы** (**matrix**) — двумерные массивы, как правило числовые.
- **Многомерные массивы** (**array**) — массивы, чья размерность больше двух.
- **Факторы** (**factor**)— структура, полезная при работе с категориальными данными, позволяет определить различные категории данных.
- **Список** (**list**) — это коллекция объектов, доступ к которым можно осуществить по номеру или имени; список похож на вектор, но его элементы могут быть различных типов.
- **Таблица данных** (**data.frame**) — наиболее общая структура, используемая при работе в **R**.

5.2 Векторы

5.2.1 Способы задания векторов в

В **R** можно выделить несколько способов задания векторов :

- создаётся нулевой вектор нужного типа (логический, числовой, символьный, комплексный) и заданного размера, в дальнейшем элементам присваиваются значения, отличные от нуля;

- сразу задаётся вектор с нужными элементами.

Первый способ реализуется при помощи функции

```
vector(mode = "тип данных", длина)
```

аргументами которой являются тип вектора (числовой — **numeric**, логический — **logical**, комплексный — **complex**, символьный — **character**), указанный в кавычках, и длина вектора - целое положительное число.

```
> vector('numeric',10)
 [1] 0 0 0 0 0 0 0 0 0 0
> vector('complex',10)
 [1] 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i
> vector('logical',10)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> vector('character',10)
[1] "" "" "" "" "" "" "" "" "" ""
```

Как видно из примера строится нулевой вектор (для логического вектора — только **FALSE**, для символьного — пустой символ).

Если задать только длину вектора

```
vector(10)
```

то будет выдано сообщение об ошибке.

Альтернативой функции **vector()** являются **numeric()**, **logical()**, **complex()**, **character()**, которые рассмотрены в разделе 2.4.1 главы 2.

Второй способ задания вектора — непосредственно задать его элементы. Это можно осуществить при помощи функции конкатенации **c()**. Аргументы этой функции являются элементами вектора.

```
> x=c(3,5,-2,4);x
[1]  3  5 -2  4
> y=c(T,F,T,T);y
[1]  TRUE FALSE  TRUE  TRUE
> z=c('a','b','ab','abc');z
[1] "a"   "b"   "ab"  "abc"
```

Если в качестве аргументов функции **c()** задать данные различных типов, то они будут приводиться к единому: логические и числовые данные приводятся к числовому типу данных; логические, числовые и символьные — к символьному типу; действительные и комплексные — к комплексному типу.

```
> x=c(2,3,-2,T,F,T);x
[1]  2  3 -2  1  0  1
> y=c(4,-6,2.8,T,'a',F,3,'abc');y
[1] "4"     "-6"    "2.8"   "TRUE"  "a"     "FALSE" "3"     "abc"
```

В функции **c()** аргументами также могут быть и вектора:

```
> x=c(1,-3.2,2);x
[1]  1.0 -3.2  2.0
> y=c(-0.6,pi,Inf,5);y
[1] -0.600000  3.141593       Inf  5.000000
> z=c(2,x,-1,y);z
[1]  2.000000  1.000000 -3.200000  2.000000 -1.000000 -0.600000  3.141593
[8]       Inf  5.000000
```

Наконец, можно задать вектор, набирая данные на клавиатуре. Это реализуется при помощи **scan()**[1].

Пример 20. *Создадим числовой вектор* **y** *при помощи функции* **scan()**.

```
> y=scan()
```

После этого следует нажать на клавишу **Enter**. *В консоли появится приглашение*

```
1:
```

после чего можно вводить данные (числа), набирая их на клавиатуре. Пробел разделяет числа,

```
1: 2 34
```

нажатие на **Enter** *означает переход на новую строку*

```
3: 13 5 6 7
7:
```

Двойное нажатие на клавишу **Enter** *завершает ввод. Выводится сообщение о количестве считанных элементов вектора.*

[1]Более подробно об этой функции, а также о считывании данных из файлов рассказано в главе 6.

```
Read 6 items
```

Чтобы вывести созданный таким образом вектор, достаточно набрать его имя:

```
> y
[1]  2 34 13  5  6  7
```

Стоит заметить, что при помощи **scan()** можно создавать только числовые вектора.

В заключение рассмотрим две полезных функции: **is.vector()** и **as.vector()**. Первая из них проверяет, является ли аргумент этой функции вектором.

Пример 21. *Создадим два объекта — вектор* **y**, *состоящий из элементов разного типа, и фактор* **z** *(факторы рассмотрены в разделе 5.6 данной главы).*

```
> y=c(T, F, 1, -3, 2,1+1i*2);y
[1]  1+0i  0+0i  1+0i -3+0i  2+0i  1+2i
> z=factor(y)
> z=factor(y);z
[1] 1+0i  0+0i  1+0i  -3+0i 2+0i  1+2i
Levels: -3+0i 0+0i 1+0i 1+2i 2+0i
```

Проверим их на принадлежность к классу векторов.

```
> is.vector(y)
[1] TRUE
> is.vector(z)
[1] FALSE
```

Вторая — **as.vector()**, — переводит свой аргумент в векторы.

Пример 22. *Продолжим пример 21 и переведём созданный фактор в вектор.*

```
> is.vector(z)
[1] FALSE
> w=as.vector(z)
```

Проверим теперь, является **w** *вектором*

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

и выведем его на экран

```
>w
[1] "1+0i"  "0+0i"  "1+0i"  "-3+0i" "2+0i"  "1+2i"
```

Почему в последнем примере вектор стал символьным, а не числовым, будет объяснено далее (раздел 5.6).

5.2.2 Символьные векторы и строки

Задание символьных векторов

Как уже говорилось в разделе 2.4.1 символьные переменные в **R** задаются при помощи двойных или одинарных кавычек, либо же при помощи функции **character()**.

```
y=character(10);y
 [1] "" "" "" "" "" "" "" "" "" ""
>  for (i in 1:length(y)) y[i]=letters[i]
> y
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

При этом не важно, присваивается переменной один символ или целая строка.

```
> x='a';x
[1] "a"
> y='тоже символьная переменная';y
[1] "тоже символьная переменная"
```

Из отдельных символьных переменных можно создавать векторы с помощью функции **c()**.

```
> времена_года=c('зима','весна','лето','осень')
> времена_года
[1] "зима"  "весна" "лето"  "осень"
```

Также для создания символьного вектора можно воспользоваться функцией **character(n)**, задающей пустой символьный вектор длины n.

```
> character(10)
 [1] "" "" "" "" "" "" "" "" "" ""
```

Присвоение значений таким векторам происходит при помощи индексов (см. раздел 5.2.6 данной главы) и управляющих конструкций (см. главу 4).

Если нужно создать символьный вектор, состоящий из прописных или заглавных букв латинского алфавита, то можно воспользоваться функциями **letters** и **LETTERS**.

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
[20] "T" "U" "V" "W" "X" "Y" "Z"
```

Функция paste()

Конкатенацию (склейка) строк осуществляет функция **paste()**. В простейшем варианте, когда элементы — символьные строки, происходит склейка указанных строк со вставкой пробела в качестве разделителя.

```
paste("зима", "первый", "сезон", "года")
[1] "зима первый сезон года"
```

Если аргументы функции **paste()** — массивы, то склеиваются соответствующие компоненты. При этом, если длины массивов различны, происходит циклическая подстановка меньшего из них.

```
> paste(c('зима','весна','лето','осень'),c('-время года'),sep='')
[1] "зима-время года"  "весна-время года" "лето-время года"  "осень-время года"
```

Если один из векторов — числовой, то он автоматически будет конвертирован в символьный:

```
> paste("x", 1:5)
> [1] "x 1" "x 2" "x 3" "x 4" "x 5"
```

Можно заменить разделитель на другой:

```
> paste("x", 1:5, sep = "")
> [1] "x1" "x2" "x3" "x4" "x5"
```

Разбиение строк на символьные переменные

Функция **substr(x, start, stop)** позволяет выделять (заменять) подстроки в строке. Её аргументы:

- **x** — исходная строка;
- **start** — номер элемента строки **x**, с которого начинается выделение;
- **stop** — номер элемента строки **x**, на котором заканчивается выделение.

Пример 23. *Выделим различные подстроки из строки* **сытое брюхо к учению глухо**.

```
phrase='сытое брюхо к учению глухо'
q=character(26)
for (i in 1:26) q[i]=substr(phrase,1,i)
```

Получим следующий символьный вектор **q**

```
> q
 [1] "с"                          "сы"
 [3] "сыт"                        "сыто"
 [5] "сытое"                      "сытое "
 [7] "сытое б"                    "сытое бр"
 [9] "сытое брю"                  "сытое брюх"
[11] "сытое брюхо"                "сытое брюхо "
[13] "сытое брюхо к"              "сытое брюхо к "
[15] "сытое брюхо к у"            "сытое брюхо к уч"
[17] "сытое брюхо к уче"          "сытое брюхо к учен"
[19] "сытое брюхо к учени"        "сытое брюхо к учению"
[21] "сытое брюхо к учению "      "сытое брюхо к учению г"
[23] "сытое брюхо к учению гл"    "сытое брюхо к учению глу"
[25] "сытое брюхо к учению глух"  "сытое брюхо к учению глухо"
```

С помощью Функции **strsplit(x, split=character(0))** можно разбить символьный вектор **x** на отдельные символы.

```
phrase='сытое брюхо к учению глухо'
strsplit(phrase,split=character(0))
[[1]]
 [1] "с" "ы" "т" "о" "е" " " "б" "р" "ю" "х" "о" " " "к" " " "у" "ч" "е" "н" "и"
[20] "ю" " " "г" "л" "у" "х" "о"
```

Если для аргумента **split** задать некое значение (символьное), то разбиение элементов исходного вектора будет происходить относительно заданного значения.

```
 strsplit(phrase,split=' ')
[[1]]
[1] "сытое"  "брюхо"  "к"      "учению" "глухо"
```

Здесь разбиение производилось относительно пробела.

В заключение ещё одна полезная функция, позволяющая определить количество символов в строке или символьном векторе. Это функция **nchar()**.

```
> nchar(phrase)
[1] 26
> nchar(q)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26
```

5.2.3 Числовые векторы

Как уже упоминалось в разделе 5.2.1 векторы в R формируются функцией **c()**.

Аргументами функции **c()** сами могут являться векторами. В этом случае как результат получаем конкатенацию (объединение) этих векторов. Скалярные значения (т. е. числа) воспринимаются **R** как векторы длины 1. Таким образом, аргументами функции **c()** могут быть как векторы, так и скаляры. Например,

```
> c(c(1, 2, 3, 4, 5), 6, c(7, 8))
[1] 1 2 3 4 5 6 7 8
```

Вектор, состоящий из последовательных чисел, можно получить с помощью команды **начальное-значение:конечное-значение**. Например, **1:5**. На эту команду похожа функция **seq()**, которая генерирует арифметическую прогрессию. Нужно задать начальное значение, конечное значение и либо шаг прогрессии:

```
> seq(0, 1, by = 0.1)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

либо количество элементов последовательности:

```
> seq(0, 1, len = 11)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

В результате будет сформирован вектор, состоящий из заданного числа элементов, равномерно распределённых на заданном отрезке.

Функция **rep(v, k)** создаёт вектор, состоящий из k копий вектора **v**, если k — это число:

```
> rep(c(1, 2), 3)
[1] 1 2 1 2 1 2
```

либо, если $k = (k_1, k_2, \ldots, k_n)$ — вектор, то элементы вектора **v** будут повторяться $k_1, k_2, \ldots, k_n$ раз соответственно.

```
> x=rep(c(1,2,3,4,5),c(1,2,3,4,5));x
 [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
```

Также можно создавать вектора при помощи функций, генерирующих случайную последовательность чисел, подчиняющихся одному из реализованных в **R** вероятностных законов[2].

Элементы вектора могут иметь значения **Inf**, **NaN**, **NA**. Например,

[2]Смотри главу 8

```
> age = c(23, NA, NA, 18, 19, Inf, NaN)
```

Над векторами можно выполнять арифметические операции и элементарные функции. Бинарная операция над двумя векторами одинаковой длины производится над каждой парой элементов, и результатом является вектор той же длины, что и исходные. В случае, когда размерность векторов не совпадает, производится приведение длины более короткого вектора к длине длинного. Приведение выполняется следующим образом — элементы вектора дублируются необходимое число раз полностью (если длина большего вектора кратна длине меньшего вектора)

```
> c(1, 2, 3, 4) + c(1, 2)
[1] 2 4 4 6
> c(1, 2, 3, 4) - c(1, 2)
[1] 0 0 2 2
> c(1, 2, 3, 4) * c(1, 2)
[1] 1 4 3 8
> c(1, 2, 3, 4)/c(1, 2)
[1] 1 1 3 2
> c(1, 2, 3, 4)^c(1, 2)
[1] 1 4 3 16
> c(1, 2, 3, 4)/c(1, 0)
[1] 1 Inf 3 Inf
> 2*c(1, 2, 3, 4, 5)
[1] 2 4 6 8 10
```

или частично (в противном случае), при этом выдаётся предупреждение, но результат всё равно вычисляется.

```
> c(3, 1, 4, 1, 5, 9, 2) + c(9, 5)
[1] 12 6 13 6 14 14 11
```

Выражения вида **вектор*число**, **вектор/число**, **вектор+число**, **вектор-число** означают, что каждый элемент вектора умножается или делится на число, либо к каждому элементу вектора прибавляется (вычитается) число

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> x1*2
 [1] 20 -6 12  4 -8 NA  2  4  6  8 10
> x1/4
 [1]  2.50 -0.75  1.50  0.50 -1.00    NA  0.25  0.50  0.75  1.00  1.25
```

```
> x1+5
 [1] 15  2 11  7  1 NA  6  7  8  9 10
> x1-2
 [1]  8 -5  4  0 -6 NA -1  0  1  2  3
```

Элементарные математические функции применяются к каждой компоненте вектора.

```
> x = c(0, pi/2, pi)
> sin(x)
[1] 0.000000e+00 1.000000e+00 1.224606e-16
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> sqrt(x1)
 [1] 3.162278   NaN  2.449490  1.414214  NaN  NA  1.000000  1.414214
 [9] 1.732051 2.000000 2.236068
Предупреждение
In sqrt(x1) : созданы NaN
```

Функции для работы с векторами

В **R** реализовано много полезных функций для работы с векторами, позволяющих избежать использования циклов:

- **length()** — длина вектора.

```
> x=c(1:5,NA,NaN,6:10);x
 [1]   1   2   3   4   5  NA NaN   6   7   8   9  10
> length(x)
[1] 12
```

- **max()** и **min()** — нахождение максимального и минимального элементов в заданном векторе. Если хотя бы один элемент равен **NA**, то результат поиска максимума (минимума) — **NA**, если есть **NaN** — результат **NaN**. Для устранения **NA** (**NaN**) из расчётов достаточно задать **max(x,na.rm=T)** и **min(x, na.rm=T)**. Для числового вектора нулевой длины максимальное значение равно **-Inf**, а минимальное — **Inf**.

```
> x=c(1:5,NA,NaN,6:10);x
 [1]   1   2   3   4   5  NA NaN   6   7   8   9  10
> min(x)
[1] NaN
```

```
> max(x)
[1] NaN
> min(x,na.rm=T)
[1] 1
> max(x,na.rm=T)
[1] 10
```

- **pmax()** и **pmin()** — параллельный максимум (минимум) для любого заданного числа векторов. Результат — вектор, длина которого равна длине максимального из сравниваемых векторов и элементы которого есть максимальные (минимальные) значения сравниваемых векторов, находящиеся на одинаковых позициях. Т.е. $\mathrm{pmax}(x, y) = (\max(x_1, y_1), \max(x_2, y_)), \ldots)$, $\mathrm{pmin}(x, y) = (\min(x_1, y_1), \min(x_2, y_)), \ldots)$. Если вектора-аргумент имеют разную длину, то наименьший вектор приводится при помощи повторения элементов к длине наибольшего вектора.

```
> x=rpois(10,1);x
 [1] 2 0 1 3 1 1 1 1 0 2
> y=rpois(6,1);y
[1] 0 4 2 1 0 0
> pmax(x,y)
 [1] 2 4 2 3 1 1 1 4 2 2
> pmin(x,y)
 [1] 0 0 1 1 0 0 0 1 0 1
```

Если среди сравниваемых элементов исходных векторов есть **NA** (**NaN**), то результатом будет **NA** (**NaN**). Если же в сравниваемых векторах все элементы принимают значения **NA** (**NaN**), то даже используя опцию **na.rm**=**TRUE**, в результате получим

```
> x=c(NA, NaN,NaN)
> y=c(NaN,NA,NaN)
> pmin(x,y,na.rm=T)
[1] NaN  NA NaN
> pmax(x,y,na.rm=T)
[1] NaN  NA NaN
```

- **mean()**[3] — среднее арифметическое вектора. Если есть хотя бы один элемент, чьё значение **NA** (**NaN**), то результатом суммирования также будет **NA** (**NaN**). Чтобы избежать этого, также нужно задать дополнительный аргумент **mean(x,na.rm = TRUE)**.

[3] С этой функцией ещё встретимся в главе 8

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> mean(x1)
[1] NA
> mean(x1,na.rm=T)
[1] 2.6
```

- **range()** — вектор, состоящий из двух элементов — минимального и максимального значений своего аргумента (вектора).

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> range(x1)
[1] NA NA
> range(x1,na.rm=T)
[1] -4 10
```

- **sum()** — сумма элементов вектора. Если есть хотя бы один элемент, чьё значение **NA** (**NaN**), то результатом суммирования также будет **NA** (**NaN**). ЧТобы избежать этого, также нужно задать дополнительный аргумент **sum(x,na.rm = TRUE)**.

```
> x=c(1:5,NA,6:10)
> sum(x)
[1] NA
> sum(x,na.rm=T)
[1] 55
```

- **prod()** — произведение компонент вектора. Если среди элементов исходного вектора есть **NA** (**NaN**), то результатом будет **NA** (**NaN**).Чтобы убрать **NA** (**NaN**) из рассмотрения, нужно задать **prod(x,na.rm=T)**.

```
> x=c(1:5,NA,NaN,6:10);x
 [1]   1   2   3   4   5  NA NaN   6   7   8   9  10
> prod(x)
[1] NA
> prod(x,na.rm=T)
[1] 3628800
>
```

- **sort()** — возвращает вектор той же длины, что и исходный, с элементами, отсортированными в порядке возрастания (по умолчанию), либо в порядке убывания — **sort(x,decreasing=T)** (или **sort(x,dec=T)**). Значения **NA** (**NaN**) автоматически опускаются при рассмотрении.

```
 x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> sort(x1)
 [1] -4 -3  1  2  2  3  4  5  6 10
> sort(x1,dec=T)
 [1] 10  6  5  4  3  2  2  1 -3 -4
> sort(x1,decreasing=T)
 [1] 10  6  5  4  3  2  2  1 -3 -4
```

- **rev(sort())** — сортировка вектора в убывающем порядке.

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> rev(sort(x1))
 [1] 10  6  5  4  3  2  2  1 -3 -4
```

- **rank()** — присваивание рангов элементам вектора в порядке возрастания значения этих элементов в соответствии с одним из заданных методов (**random**, **average**, **first**, **max**, **min**).

Если все элементы вектора различны, то результат присваивания рангов для всех методов один и тот же.

```
x=c(5:1,6,9,10,8,7);x
 [1]  5  4  3  2  1  6  9 10  8  7
random=rank(x,ties='random')
average=rank(x,ties='average')
first=rank(x,ties='first')
max=rank(x,ties='max')
min=rank(x,ties='min')
rbind(x,random,average,first,max,min)
        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x          5    4    3    2    1    6    9   10    8     7
random     5    4    3    2    1    6    9   10    8     7
average    5    4    3    2    1    6    9   10    8     7
first      5    4    3    2    1    6    9   10    8     7
max        5    4    3    2    1    6    9   10    8     7
min        5    4    3    2    1    6    9   10    8     7
```

Если же среди элементов вектора есть повторяющиеся, то ранги одинаковым элементам вектора присваиваются в каждом методе по своему.

```
        [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x        6.0   11 10.0    9   13   12    8    7 10.0   6.0
random   2.0    8  7.0    5   10    9    4    3  6.0   1.0
average  1.5    8  6.5    5   10    9    4    3  6.5   1.5
first    1.0    8  6.0    5   10    9    4    3  7.0   2.0
max      2.0    8  7.0    5   10    9    4    3  7.0   2.0
min      1.0    8  6.0    5   10    9    4    3  6.0   1.0
```

Метод **random** — ранг каждого из одинаковых элементов определяется случайным образом; **average** — одинаковым элементам присваивается среднее арифметическое их рангов; **first** — ранги одинаковых элементов определяются их расположением в векторе; **max** — одинаковым элементам присваивается максимальный из определённых для них рангов; **min** — одинаковым элементам присваивается минимальный из определённых для них рангов.

Аргумент **na.last** функции **rank()** отвечает за **NA**.

```
> last=rank(x,na.last=T)
> fierst=rank(x,na.last=F)
> Na=rank(x,na.last=NA)
> keep=rank(x,na.last='keep')
> rbind(x,last,first,keep)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x        8 11.0   13   NA  7.0 11.0   10   NA   14   7.0
last     3  5.5    7    9  1.5  5.5    4   10    8   1.5
first    5  7.5    9    1  3.5  7.5    6    2   10   3.5
keep     3  5.5    7   NA  1.5  5.5    4   NA    8   1.5
> Na
[1] 3.0 5.5 7.0 1.5 5.5 4.0 8.0 1.5
```

Значение аргумента **na.last=T** — **NA** присваиваются наибольшие ранги среди элементов вектора; **na.last=F** — **NA** присваиваются наименьшие ранги среди элементов вектора; **na.last=NA** — перед присваиванием рангов элементы **NA** удаляются из вектора; **na.last='keep'** — элементам вектора, чьи значения есть **NA**, присваивается ранг **NA**.

- **match(x,y)** — определяет, на каком месте в векторе **y** впервые встречаются элементы вектора **x**.

```
> x
 [1]  8 11 13  9  7 11 10 12 14  7
> y
[1]  9 10  9 10  9
```

```
> match(x,y)
 [1] NA NA NA  1 NA NA  2 NA NA NA
> match(y,x)
[1] 4 7 4 7 4
```

Кумулятивные (накопительные) функции

К кумулятивным функциям в **R** относятся следующие функции:

- **cumsum(x)** — кумулятивная сумма по аргументу **x** (последовательное сложение элементов вектора);
- **cumprod(x)** — кумулятивное произведение;
- **cummax(x)** — кумулятивный максимум (последовательный максимум по элементам **x**);
- **cummin(x)** — кумулятивный минимум (последовательный минимум по элементам вектора).

```
> x=1:10
> cumsum(x)
 [1]  1  3  6 10 15 21 28 36 45 55
> y=-5:5
> cumprod(y)
 [1]   -5   20  -60  120 -120    0    0    0    0    0    0
> x=-5:4
> cummax(x)
 [1] -5 -4 -3 -2 -1  0  1  2  3  4
> cummin(x)
 [1] -5 -5 -5 -5 -5 -5 -5 -5 -5 -5
```

5.2.4 Логические векторы

R умеет работать с логическими векторами (и, следовательно, с логическими скалярами), элементы которого могут иметь значения **TRUE** и **FALSE**, а также значение **NA**. Логические векторы получаются в результате сравнений и применения к логическим векторам логических функций[4]. Операнды могут иметь разную длину. Сравнения и логические функции выполняются поэлементно и, если требуется, с циклическим сдвигом, как и в случае арифметических операций.

[4]Смотри раздел 3.2.1 главы 3

Пример 24. *Создадим логический вектор* **young** *той же длины, что и* **age**, *с компонентами, равными* **TRUE**, *где условие выполнено, и* **FALSE**, *где условие не выполнено.*

```
> age = c(1, 2, NA, Inf, NaN, 18, 19, 40)
> young <- (age >= 2) & (age <= 30)
> young
[1] FALSE TRUE NA FALSE NA TRUE TRUE FALSE
```

Логические векторы могут использоваться в обычной арифметике. При этом **TRUE** интерпретируется как 1, а **FALSE** как 0.

Функция проверки на принадлежность либо к типу данных, либо к классу данных, а также функции проверки на соответствие некоторым значениям возвращают логический вектор той же длины, что и проверяемый объект, с компонентами **TRUE** и **FALSE**.

Пример 25. *Проверим, являются ли элементы вектора* **a** *переменными вида* **NA** *или* **NaN**. *Для этого воспользуемся функцией* **is.na()**.

```
> a = c(0, 1, Inf, NaN, NA)
> is.na(a)
[1] FALSE FALSE FALSE TRUE TRUE
```

В результате получим логический вектор, чья длина совпадает с длиной вектора **a**, *с компонентами, равными* **FALSE**, *где соответствующие значения вектора* **a** *не равны* **NaN** *или* **NA**, *и* **TRUE** *в остальных случаях.*

Функция **is.nan(a)** *также возвращает логический вектор той же длины, что и проверяемый объект (вектор* **a**), *с элементами* **TRUE**, *где соответствующие значения исходного вектора* **a** *равны* **NaN**, *и* **FALSE** *в остальных случаях.*

```
> is.nan(a)
[1] FALSE FALSE FALSE TRUE FALSE
```

5.2.5 Задание имён элементам векторов

Иногда при работе с векторами бывает полезно задать имена элементам вектора. Сделать это можно при помощи функции **names(имя вектора)**.

Пример 26. *Предположим, имеются следующие данные — количество студентов в различных группах (НК-201, НП-201, НП-202, НИ-201, НП-203). Эти числа приведены в векторе* **x**.

```
> group=c(17,19,25,13,7)
```

Однако, только тот,кто составлял этот вектор знает, какой элемент соответствует какой группе.

Присвоим имена каждому элементу вектора, тогда смысл его станет понятен любому.

```
names(group)=c('НП-201','НП-202','НП-203','НК-201','НИ-201')
```

и посмотрим на результат

```
> group
НП-201 НП-202 НП-203 НК-201 НИ-201
    17     19     25     13      7
```

После присваивания имён элементам вектора обращаться к элементам можно как с помощью индекса, так и имени:

```
> group[2]
НП-202
    19
> group['НП-203']
НП-203
    25
> group['НП-203']=18
> group
НП-201 НП-202 НП-203 НК-201 НИ-201
    17     19     18     13      7
```

5.2.6 Векторы и индексы

Индексация векторов

Доступ к элементам вектора осуществляется оператором **[i]**, где i — номер нужного элемента. Например, **u[5]** — это 5-й элемент вектора **u**. Нумерация элементов начинается с 1. Выражения вида **u[i]** могут встречаться и в левой части от знака присваивания. При этом если вектор имеет длину не меньше i, то **u[i]** просто примет новое значение. В противном случае вектор **u** увеличит свою длину до i, элемент **u[i]** примет новое значение, а остальным новые компонентам будут присвоены значения **NA**.

```
> u <- 1
> u[5] <- 5
> u
[1] 1 NA NA NA 5
```

Выражение вида **u[-i]** означает, что будет создан новый вектор путём удаления i-го элемента из исходного вектора **u**.

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> length(x1)
[1] 11
> x1[-6]
 [1] 10 -3  6  2 -4  1  2  3  4  5
```

Обращение вида **u[]** приводит к созданию нового вектора, состоящего из всех элементов исходного вектора **u**.

```
> x1[]
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
```

Векторы в качестве индексов

Пусть **v** — некоторый вектор (числовой, логический, символьный). Напомним, что обращение к конкретному элементу этого вектора осуществляется с помощью команды индексирования **v[i]**. В качестве индекса **x** может выступать не только скаляр, но и вектор. В этом случае строится новый вектор, состоящий из тех элементов вектора **v**, которые удовлетворяют заданным условиям.

Что будет получено в результате применения в качестве индекса вектора? Возможны следующие случаи:

- **x** — это логический вектор. В этом случае желательно, чтобы длины векторов **x** и **v** совпадали. Если логический вектор **x**, используемый в качестве индекса короче вектора данных **v**, то длина вектора **x** доводится до длины вектора **v** цикличным повторением элементов. Если вектор-индекс больше исходного вектора **v**, то вектор **v** доводится до размера вектора **x** добавлением элементов **NA**. Новый вектор **v[x]** состоит только из тех компонент исходного вектора **v**, для которых соответствующее значение в векторе-индексе **x** есть **TRUE**.

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> x2=c(T,F,T,F,F)
> y=x1[x2];y
[1] 10  6 NA  2  5

> x3=rep(c(T,F),10);x3
```

```
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[13]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
> y=x1[x3];y
 [1] 10  6 -4  1  3  5 NA NA NA NA
```

- **x** — это положительный целочисленный вектор. Тогда компоненты вектора **x** интерпретируются как обычные индексы, и **v[x]** — вектор, состоящий из элементов вектора **v** в той последовательности, в которой они указаны в векторе-индексе **x**.

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> x=c(1,3,5,15)
> y=x1[x];y
[1] 10  6 -4 NA
```

- **x** — отрицательный целочисленный вектор. Абсолютные значения вектора **i** интерпретируются как номера элементов, исключаемых из **v**.

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> x=c(-1,-3,-5)
> y=x1[x];y
[1] -3  2 NA  1  2  3  4  5
> y=x1[-(2:6)];y
[1] 10  1  2  3  4  5
```

- **x** — символьный вектор. Его значения интерпретируются как имена элементов вектора **v**. В результате, **v[x]** — вектор, сформированный из соответствующих элементов вектора **v** в той последовательности, в которой они указаны в **x**.

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> names(x1)=letters[1:length(x1)];x1
 a  b  c  d  e  f  g  h  i  j  k
10 -3  6  2 -4 NA  1  2  3  4  5
>
> y=x1[c('a','c','f')];y
 a  c  f
10  6 NA
> y=x1[letters[3:8]];y
```

```
 c  d  e  f  g  h
 6  2 -4 NA  1  2
> y=x1[c('g','a','j','d')];y
 g  a  j  d
 1 10  4  2
```

Также в качестве индексов можно указывать различные операции, приводящие к созданию логического вектора[5]. Например

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> y1=x1[x1<5];y1
[1] -3  2 -4 NA  1  2  3  4
> y2=x1[(-4<x1)&(x1<3)];y2
[1] -3  2 NA  1  2
> y3=x1[(x1<=-2)|(x1>3)];y3
[1] 10 -3  6 -4 NA  4  5
> y4=x1[!((-3<=x1)&(x1<3))];y4
[1] 10  6 -4 NA  3  4  5
```

или

```
> y5=x1[!is.na(x1)];y5
 [1] 10 -3  6  2 -4  1  2  3  4  5
```

Здесь формируется вектор **y5** только из тех компонент вектора **x1**, которые не являются **NA** и **NaN**.

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> x1[8]=NaN;x1
 [1]  10  -3   6   2  -4  NA   1 NaN   3   4   5
> y6=x1[!is.nan(x1)];y6
 [1] 10 -3  6  2 -4 NA  1  3  4  5
```

Команда

```
> x[is.na(x)] <- 0
```

заменяет значения **NA** и **NaN** нулями.

Выражение

[5]Смотри раздел 3.2.1 главы 3

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
 [1] 10 -3  6  2 -4 NA  1  2  3  4  5
> z=(x1 + 1)[(!is.na(x1)) & x1 > 0];z
[1] 11  7  3  2  3  4  5  6
```

создаёт вектор **z** и размещает в нем элементы вектора **x1** + **1** (к каждому элементу вектора **x1** прибавлена 1), удовлетворяющие заданным условиям.

5.2.7 Функция which()

При работе с массивами бывает нужно определить для некоторого элемента(ов) вектора его индекс. Для этого служит функция **which()**, работу с которой рассмотрим на примере.

Пример 27. *Пусть задан вектор* **x**.

```
> x
 [1] 10  4  7  9  7  8  6 14 10 10  5  8 11 11 20
> length(x)
[1] 15
```

Определим номера элементов вектора **x**, *которые больше 10.*

```
> which(x>10)
[1]  8 13 14 15
```

Теперь, если нужно в векторе **x** *найти, к примеру, второй элемент, превосходящий 10, то достаточно указать*

```
> x[13]
[1] 11
```

либо

```
> y=which(x>10)
> x[y[2]]
[1] 11
```

Другой пример.

Пример 28. *Найдём в векторе* **x** *второй элемент, кратный 5.*

```
> x
 [1] 10  4  7  9  7  8  6 14 10 10  5  8 11 11 20
> y=which(x\%\%5==0);y
[1]  1  9 10 11 15
> x[y[2]]
[1] 10
```

С функцией **which()** схожи ещё две функции — **which.max()** и **which.min()**, которые находят, соответственно, номер максимального и минимального элементов вектора.

```
> which.max(x)
[1] 15
> which.min(x)
[1] 2
```

С помощью **which()** можно находить элементы вектора, чьи значения наиболее близки к некоторому заданному.

Пример 29. *Снова воспользуемся вектором* **x** *из примера 27 и найдём все элементы вектора, наиболее близкие к 12*

```
> y=which(abs(x-12)==min(abs(x-12)));y
[1] 13 14
```

и выведем их на экран.

```
> x[y]
[1] 11 11
```

5.3 Матрицы

5.3.1 Задание матрицы

Числовую матрицу можно создать из числового вектора с помощью функции **matrix()**

```
matrix(x, nrow, ncol, byrow, dimnames)
```

Для задания матрицы необходим массив данных **x**, нужно указать число строк **nrow** = **m** и/или число столбцов **ncol** = **n** (по умолчанию, число строк равняется числу столбцов и равно 1); определить как элементы вектора **x** заполняют матрицу — по строкам или по столбцам (по умолчанию матрица заполняется по столбцам). В результате элементы из вектора будут записаны в матрицу указанных размеров. Аргумент **dimnames** — список из двух компонент, первая из которых задаёт названия строк, а вторая — названия столбцов (по умолчанию имена строк и столбцов не задаются).

```
> matrix(1:6, nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    3    5
```

```
[2,]    2    4    6

> matrix(1:6, nrow = 2, ncol = 3, byrow=T)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

> matrix(1:6, nrow = 2, ncol = 3, byrow=T,list(c(1,2),c('A','B','C')))
  A B C
1 1 2 3
2 4 5 6
>
```

Формально нужно, чтобы длина вектора **x** была кратна произведению требуемого числа строк на требуемое число столбцов,

```
matrix(1:2, nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    2    2    2
matrix(1, nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
```

но это не обязательно:

```
> matrix(1:12, nrow = 5, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8    1
[4,]    4    9    2
[5,]    5   10    3
Предупреждение
In matrix(1:12, nrow = 5, ncol = 3) :
  длина данных [12] не является множителем количества строк [5]
```

Если указывается только одна из размерностей (например, только число столбцов), то желательно, чтобы длина вектора была кратна этой размерности. Вторая размерность будет определена как отношение длины вектора к первой размерности.

```
> matrix(1:12, ncol = 3)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Если же число столбцов (строк) не является делителем длины вектора **x**, то матрица всё равно будет построена (правда с предупреждением). Вторая размерность будет определена как ближайшее большее целое число к остатку от деления длины вектора на заданную размерность.

```
> A=matrix(1:12, ncol = 5);A
Предупреждение
In matrix(1:12, ncol = 5) :
  длина данных [12] не является множителем количества столбцов [5]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10    1
[2,]    2    5    8   11    2
[3,]    3    6    9   12    3
> B=matrix(1:12, nrow = 5);B
Предупреждение
In matrix(1:12, nrow = 5) :
  длина данных [12] не является множителем количества строк [5]
     [,1] [,2] [,3]
[1,]    1    6   11
[2,]    2    7   12
[3,]    3    8    1
[4,]    4    9    2
[5,]    5   10    3
```

Функции **nrow(A)**, **ncol(A)** и **dim(A)**возвращают число строк, число столбцов и размерность матрицы **A** соответственно.

```
> nrow(A)
[1] 3
> ncol(B)
[1] 3
> dim(A)
[1] 3 5
```

Функция **cbind(A, B)**

```
> C = cbind(A, B)
```

создаёт матрицу из матриц (векторов), приписывая справа к **A** матрицу (вектор) **B** (для этого число строк у **A** и **B** должно совпадать).

```
> A=matrix(1:12,nrow=3);A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> B=matrix(13:24,nrow=3);B
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
> Z=cbind(A,B);Z
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    4    7   10   13   16   19   22
[2,]    2    5    8   11   14   17   20   23
[3,]    3    6    9   12   15   18   21   24
```

Функция **rbind(A, B)**

```
> A = rbind(A, B)
```

создаёт матрицу, приписывая снизу к матрице **A** матрицу **B** (для этого число столбцов у исходных матриц должно совпадать). Заметим, что в списках аргументов функций **cbind()** и **rbind** можно указать более двух матриц.

```
> Z=rbind(A,B);Z
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
[4,]   13   16   19   22
[5,]   14   17   20   23
[6,]   15   18   21   24
```

Чтобы задать диагональную матрицу достаточно воспользоваться функцией **diag(x,nrow,ncol)**.

```
> diag(1,3,3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Для построения квадратной единичной матрицы нужно задать только число строк **nrow** в матрице (если задать число столбцов **ncol**, то будет выведено сообщение об ошибке).

```
> diag(nrow=4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

Если аргумент **X** функции **diag(X)** есть матрица, то в результате применения функции будет построен вектор из элементов **X**, расположенных на главной диагонали.

```
> X=matrix(1:16,nrow=4);X
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> diag(X)
[1]  1  6 11 16
```

После того, как матрица создана, ее можно изменять, присваивая её элементам новые значения (см. индексы). Есть и другой способ редактирования матрицы. В меню консоли надо выбрать **Правка**, затем нажать **Редактор данных** и в выведенном окне задать имя нужной матрицы (тоже самое можно сделать и при помощи функции **fix(имя объекта)**). В результате будет выведено новое рабочее окно, похожее на страницу **MS Excel**.

Пример 30. *Создадим матрицу* **A**

```
 A=matrix(1:16,nrow=4);A
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

и изменим её в редакторе.

В редакторе также можно присваивать (менять) имена строкам и столбцам.

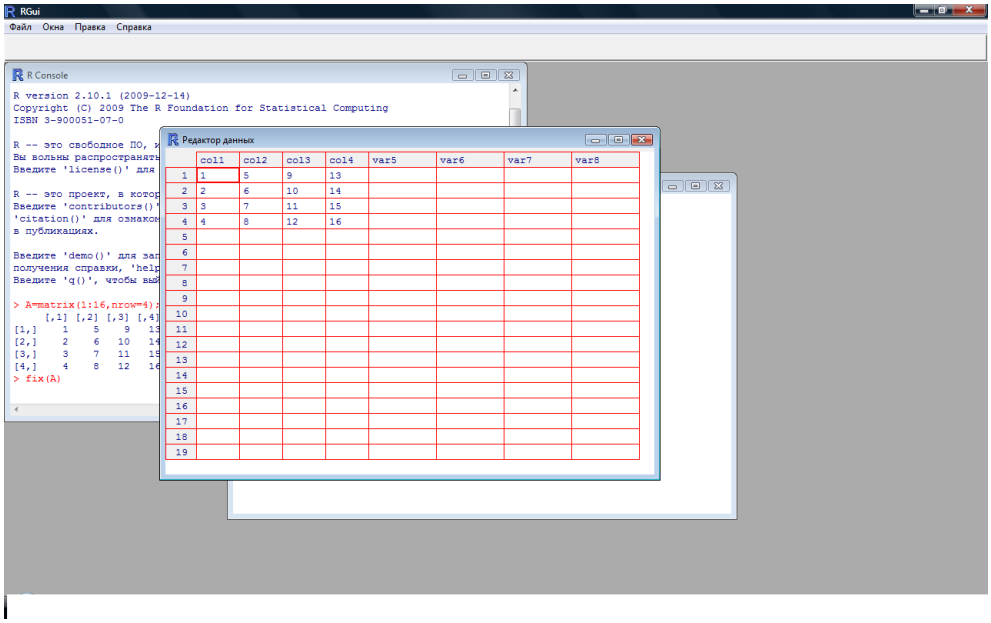

Рис. 5.1: Редактор матриц

5.3.2 Операции над матрицами

Арифметические операции над матрицами осуществляются поэлементно, поэтому, чтобы, к примеру, сложить две матрицы, они должны иметь одинаковые размеры:

```
A = matrix(1:9, nrow = 3);A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
B = matrix(-(1:9), ncol = 3,byrow=T);B
     [,1] [,2] [,3]
[1,]   -1   -2   -3
[2,]   -4   -5   -6
[3,]   -7   -8   -9
A + B
     [,1] [,2] [,3]
[1,]    0    2    4
[2,]   -2    0    2
[3,]   -4   -2    0
```

Впрочем, можно осуществлять смешанные операции, когда один из операндов — матрица, а другой — вектор (в частности, скаляр). В этом случае матрица рассматривается как вектор, составленный из ее элементов, записанных по столбцам, и действуют те же правила, что и для арифметических операций над векторами:

```
> A+3
     [,1] [,2] [,3]
[1,]    4    7   10
[2,]    5    8   11
[3,]    6    9   12
> B+3
     [,1] [,2] [,3]
[1,]    2    1    0
[2,]   -1   -2   -3
[3,]   -4   -5   -6
> (1:3)*A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    4   10   16
[3,]    9   18   27
> A*(1:3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    4   10   16
[3,]    9   18   27
> (1:9)+A
     [,1] [,2] [,3]
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
> B^2
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
[3,]   49   64   81
```

Если длины объектов не кратны, то выводится предупреждение

```
> (0:3)*A
     [,1] [,2] [,3]
[1,]    0   12   14
```

```
[2,]    2    0   24
[3,]    6    6    0
Предупреждение
In (0:3) * A :
  длина большего объекта не является произведением длины меньшего объекта
```

Элементарные математические функции также применяются поэлементно:

```
> sqrt(A)
         [,1]     [,2]     [,3]
[1,] 1.000000 2.000000 2.645751
[2,] 1.414214 2.236068 2.828427
[3,] 1.732051 2.449490 3.000000
> log(abs(B))
         [,1]      [,2]     [,3]
[1,] 0.000000 0.6931472 1.098612
[2,] 1.386294 1.6094379 1.791759
[3,] 1.945910 2.0794415 2.197225
```

Функция **outer(x, y, «операция»)** применяет заданную **операцию** к каждой паре элементов векторов **x** и **y**. Получим матрицу, составленную из результатов выполнения этой операции. Число строк матрицы — длина вектора **x**, а число столбцов — длина вектора **y**.

```
> x =1:5; x
[1] 1 2 3 4 5
> y =-2:3;y
[1] -2 -1  0  1  2  3
> outer(x, y, "*")
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   -2   -1    0    1    2    3
[2,]   -4   -2    0    2    4    6
[3,]   -6   -3    0    3    6    9
[4,]   -8   -4    0    4    8   12
[5,]  -10   -5    0    5   10   15
> outer(x, y, "^")
          [,1]      [,2] [,3] [,4] [,5] [,6]
[1,] 1.0000000 1.0000000    1    1    1    1
[2,] 0.2500000 0.5000000    1    2    4    8
[3,] 0.1111111 0.3333333    1    3    9   27
[4,] 0.0625000 0.2500000    1    4   16   64
[5,] 0.0400000 0.2000000    1    5   25  125
```

Вместо **outer(x, y, "*")** (внешнее произведение векторов) можно использовать **x %o% y**:

```
> x%o%y
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]   -2   -1    0    1    2    3
[2,]   -4   -2    0    2    4    6
[3,]   -6   -3    0    3    6    9
[4,]   -8   -4    0    4    8   12
[5,]  -10   -5    0    5   10   15
```

Транспонирование матрицы осуществляет функция **t(A)**, а матричное произведение — операция %*%:

```
> A = matrix(1:9, nrow = 3);
> B = matrix(-(1:9), ncol = 3,byrow=T)
> A%*%B
     [,1] [,2] [,3]
[1,]  -66  -78  -90
[2,]  -78  -93 -108
[3,]  -90 -108 -126
> B%*%A
     [,1] [,2] [,3]
[1,]  -14  -32  -50
[2,]  -32  -77 -122
[3,]  -50 -122 -194
```

Для решения системы линейных уравнений $\mathbf{Ax} = \mathbf{b}$ с квадратной невырожденной матрицей **A** есть функция **solve(A, b)**:

```
> A = matrix(c(3,4,4,4), nrow = 2);A
     [,1] [,2]
[1,]    3    4
[2,]    4    4
> b=c(1,0)
> solve(A,b)
[1] -1  1
```

Системы линейных уравнений, чьи матрицы коэффициентов имеют верхний треугольный или нижний треугольный вид (т.е, либо все элементы под главной диагональю равны нулю, либо над главной диагональю) можно решать с помощью функций **backsolve (A, b)** и forwardsolve(B b), где **A** и **B** — верхняя треугольная и нижняя треугольная матрицы, **b** — вектор свободных коэффициентов.

```
> A = matrix(c(3,0,4,4), nrow = 2);A
     [,1] [,2]
[1,]    3    4
[2,]    0    4
> b=c(1,1)
> backsolve(A,b)
[1] 0.00 0.25
> B= matrix(c(3,4,0,4), nrow = 2);B
     [,1] [,2]
[1,]    3    0
[2,]    4    4
> forwardsolve(B,b)
[1]  0.33333333 -0.08333333
```

Функция **det(A)** находит определитель матрицы, а **solve(A)** — обратную матрицу:

```
> det(A)
[1] -4
> solve(A)
     [,1]  [,2]
[1,]   -1  1.00
[2,]    1 -0.75
```

В **R** есть функция **determinant()**, полная форма которой

```
determinant(x, logarithm = TRUE, ...)
```

В отличие от **det()** возвращает не скаляр (определитель матрицы), а список, состоящий из двух элементов, первый из которых либо модуль определителя (логический аргумент **logarithm** принимает значение **FALSE**), либо логарифм модуля определителя (**logarithm** = **TRUE**), а второй — либо -1 (определитель отрицателен), либо 1 (определитель неотрицателен).

```
> X
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    1    1
[3,]    4    1    2
> determinant(X)
$modulus
[1] 1.609438
attr(,"logarithm")
```

```
[1] TRUE
$sign
[1] -1
attr(,"class")
[1] "det"
```

Кроме функции **solve()** для нахождения обратной матрицы можно использовать и функцию **ginv()** (но для этого сначала надо подключить пакет **MASS**).

```
> X=matrix(c(1,2,4,2,1,1,3,1,2),nrow=3);X
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    1    1
[3,]    4    1    2
> solve(X)
              [,1] [,2] [,3]
[1,] -2.000000e-01  0.2  0.2
[2,] -3.172066e-17  2.0 -1.0
[3,]  4.000000e-01 -1.4  0.6
> library(MASS)
> ginv(X)
              [,1] [,2] [,3]
[1,] -2.000000e-01  0.2  0.2
[2,] -2.224918e-16  2.0 -1.0
[3,]  4.000000e-01 -1.4  0.6
```

Рассмотрим ещё ряд функций полезных при работе с матрицами. Это:

- **colSums(X, na.rm)** — сумма элементов по столбцам;
- **rowSums(X, na.rm)** — сумма элементов по строкам;
- **colMeans(X na.rm)** — средние значения по столбцам;
- **rowMeans(X na.rm)** — средние значения по строкам.

Аргументы функций: **X** — исходный числовой массив (матрица), **na.rm** — логический аргумент, нужно ли убирать из рассмотрения **NA** (по умолчанию **na.rm**=**FALSE**).

```
> A=matrix(1:12,nrow=3);A
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
```

```
[2,]    2    5    8   11
[3,]    3    6    9   12
> colSums(A)
[1]  6 15 24 33
> rowSums(A)
[1] 22 26 30
> colMeans(A)
[1]  2  5  8 11
> rowMeans(A)
[1] 5.5 6.5 7.5
```

Собственные вектора и собственные числа матриц[6] находятся при помощи

```
eigen(X, symmetric, only.values = FALSE)
```

где **X** — исходная матрица. **symmetric** — логический аргумент, если его значение есть **TRUE**, то предполагается, что матрица **X** — симметричная (Эрмитова для комплексных чисел), и берутся элементы. лежащие только на главной диагонали и под нею. Если аргумент **symmetric** не задан, то матрица будет проверена на симметричность. Логический аргумент **only.values** определяет, нужно ли выводить только собственные числа или ещё и собственные вектора.

```
> X
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    1    1
[3,]    4    1    2
> eigen(X)
$values
[1]  5.892488 -2.266818  0.374330
$vectors
          [,1]       [,2]        [,3]
[1,] 0.5917695  0.7343437 -0.01899586
[2,] 0.3865001 -0.2573049 -0.83006716
[3,] 0.7074083 -0.6281191  0.55733982
```

Возвращает вектор собственных значений, расположенных в порядке убывания их модулей (собственные значения могут быть и комплексными) и матрицу, чьи столбцы есть собственные векторы исходной матрицы.

Функции

[6]Напомним, что собственными значениями матрицы **A** называются корни λ_i характеристического уравнения $|A - \lambda I| = 0$, собственные векторы **x** — векторы, удовлетворяющие $Ax = \lambda_i x$

```
lower.tri(X, diag = FALSE)
```

и

```
upper.tri(X, diag = FALSE)
```

строят логические матрицы(чьи размерности совпадают с размерностью матрицы **X**), в которых на диагоналях ниже главной (**lower.tri(X, diag = FALSE)**) или выше главной (**upper.tri(X, diag = FALSE)**) стоят логические переменные **TRUE**. Аргумент **diag** отвечает за главную диагональ.

5.3.3 Операции с индексами

Доступ к элементам матрицы происходит по индексу. **A[i, j]** ссылается на элемент i-й строки и j-го столбца матрицы **A**. На месте индексов i и j могут стоять векторы.

- **i** и **j** — положительные целочисленные векторы, тогда **A[i,j]** — подматрица матрицы A, образованная элементами, стоящими на пересечении строк с номерами из вектора **i** и столбцов с номерами из **j**.
- **i** и **j** — отрицательные целочисленные векторы, тогда **A[i,j]** — подматрица, полученная из исходной матрицы удалением элементов на соответствующих строках и столбцах.
- **i** и **j** — логические векторы. Тогда **A[i,j]** — новая матрица, состоящая из тех элементов исходной матрицы, для которых элементы векторов **i** и **j** принимают значение **TRUE**.
- **i** и **j** — символьные векторы, т.е.векторы с именами строк и столбцов. Новая матрица образована элементами исходной, находящимися на пересечении столбцов и строк с заданными именами.
- **A[i,]** — эквивалентно **A[i, 1:ncol(A)]**.
- **A[, j]** — эквивалентно **A[1:nrow(A), j]**.

Возможен доступ к элементам матрицы с помощью одного индекса:

- **A[5]** означает 5-й элемент матрицы **A**, если считать, что элементы пронумерованы по столбцам.
- Если **i** — вектор, то **A[i]** означает выборку соответствующих элементов и т. д.

Можно задать имена столбцам и строкам матрицы

```
> A=matrix(1:12,nrow=3)
> rownames(A)=letters[1:nrow(A)]
> colnames(A)=LETTERS[1:ncol(A)]
> A
  A B C  D
a 1 4 7 10
b 2 5 8 11
c 3 6 9 12
>
```

После этого доступ к строкам и столбцам, как уже говорилось выше может происходить по имени.

```
> A['a','C']
[1] 7
> A[2,'B']
[1] 5
> A[,'B']
a b c
4 5 6
> A['b',]
 A  B  C  D
 2  5  8 11
```

5.4 Многомерные массивы

Матрицы — это частный случай многомерных массивов. Матрицы имеют две размерности. В общем случае массивы могут иметь больше размерностей. Работа с многомерными массивами в R во многом аналогична работе с матрицами. Основной способ их создания — функция **array(X, вектор размерностей)**. Указываются элементы массива и все его размерности.

Пример 31. *Создадим массив размерности* $3 \times 5 \times 4$

```
> A=array(1:60, c(3,5,4))
```

При вызове массив выводится послойно.

```
> A
, , 1
     [,1] [,2] [,3] [,4] [,5]
```

```
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
, , 2
     [,1] [,2] [,3] [,4] [,5]
[1,]   16   19   22   25   28
[2,]   17   20   23   26   29
[3,]   18   21   24   27   30
, , 3
     [,1] [,2] [,3] [,4] [,5]
[1,]   31   34   37   40   43
[2,]   32   35   38   41   44
[3,]   33   36   39   42   45
, , 4
     [,1] [,2] [,3] [,4] [,5]
[1,]   46   49   52   55   58
[2,]   47   50   53   56   59
[3,]   48   51   54   57   60
```

Можно задать имена размерностям:

```
> dim1 =c('A', 'B', 'C')
> dim2 =c('X1', 'X2', 'X3', 'X4', 'X5')
> dim3 = c('Зима','Весна','Лето','Осень')
> dimnames(A) = list(dim1, dim2, dim3)
> A
, , Зима
  X1 X2 X3 X4 X5
A  1  4  7 10 13
B  2  5  8 11 14
C  3  6  9 12 15
, , Весна
  X1 X2 X3 X4 X5
A 16 19 22 25 28
B 17 20 23 26 29
C 18 21 24 27 30
, , Лето
  X1 X2 X3 X4 X5
A 31 34 37 40 43
B 32 35 38 41 44
C 33 36 39 42 45
, , Осень
```

```
  X1 X2 X3 X4 X5
A 46 49 52 55 58
B 47 50 53 56 59
C 48 51 54 57 60
```

и после этого обращаться к элементу по имени его строки, столбца, слоя и т. д.

```
> A[,,"Осень"]
  X1 X2 X3 X4 X5
A 46 49 52 55 58
B 47 50 53 56 59
C 48 51 54 57 60
> A[,'X2',"Зима"]
A B C
4 5 6
```

Отметим, что подобным образом происходит работа с массивами, состоящими из символьных строк, логическими массивами и массивами, полученными на основе списков.

5.5 Списки

Списки в **R** — это коллекции объектов, доступ к котором можно производить по номеру или имени.

Список может содержать объекты (компоненты) разных типов, что отличает списки от векторов. Компонентами списка могут быть в том числе векторы и другие списки.

Функция

```
> list(объект1, объект2, ...)
```

создаёт список, содержащий указанные объекты.

Пример 32. *Создадим список* **writer** *с указанными полями: фамилия, имя год рождения, год смерти, логическое поле семейного положения, профессия, ФИО жены (мужа) (если нет — NA), количество детей (если есть), три наиболее известных произведения (три поля).*

```
writer=list("Шекспир", "Уильям", "1564", "1616", T, "драматург",
"Хатауэй Анна",3,"Ромео и Джульетта", "Гамлет", "Отелло")
```

Выведем на экран.

```
>writer
[[1]]
[1] "Шекспир"
[[2]]
[1] "Уильям"
[[3]]
[1] "1564"
[[4]]
[1] "1616"
[[5]]
[1] TRUE
[[6]]
[1] "драматург"
[[7]]
[1] "Хатауэй Анна"
[[8]]
[1] 3
[[9]]
[1] "Ромео и Джульетта"
[[10]]
[1] "Гамлет"
[[11]]
[1] "Отелло"
```

Также список можно создать с помощью **vector("list длина)**. Будет создан пустой список заданной длины. Функция **list()** без аргументов также создаёт пустой список.

Проверка объекта на принадлежность к спискам осуществляется с помощью **is.list()**, перевод объекта в списки — **as.list()**.

К компонентам можно обращаться по номеру. Номер указывается в двойных квадратных скобках после имени списка.

```
> writer[[1]]
[1] "Шекспир"
> writer[[9]]
[1] "Ромео и Джульетта"
```

Можно создавать новые компоненты:

```
writer[[12]]='Англия'
writer
[[1]]
```

```
[1] "Шекспир"
[[2]]
[1] "Уильям"
[[3]]
[1] "1564"
[[4]]
[1] "1616"
[[5]]
[1] TRUE
[[6]]
[1] "драматург"
[[7]]
[1] "Хатауэй Анна"
[[8]]
[1] 3
[[9]]
[1] "Ромео и Джульетта"
[[10]]
[1] "Гамлет"
[[11]]
[1] "Отелло"
[[12]]
[1] "Англия"
```

Имена элементов списка задаются также, как и в случае векторов:

```
> names(writer)<-c("фамилия","имя","год рождения","год сметри",
"семейный статус","профессия","имя жены","число детей",
"произведение1","произведение2","произведение3")
writer
$фамилия
[1] "Шекспир"
$имя
[1] "Уильям"
$‘год рождения‘
[1] "1564"
$‘год сметри‘
[1] "1616"
$‘семейный статус‘
[1] TRUE
$профессия
[1] "драматург"
```

```
$‘имя жены‘
[1] "Хатауэй Анна"
$‘число детей‘
[1] 3
$произведение1
[1] "Ромео и Джульетта"
$произведение2
[1] "Гамлет"
$произведение3
[1] "Отелло"
$<NA>
[1] "Англия"
```

Заметим, что если название поля списка состоит белее чем из одного слова, то это название выводится в одинарных кавычках. Присвоим имя последнему элементу списка.

```
> names(writer)[12]='Страна'
> writer
$фамилия
[1] "Шекспир"
$имя
[1] "Уильям"
$‘год рождения‘
[1] "1564"
$‘год сметри‘
[1] "1616"
$‘семейный статус‘
[1] TRUE
$профессия
[1] "драматург"
$‘имя жены‘
[1] "Хатауэй Анна"
$‘число детей‘
[1] 3
$произведение1
[1] "Ромео и Джульетта"
$произведение2
[1] "Гамлет"
$произведение3
[1] "Отелло"
$Страна
```

```
  Страна
"Англия"
```

Имена компонент списка можно указать сразу при его создании:

```
writer=list(фамилия="Шекспир",имя="Уильям",'год рождения'="1564",
'год смерти'="1616", 'семейное положение'=T, профессия="драматург",
'имя жены'="Хатауэй Анна",'число детей'=3, произведение1="Ромео и Джульетта",
произведение2="Гамлет", произведени3="Отелло")
```

Теперь к компонентам вектора можно обращаться по имени. Для этого есть две возможности. Имя можно указывать после знака **$**, который приписывается к названию списка.

```
> writer$"имя жены"
[1] "Хатауэй Анна"
> writer$'фамилия'
[1] "Шекспир"
> writer$профессия
[1] "драматург"
```

Стоит заметить, что если название какого-либо элемента списка состоит из двух и более слов, то обращение вида

```
writer$имя жены
```

приведёт к появлению сообщения об ошибке.

Второй способ — имя элемента списка задаётся в кавычках и двойных квадратных скобках

```
writer[['имя']]
[1] "Уильям"
writer[["имя"]]
[1] "Уильям"
```

Одинарные квадратные скобки создают новые списки на основе первоначального. Например:

- **lst[1:4]** — новый список, состоящий из первых четырёх элементов исходного списка **lst**;
- **lst[-(2:4)]** — список, полученный исключением второго, третьего и четвёртого элемента из исходного списка;
- **lst[c("name "occupation")]** — список, состоящий из элементов списка **lst** с указанными именами.

Правила использования индексов аналогичны соответствующим правилам для векторов.

Важно понимать различие между правилами использования одинарных и двойных квадратных скобок:

- **lst[[1]]** — первая компонента списка;
- **lst[1]** — новый список, состоящий из первой компоненты списка исходного списка.

Отметим, что в двойных квадратных скобках в качестве индексов не могут использоваться векторы.

Списки являются рекурсивным типом данных, т. е. компоненты списка могут сами быть списками.

```
> Pushkin <- list(name = "Александр Сергеевич Пушкин", year = 1799)
> Gogol <- list(name = "Николай Васильевич Гоголь", year = 1809)
> lst <- list(Pushkin, Gogol)
> lst[[1]]@name
[1] "Александр Сергеевич Пушкин"
> length(lst)
[1] 2
```

Функция **c()** осуществляет конкатенацию двух или более списков:

```
> clst <- c(Pushkin, Gogol)
> clst
@name
[1] "Александр Сергеевич Пушкин"
@year
[1] 1799
@name
[1] "Николай Васильевич Гоголь"
@year
[1] 1799
> length(clst)
[1] 4
> clst@name
[1] "Александр Сергеевич Пушкин"
> clst@year
[1] 1799
> clst[[3]]
[1] "Николай Васильевич Гоголь"
```

```
> clst[[4]]
[1] 1809
```

Функция **unlist(x, recursive, use.names = TRUE)** преобразовывает список **x** в вектор, элементами которого являются компоненты списка. Логический аргумент **recursive** определяет, нужно ли раскладывать компоненты списка на вектора (**logical**, **integer**, **real**, **complex**, **character**) или нет (по умолчанию **recursive = TRUE**). Логический аргумент **use.names** позволяет сохранить имена компонент списка и присвоить их элементам создаваемого вектора (**use.names = TRUE** — значение по умолчанию).

Пример 33. *Для матрицы* **X** *с помощью функции* **determinant()**[7] *создадим список,*

```
> X
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    1    1
[3,]    4    1    2
> Y=determinant(X);Y
$modulus
[1] 1.609438
attr(,"logarithm")
[1] TRUE

$sign
[1] -1

attr(,"class")
[1] "det"
```

который преобразуем в вектор.

```
> z=unlist(Y);z
  modulus      sign
 1.609438 -1.000000
```

При преобразовании списка в вектор учитывается иерархия типов данных (**logical** < **integer** < **real** < **complex** < **character**). Соответственно, в результате преобразования будет получен вектор, чей тип данных определяется наибольшим согласно иерархии типом компоненты списка.

В **R** предусмотрена и обратная функция **relist()**, позволяющая воссоздать список. Аргументы функции **relist()**:

[7] См. раздел 5.3.2

- **flesh** — вектор, который нужно преобразовать в список;
- **skeleton** — список, структура которого определяет создаваемый список.

Пример 34. *Преобразуем полученный в предыдущем примере вектор* **z**

```
> z
  modulus      sign
 1.609438 -1.000000
```

снова в список. Для этого зададим «скелет» — список, задающий структуру искомого результата.

```
> Z=list(c(1),c(1));Z
[[1]]
[1] 1

[[2]]
[1] 1
```

Воссоздаём список.

```
> Y=relist(z,Z);Y
[[1]]
[1] 1.609438

[[2]]
[1] -1
```

5.6 Факторы и таблицы

5.6.1 Факторы — factor()

Фактор — это векторный объект, кодирующий категориальные данные (классы). Факторы создаются с помощью функции **factor()**.

Пример 35. *Провели опрос, в ходе которого 150 человек был задан один и тот же вопрос:«Назовите самый известный фильм Андрея Тарковского». В результате сформирован символьный вектор* **opros**, *состоящий из 150 элементов.*

```
> length(opros)
[1] 150
```

Выделим различные категории — варианты ответов.

```
> factor(opros)
```

В результате будет выведен сам исходный вектор и указаны все различные категории (факторы).

```
 [1] не знаю          Андрей Рублев   Иваново детство Сталкер
 [5] Солярис          Зеркало          не знаю         Андрей Рублев
 [9] Иваново детство Сталкер          Солярис         Зеркало
[13] не знаю          Андрей Рублев   Иваново детство Сталкер
[17] Солярис          Зеркало          не знаю         Андрей Рублев
[21] Иваново детство Сталкер          Солярис         Зеркало
[25] не знаю          Андрей Рублев   Иваново детство Сталкер
[29] Солярис          Зеркало          не знаю         Андрей Рублев
[33] Иваново детство Сталкер          Солярис         Зеркало
[37] не знаю          Андрей Рублев   Иваново детство Сталкер
[41] Солярис          Зеркало          не знаю         Андрей Рублев
[45] Иваново детство Сталкер          Солярис         Зеркало
[49] не знаю          Андрей Рублев   Иваново детство Сталкер
[53] Солярис          Зеркало          не знаю         Андрей Рублев
[57] Иваново детство Сталкер          Солярис         Зеркало
[61] не знаю          Андрей Рублев   Иваново детство Сталкер
[65] Солярис          Зеркало          не знаю         Андрей Рублев
[69] Иваново детство Сталкер          Солярис         Зеркало
[73] не знаю          Андрей Рублев   Иваново детство Сталкер
[77] Солярис          Зеркало          не знаю         Андрей Рублев
[81] Иваново детство Сталкер          Солярис         Зеркало
[85] не знаю          Андрей Рублев   Иваново детство Сталкер
[89] Солярис          Зеркало          не знаю         не знаю
[93] Андрей Рублев   Андрей Рублев   Андрей Рублев   Андрей Рублев
[97] Андрей Рублев   Андрей Рублев   Андрей Рублев   Андрей Рублев
[101] не знаю         не знаю         не знаю         не знаю
[105] не знаю         не знаю         не знаю         не знаю
[109] не знаю         не знаю         не знаю         не знаю
[113] не знаю         не знаю         не знаю         не знаю
[117] не знаю         не знаю         не знаю         не знаю
[121] не знаю         не знаю         не знаю         не знаю
[125] не знаю         не знаю         Сталкер         Сталкер
[129] Сталкер         Сталкер         Сталкер         Сталкер
[133] Сталкер         не знаю         Солярис         не знаю
[137] Сталкер         Солярис         не знаю         Сталкер
```

```
[141] Солярис          не знаю          Сталкер          Солярис
[145] не знаю          Сталкер          Солярис          не знаю
[149] Сталкер          не знаю
Levels: Андрей Рублев Зеркало Иваново детство не знаю Солярис Сталкер
```

При выводе факторы (категории) располагаются в алфавитном порядке (если имеем дело с символьными данными) или в порядке возрастания (числовые данные),если не нужно упорядочивать категории, то следует следует воспользоваться **factor(x,ordered=F)**. Значения **NA**, если были элементами исходного вектора, игнорируются. Чтобы их учитывать, нужно использовать функцию **addNA(x)**, где **x** — исходный вектор.

Функция **ordered(x)** действует аналогично **factor()**, только категории обязательно упорядочиваются и указывается число категорий. Функция **is.ordered(x)** проверяет, упорядочены ли категории или нет.

Пример 36. *Сократим исходный вектор* **opros** *до 30 элементов и воспользуемся* **ordered()**.

```
> opros2=opros[10:40]
> opros2
```

Сократили вектор и вывели его на экран.

```
[1] "Сталкер"         "Солярис"         "Зеркало"         "не знаю"
 [5] "Андрей Рублев"   "Иваново детство" "Сталкер"         "Солярис"
 [9] "Зеркало"         "не знаю"         "Андрей Рублев"   "Иваново детство"
[13] "Сталкер"         "Солярис"         "Зеркало"         "не знаю"
[17] "Андрей Рублев"   "Иваново детство" "Сталкер"         "Солярис"
[21] "Зеркало"         "не знаю"         "Андрей Рублев"   "Иваново детство"
[25] "Сталкер"         "Солярис"         "Зеркало"         "не знаю"
[29] "Андрей Рублев"   "Иваново детство" "Сталкер"
```

Теперь проверим на упорядоченность категорий.

```
> is.ordered(opros2)
[1] FALSE
```

Используем **ordered()**

```
> opros3=ordered(opros2)
> opros3
 [1] Сталкер         Солярис          Зеркало         не знаю
 [5] Андрей Рублев   Иваново детство Сталкер         Солярис
 [9] Зеркало         не знаю          Андрей Рублев   Иваново детство
```

```
[13] Сталкер         Солярис         Зеркало         не знаю
[17] Андрей Рублев   Иваново детство Сталкер         Солярис
[21] Зеркало         не знаю         Андрей Рублев   Иваново детство
[25] Сталкер         Солярис         Зеркало         не знаю
[29] Андрей Рублев   Иваново детство Сталкер
6 Levels: Андрей Рублев < Зеркало < Иваново детство < не знаю < ... < Сталкер
```

и снова проверим на упорядоченность.

```
> is.ordered(opros3)
[1] TRUE
```

Проверка на принадлежность какого-либо объекта к факторам осуществляется при помощи функции **is.factor()**, перевод в факторы — **as.factor()** (**as.ordered()** — переводит в упорядоченные факторы).

```
> is.factor(opros2)
[1] FALSE
> as.factor(opros2)
 [1] Сталкер         Солярис         Зеркало         не знаю
 [5] Андрей Рублев   Иваново детство Сталкер         Солярис
 [9] Зеркало         не знаю         Андрей Рублев   Иваново детство
[13] Сталкер         Солярис         Зеркало         не знаю
[17] Андрей Рублев   Иваново детство Сталкер         Солярис
[21] Зеркало         не знаю         Андрей Рублев   Иваново детство
[25] Сталкер         Солярис         Зеркало         не знаю
[29] Андрей Рублев   Иваново детство Сталкер
Levels: Андрей Рублев Зеркало Иваново детство не знаю Солярис Сталкер
```

Если нужно перевести факторы в векторы, то используем **as.vector()**. При этом фактор преобразовывается в символьный вектор.

```
> x=rep(c(1,5,7,3,2),6);x
 [1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
> y=factor(x);y
 [1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
Levels: 1 2 3 5 7
> is.vector(y)
[1] FALSE
> z=as.vector(y);z
 [1] "1" "5" "7" "3" "2" "1" "5" "7" "3" "2" "1" "5" "7" "3" "2" "1"
[17] "5" "7" "3" "2" "1" "5" "7" "3" "2" "1" "5" "7" "3" "2"
> is.vector(z)
[1] TRUE
```

Функция gl()

Функция **gl()** позволяет создавать категориальные данные с заданным числом уровней (категорий, факторов).

Полная запись

```
gl(n, k, length, labels, ordered = FALSE)
```

Разберём аргументы:

- **n** — целочисленный аргумент, задаёт количество различных категорий.
- **k** — целочисленный аргумент, определяет сколько раз эти категории встречаются (число повторений набора категорий).

```
> gl(5,3)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
Levels: 1 2 3 4 5
```

- **length** — целочисленный аргумент — длина создаваемого категориального вектора (по умолчанию **length** = **n**·**k**). Если заданная длина меньше произведения **n**·**k**, то будет создан усечённый вектор, если заданная длина больше, чем **n**·**k**, то категориальный вектор длины **n**·**k** повторяется до тех пор, пока его длина не совпадёт с заданной.

```
> gl(5,3,15)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
Levels: 1 2 3 4 5
> gl(5,3,12)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
Levels: 1 2 3 4 5
> gl(5,3,25)
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 1 1 1 2 2 2 3 3 3 4
Levels: 1 2 3 4 5
```

- **labels** — символьный аргумент, вектор длины **n**, задающий названия категорий (по умолчанию задаются числовые категории **labels** = **1:n**).

```
gl(5,3,15,labels=LETTERS[1:5])
 [1] A A A B B B C C C D D D E E E
Levels: A B C D E
gl(5,3,15,labels=c('категория1','категория2','категория3','категория4',
'категория5'))
```

```
 [1] категория1 категория1 категория1 категория2 категория2 категория2
 [7] категория3 категория3 категория3 категория4 категория4 категория4
[13] категория5 категория5 категория5
Levels: категория1 категория2 категория3 категория4 категория5
```

- **ordered** — логический аргумент, определяет, нужно ли упорядочивать категории (по умолчанию значение **FALSE**).

```
gl(5,3,15,labels=c('категория1','категория2','категория3','категория4',
'категория5'), ordered=T)
 [1] категория1 категория1 категория1 категория2 категория2 категория2
 [7] категория3 категория3 категория3 категория4 категория4 категория4
[13] категория5 категория5 категория5
Levels: категория1 < категория2 < категория3 < категория4 < категория5
```

5.6.2 Таблицы — table()

Другая функция, позволяющая работать с категориальными данными — **table(имя объекта)**. Особенность этой функции заключается в том, что она не только выделяет различные категории данных, но и систематизирует их, указывая сколько элементов находится в каждой категории. В результате работы **table()** получим таблицу, первая строка которой — это различные категории, а вторая — сколько раз они встречаются.

Пример 37. *Продолжим пример 36 и применим к вектору* **opros2** *функцию* **table()**.

```
> table(opros2)
opros2
  Андрей Рублев   Зеркало Иваново детство  не знаю  Солярис
              5        5           5           5        5
       Сталкер
              6
```

Категории упорядочены. Если в первоначальном массиве данных были **NA** и (или) **NaN**, то при формировании таблицы они будут отброшены. Чтобы их учесть надо прописать дополнительный аргумент **table(x,exclude=NULL)**.

Пример 38. *Добавим в вектор* **opros2** *элементы* **NA** *и* **NaN**.

```
> opros2=c(opros2,rep(c(NA,NaN),c(6,9)))
> opros2
 [1] "Сталкер"         "Солярис"         "Зеркало"         "не знаю"
```

```
 [5] "Андрей Рублев"   "Иваново детство" "Сталкер"         "Солярис"
 [9] "Зеркало"         "не знаю"         "Андрей Рублев"   "Иваново детство"
[13] "Сталкер"         "Солярис"         "Зеркало"         "не знаю"
[17] "Андрей Рублев"   "Иваново детство" "Сталкер"         "Солярис"
[21] "Зеркало"         "не знаю"         "Андрей Рублев"   "Иваново детство"
[25] "Сталкер"         "Солярис"         "Зеркало"         "не знаю"
[29] "Андрей Рублев"   "Иваново детство" "Сталкер"         NA
[33] NA                NA                NA                NA
[37] NA                "NaN"             "NaN"             "NaN"
[41] "NaN"             "NaN"             "NaN"             "NaN"
[45] "NaN"             "NaN"
```

Построим таблицу, отбрасывая **NA** *и* **NaN**

```
> table(opros2)
opros2
  Андрей Рублев   Зеркало  Иваново детство   не знаю  Солярис
              5        5            5            5         5
        Сталкер
              6
```

и учитывая их

```
> table(opros2,exclude=NULL)
opros2
            NaN   Андрей Рублев  Зеркало Иваново детство  не знаю
              9        5            5           5            5
        Солярис    Сталкер      <NA>
              5        6          6
```

Принадлежность объекта к классу **table()** проверяется при помощи **is.table()**, а перевод в этот класс — **as.table()**.

Пример 39. *Проверим, принадлежат ли вектор и фактор к классу* **table()** *и что будет, если попытаться перевести вектор и фактор в класс* **table()**.

```
> x=rep(c(1,5,7,3,2),6);x
 [1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
> y=factor(x);y
 [1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
Levels: 1 2 3 5 7
```

Создали вектор и на основе этого вектора фактор. Проверим их на принадлежность к классу **table()**.

```
> is.table(x)
[1] FALSE
> is.table(y)
[1] FALSE
```

Попробуем рассматривать вектор **x** *как* **table()**.

```
> z=as.table(x);z
 A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W  X  Y  Z
 1  5  7  3  2  1  5  7  3  2  1  5  7  3  2  1  5  7  3  2  1  5  7  3  2  1
A1 B1 C1 D1
 5  7  3  2
> is.table(z)
[1] TRUE
```

Аналогично и с фактором **y**.

```
> z=as.table(y);z
Ошибка в as.table.default(y) : не могу преобразовать в таблицу
> is.table(z)
[1] FALSE
```

Как показано в примере вектора, факторы и таблицы (**table()**) —это разные структуры. Вектора можно рассматривать как таблицы (при этом каждый элемент вектора — это сколько раз встречается некая категория, имена категорий образуются при помощи латинского алфавита), а факторы — нельзя.

Таблицу можно преобразовать в вектор:

```
> as.vector(z)
 [1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
```

Заметим, что в этом случае (в отличие от преобразования факторов в векторы) числа не становятся символами.

5.7 Таблицы данных

Таблицы (фреймы данных) (data frames) — один из самых важных типов данных в **R**, позволяющий объединять данные различных типов вместе.

Если слегка упростить, то таблица данных — эта двумерная таблица, в которой (в отличие от числовых матриц), разные столбцы могут содержать данные разных типов (но все данные в одном столбце имеют один тип). Например, такая таблица может содержать результаты эксперимента. Создать фрейм данных можно с помощью функции **data.frame()**:

```
> frm <- data.frame(data1, data2, ...)
```

Здесь многоточие означает, что список данных может содержать произвольное число элементов. В качестве данных **(data1, data2, . . .)** могут выступать векторы (числовые, символьные или логические), факторы, матрицы (число- вые, символьные или логические), списки или другие таблицы данных. При этом все векторы должны иметь одинаковую длину, а матрицы и таблицы — одинаковое (такое же) число строк. Могут также встречаться векторы, длина которых меньше, но в этом случае эта длина должна являться делителем максимальной встречающейся длины. То же требование предъявляется к компонентам списков. Функция **data.frame** просто собирает все данные вместе. Символьные векторы конвертируются в факторы. Остальные данные собираются во фрейм такими, какие они есть.

Пример 40. *Создадим таблицу данных по четырём студентам — год рождения и год поступления в ВУЗ.*

```
Y = matrix(c(1988,1987,1989,1989,2005,2005,
2006, 2005), nrow = 4)
rownames(Y) = c("Иванов", "Ульянов", "Краснова", "Устюгов")
colnames(Y) = c("год рождения", "год поступления")
```

При обращении получим следующую таблицу:

```
> Y
         год рождения год поступления
Иванов           1988            2005
Ульянов          1987            2005
Краснова         1989            2006
Устюгов          1989            2005
```

Добавим новые столбцы — есть ли у студента задолженности и на каком курсе.

```
n = c(FALSE, TRUE, FALSE, FALSE)
y=c(2,1,3,2)
Y1=data.frame(Y,n,y)
Y1
         год рождения год поступления     n y
Иванов           1985            2002 FALSE 2
Ульянов          1987            2004  TRUE 1
Краснова         1986            2003 FALSE 3
Устюгов          1984            2001 FALSE 2
```

Переименуем последние столбцы и снова выведем таблицу

```
colnames(Y1)[3] = "задолженность"
colnames(Y1)[4] = "курс"
Y1
         год рождения год поступления задолженность курс
Иванов           1985            2002         FALSE    2
Ульянов          1987            2004          TRUE    1
Краснова         1986            2003         FALSE    3
Устюгов          1984            2001         FALSE    2
```

Если среди аргументов функции **data.frame()** есть список, то каждая его компонента превращается в столбец фрейма и имена столбцов формируются самостоятельно, что видно из примера.

```
> lst = list(1:4,FALSE, c("A", "B"))
> data.frame(lst)
  X1.4 FALSE. c..A....B..
1    1  FALSE           A
2    2  FALSE           B
3    3  FALSE           A
4    4  FALSE           B
```

Доступ к элементам таблицы осуществляется двумя способами: в «матричном» или «списковом» стилях. В первом случае указываются номера или имена строки и столбца. Во втором таблица рассматривается как список, элементами которого являются столбцы таблицы, и чтобы обратиться к его элементу нужно указать имя или номер этого элемента (т. е. имя или номер столбца), а затем имя или номер строки.

Пример 41. *Вернёмся к примеру 40.*

```
Y1["Иванов", "курс"]
[1] 2

Y1["Устюгов",]
        год.рождения год.поступления задолженность курс
Устюгов         1989            2005         FALSE    2

Y1[4, 3]
[1] FALSE

Y1[,c(2,4)]
```

```
         год.поступления курс
Иванов              2005    2
Ульянов             2005    1
Краснова            2006    3
Устюгов             2005    2

Y1$курс
[1] 2 1 3 2

Y1[["задолженность"]][3]
[1] FALSE

Y1[["задолженность"]][2]
[1] TRUE
```

Редактировать таблицы данных можно и в редакторе (см. раздел 5.3.1 этой главы).

Глава 6

Ввод и вывод данных в R

В пакете **R** реализовано большое количество функций, позволяющих считывать данные из файлов, а также записывать данные. В этой главе будут рассмотрены только основные из них (с остальными функциями ввода/вывода желающие могут самостоятельно ознакомиться).

6.1 Ввод данных в R

В этой части рассмотрим следующие функции, позволяющие вводить данные в **R**.

- **scan( )**;
- **read.table( )** и **read.csv( )**.

6.1.1 Функция scan( )

Данные считываются в вектор или в список с консоли или из файла.
Полный вид:

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",
skip = 0, nlines = 0, na.strings = "NA",
flush = FALSE, fill = FALSE, strip.white = FALSE,
quiet = FALSE, blank.lines.skip = TRUE, multi.line = TRUE,
comment.char = "", allowEscapes = FALSE,
encoding = "unknown")
```

Аргументы:

- **file** — имя файла, откуда считываются данные; если **file**=, то производится ввод с клавиатуры; если задано только имя файла, то он (файл) ищется в текущей директории; иначе задаётся полный путь к файлу; может быть и URL.

- **what** — задаётся тип считываемых данных: **logical**, **integer**, **numeric**, **complex**, **character**, **list**; если считывается список, то строки в файле воспринимаются как поля списка указанных выше типов;

- **nmax** — целое положительное число — максимальное число данных для чтения или максимальное число записей в списке; при пропуске этого аргумента или при неправильном его задании файл считывается до конца;

- **n** — целое положительное число — максимальное число данных для чтения; неправильные значения или не типа **integer** игнорируются

- **sep** — разделитель полей; по умолчанию - пробел;

- **quote** — вид кавычек (двойные или одинарные);

- **dec** — десятичный разделитель (точка или запятая);

- **skip** — целое положительное число — число строк файла, которые следует пропустить перед чтением;

- **nlines** — целое положительное число — максимальное число строк для считывания;

- **na.strings** — символьный вектор — его элементы интерпретируются как пропущенные значения **NA**; пустые поля по умолчанию считываются как **NA**;

- **flush** — логический аргумент — значение **TRUE** позволяет добавлять комментарии к считываемым данным после последнего считанного поля (но не более одного);

- **fill** — логический аргумент — значение **TRUE** добавляются пустые поля к строкам, в которых количество полей данных меньше определённого параметром **what**;

- **strip.white** — логический вектор; используется только если задан параметр **sep**, удаляет пустое пространство (пробел) перед символьными переменными и после них;

- **quiet** — логический аргумент; при значении **FALSE** функция выведет сообщение о том, сколько элементов было прочитано;
- **blank.lines.skip** — логический аргумент; при значении **TRUE** пустые строки игнорируются (не считываются) (заметим, что параметры **skip** и **nlines** всё равно будут учитывать все пустые строки);
- **multi.line** — логический аргумент; используется если аргумент **what** принимает значение **list**; при значении **FALSE** все записи будут считаны в одну строку; если же и **fill=T**, то чтение при достижении конца строки будет прекращено;
- **comment.char** — символьный аргумент, определяет знак комментария;
- **allowEscapes** — логический аргумент — нужно ли следующие последовательности символов $\backslash n$, $\backslash a$, $\backslash b$, $\backslash f$, $\backslash r$, $\backslash t$, $\backslash v$ при чтении рассматривать как команды (**TRUE**) или просто как символы (**FALSE**);
- **encoding** — символьный аргумент, задаёт кодировку считываемого файла.

6.1.2 Функции read.table() и read.csv()

Если исходные данные представлены в виде таблицы и итоговый результат должен быть таблицей (фреймом данных), то удобнее воспользоваться функцией **read.table( )** либо **read.csv()**.

Полная запись функции:

```
read.table(file, header = FALSE, sep = "", quote = "\"'",
dec = ".", row.names, col.names,
as.is = !stringsAsFactors,
na.strings = "NA", colClasses = NA, nrows = -1,
skip = 0, check.names = TRUE, fill = !blank.lines.skip,
strip.white = FALSE, blank.lines.skip = TRUE,
comment.char = "#",
allowEscapes = FALSE, flush = FALSE,
stringsAsFactors = default.stringsAsFactors(),
fileEncoding = "", encoding = "unknown")
```

Упрощённые варианты функции:

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",
fill = TRUE, comment.char="", ...)
```

```
read.csv2(file, header = TRUE, sep = ";", quote="\"", dec=",",
fill = TRUE, comment.char="", ...)
read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",
fill = TRUE, comment.char="", ...)
read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec=",",
fill = TRUE, comment.char="", ...)
```

которые особенно полезны при считывании данных из файлов **Excel**. Для этого исходный файл сохраняется в формате **.csv** и затем прочитывается при помощи одной из четырёх приведённых выше функций (лучше воспользоваться **read.csv2()**).

Аргументы:

- **file** — обязательный аргумент, имя файла;
- **header** — логический параметр; при значении TRUE считываются имена переменных из файла;
- **sep** — разделитель полей; по умолчанию - пробел;
- **quote** — вид кавычек (двойные или одинарные);
- **dec** — десятичный разделитель в числах (точка или запятая);
- **row.names** — вектор имён строк; представляет собой либо вектор с именами строк итоговой таблицы; либо число — номер столбца исходной таблицы с названиями строк; либо имя столбца считываемой таблицы, где приведены названия строк; если этот параметр не задан, то строки в итоговой таблице будут пронумерованы;
- **col.names** — вектор имён столбцов в итоговой таблице; по умолчанию — **«V<номер столбца>»**;
- **as.is** — нужно ли символьные переменные, не преобразованные в числовые или логические, переводить в факторы. **as.is** — либо логический, либо числовой вектор, определяющий столбцы, неконвертируемые в факторы.
- **colClasses** — символьный вектор; определяет классы данных в столбцах (символьные, логические, числовые, даты). Возможные значения: **NA** — автоматическая конвертация типов данных, **NULL** — столбец пропускается (данные не преобразовываются), **тип данных** в который будут переведены элементы столбца, **factor**;

- **na.strings** — символьный вектор, элементы которого при чтении исходной таблицы в файле будут интерпретироваться как **NA**;
- **nrows** — целочисленный аргумент; определяет максимальное число считываемых строк;
- **skip** — положительный целочисленный аргумент; определяет число строк, пропускаемых перед чтением;
- **check.names** — логический аргумент; при значении **TRUE** имена переменных будут проверены на синтаксическую правильность и отсутствие дублирования;
- **fill** — логический аргумент; при значении **TRUE** строки разной длины будут приведены к единой (максимальной) добавлением пустых полей;
- **strip.white** — логический аргумент; используется только если определён разделитель **sep**, позволяет убирать пробелы перед и после символьных переменных;

Примечания:

- Функция **read.table()** является основной для считывания данных из таблиц.
- Поле таблицы считается пустым, если в нём ничего нет (до знака комментария или до символа окончания строки).
- Если параметр **row.names** не определён (т.е. не заданы имена строк результирующей таблицы), а длина заголовка на единицу меньше числа столбцов, то первый столбец будет рассматриваться как столбец с названиями строк.
- Число столбцов в считываемой таблице определяется автоматически после прочтения первых пяти строк.
- Всё, что находится в исходной таблице после знаков комментария, не считывается.
- Задание параметра **nrows** (пусть даже с очень избыточными значениями) позволяет уменьшить затраты памяти.
- Для чтения больших матриц предпочтительнее использовать функцию **scan()**.

6.2 Вывод данных в R

Здесь мы рассмотрим следующие функции:

- **write()**;
- **cat()**;
- **write.table()**, **write.csv()** и **write.csv()**.

6.2.1 Функция write()

Функция **write()** предназначена для записи данных (в основном матриц) в **R**. Её вид:

```
write(x, file = "data",
ncolumns = if(is.character(x)) 1 else 5,
append = FALSE, sep = " ")
```

Аргументы:

- **x** — данные, которые нужно записать;
- **file** — имя файла, куда будет записана информация;
- **ncolumns** — число столбцов, в которых будет записана информация;
- **append** — логический аргумент — если значение **TRUE**, то данные допишутся в исходный файл, если же **FALSE**, то файл будет переписан заново;
- **sep** — разделитель столбцов ($\backslash t$ — табуляция)

Если в качестве аргумента функции задать только данные — **write(x)**, то они будут выведены на экран.

6.2.2 Функция cat()

Функция **cat()** имеет вид

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
append = FALSE)
```

Её аргументы:

- **...** — объекты **R**, которые будут записаны в файл.

- **file** — имя файла, куда будет записана информация; если этот аргумент отсутствует, то данные будут выведены на экран;
- **sep** — разделитель элементов записываемого объекта;
- **fill** — логический или положительный целочисленный аргумент — контролирует создание новых строк в записываемом файле. Если **fill** — числовой, то задаётся длина строки (количество символов в строке). Если **fill** — логический и его значение **FALSE**, то новые строки создаются только при наличии в записываемых данных символа "$\backslash n$"; если значение **TRUE**, то задаётся дополнительный аргумент **width**, определяющий длину создаваемой в файле строки;
- **labels** — символьный вектор, задающий названия строк. Игнорируется, если если аргумент **fill**=**FALSE**.
- **append** — логический аргумент — используется, если задано имя файла. Если значение аргумента **TRUE**, то новые данные добавляются к исходному файлу, если **FALSE**, то записываются вместо старых.

Функция **cat()** преобразовывает исходные данные в символьные, объединяет их в единый символьный вектор и записывает в заданный файл.

6.2.3 Функции write.table(), write.csv() и write.csv()

Функции **write.table()** записывает таблицу данных (или матрицу) в заданный файл. Если записываемый объект не является таблицей (фреймом данных), то он автоматически будет конвертирован.

Вид функции:

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",
eol = "\n", na = "NA", dec = ".", row.names = TRUE,
col.names = TRUE, qmethod = c("escape", "double"))
```

Разновидности:

```
write.csv(...)
write.csv2(...)
```

Аргументы:

- **x** — записываемый объект. Предпочтительнее, чтобы это была матрица или таблица (фрэйм) данных.

- **file** — имя файла, в который будут записаны данные.
- **append** — логический аргумент — используется, если только задано имя файла. Если значение аргумента **TRUE**, то новые данные добавляются к исходному файлу, если **FALSE** — записываются вместо старых.
- **quote** — логический или числовой аргумент. Если **quote** является логическим аргументом и его значение **TRUE**, то все символьные переменные и факторы будут записаны в файл в двойных кавычках. Если значение аргумента **FALSE** — символьные переменные и факторы записываются без кавычек. Если **quote** — числовой вектор, то его элементы задают номера столбцов, в которых данные должны быть записаны в кавычках. Названия столбцов и строк по умолчанию будут записаны в кавычках.
- **sep** — аргумент, определяющий разделитель полей; значения в каждой строке исходных данных разделяются этим символом.
- **eol** — символ окончания строки; для Windows и Unix - "$\backslash n$", "$\backslash r$"; для MacOS - "$\backslash r$".
- **na** — символьный аргумент для обозначения отсутствующих элементов в исходных данных.
- **dec** — десятичный разделитель в числах (точка или запятая).
- **row.names** — аргумент, задающий названия строк в файле. Аргумент либо логический (если значение **TRUE**, то используются имена строк, указанные в исходных данных), либо представляет собой символьный вектор, непосредственно задающий имена строк.
- **col.names** — аналогичный аргумент, задающий имена столбцов.
- **qmethod** — символьный вектор, определяющий как поступать с вложенными друг в друга кавычками — ""a"". Два значения — **escape** (внешние кавычки убираются - по умолчанию) и **double** (все кавычки остаются).

Для функций **write.csv( )** и **write.csv2( )** аргументы **col.names**, **sep**, **dec** и **qmethod** не могут быть изменены.

Примечания:

- Если в исходной записываемой таблице данных нет столбцов, то при записи имена строк будут присвоены только в том случае, когда **row.names=TRUE**, и наоборот.

- Действительные и комплексные числа записываются с максимально возможной точностью.

- Если в исходной записываемой таблице данных имелись столбцы с матричной структурой, то при записи каждый из столбцов этой матрицы будет представлен в виде отдельного столбца; в связи с этим аргументы **col.names** и **quote** должны соответствовать числу столбцов результирующей таблицы, а не исходной.

- Каждый столбец в таблице данных, являющийся списком или датой, будет переконвертирован в символы.

- Только символьные столбцы (или столбцы, переконвертированные в символы) будут при записи заключены в кавычки (если это указано в аргументе **quote**)

- Аргумент **dec** применяется только к тем столбцам, которые не были переконвертированы в символьные (т.е. только к числовым данным).

*.CSV файлы

.

Функции **write.csv( )** и **write.csv2( )** используются для записи файлов в формате ***.csv**.

Если параметр **row.names=TRUE**, то значения параметров **qmethod** и **col.names** устанавливается **NA**; если же **row.names=FALSE**, то **qmethod= col.names=TRUE**.

Для функции **write.csv( )** параметр **dec** принимает значение **"."**, а параметр **sep** — **","**.

Для функции **write.csv2( )** десятичный разделитель **dec=","**, разделитель полей **sep=";"**.

Замечание.

- Попытки изменить значения по умолчанию параметров **col.names**, **sep**, **dec** и **qmethod** игнорируются и будет выдано предупреждение.

- Если таблица данных содержит большое число столбцов (более ста), функция **write.table( )** работает медленно, так как каждый столбец может быть переменной своего класса и поэтому обрабатывается отдельно.

Пример 42. *Создадим таблицу данных*

```
> x<-data.frame(a=letters[1:3], b=1:3);x
  a b
1 a 1
2 b 2
3 c 3
> rownames(x)=LETTERS[1:3]
> x
  a b
A a 1
B b 2
C c 3
```

и затем при помощи различных функций запишем эту таблицу и прочитаем созданный файл.

Запись

```
write.table(x, file="example.csv", sep=",", row.names=T, col.names=NA,
qmethod="double")
```

и чтение

```
> read.table("example.csv", header=T, sep=",", row.names=1)
  a b
A a 1
B b 2
C c 3
```

или

```
> write.csv(x, file="example.csv",row.names=T)

> read.csv("example.csv", row.names=1)
  a b
A a 1
B b 2
C c 3
```

Глава 7

Базовая графика в R

В базовой конфигурации **R** один из пакетов (графических пакетов несколько, но рассмотрим только один) отвечает за графику — пакет **graphics**. Выделим три группы входящих в этот пакет функций:

- функции высокого уровня (высокоуровневые);
- функции низкого уровня (низкоуровневые);
- интерактивные функции;

Функции первого типа (высокоуровневые) приводят к созданию графического окна, в котором строится заданное изображение.

Низкоуровневые функции позволяют изменить (дополнить) уже построенное ранее изображение. Сами они не могут создать графическое окно. Как правило, большинство низкоуровневых функций дублируют аргументы высокоуровневых функций.

Наконец, интерактивные функции позволяют по построенному изображению (графику) получить некоторую информацию.

В следующих разделах этой главы более подробно будут рассмотрены графические функции.

7.1 Функции высокого уровня

Выделим основные функции высокого уровня:

- **par()** — создание графического окна с заданными параметрами;
- **barplot()** — построение мозаичной диаграммы (статистика);

- **boxplot()** — построение **boxplot** — особой диаграммы для статистического анализа данных относительно предполагаемого распределения (статистика);
- **boxplot.matrix()** — аналогично **boxplot**;
- **bxp()** — аналогично **boxplot**;
- **contour()** — построение графика с контурными линиями (линиями уровня);
- **curve()** — построение линии (кривой);
- **dotchart()** — точечные диаграммы (статистика);
- **frame()** — задание нового графического окна;
- **plot.new()** — задание нового графического окна;
- **hist()** — построение гистограммы (статистика);
- **image()** — построение цветной прямоугольной сетки согласно заданным цветам;
- **matplot()** — изображение элементов столбцов одной матрицы относительно элементов соответствующих столбцов другой матрицы;
- **mosaicplot()** — построение мозаичной диаграммы (статистика);
- **persp()** — построение трёхмерных графиков;
- **pie()** — построение круговой диаграммы;
- **plot()** — основная функция построения двумерных графиков;
- **plot.data.frame()** — графический анализ таблиц данных;
- **plot.default()** — построение диаграммы рассеивания ;
- **plot.factor()** — построение диаграммы рассеивания для факторов;
- **plot.formula()** — построение диаграммы рассеивания с помощью структуры **formula**;
- **plot.histogram()** — построение гистограммы (статистика);

- **plot.table()** — графический анализ таблицы данных (построение мозаичных диаграмм);
- **screen** — управление графическим окном;
- **stem()** — построение древесной диаграммы (статистика);
- **stripchart()** — построение диаграммы рассеивания (аналог **boxplot()**).

Рассматривать же будем только некоторые из вышеприведённых функций. Функции, относящиеся к статистике, будут рассмотрены отдельно в пособии, посвящённом теории вероятностей и математической статистики в **R**.

7.1.1 Функция par()

Начнём подробное рассмотрение высокоуровневых графических функций с **par()**, определяющей **все** параметры создаваемого графического окна. Вид функции:

```
par(..., no.readonly = FALSE)
```

Здесь ... — аргументы вида **имя=значения** или список именованных аргументов, определяющих вид графического окна. Для того, чтобы узнать все аргументы функции **par()**, необходимо обратиться к ней в **R**.

```
> par()
```

Результат — создание графического окна и вывод списка (из 70 компонент) всех его параметров (изменяемых и неизменяемых), т.е. аргументов функции и их значений, соответствующих данному графическому окну.

```
$xlog         $cin          $family       $lheight      $mkh
$ylog         $col          $fg           $ljoin        $new
$adj          $col.axis     $fig          $lmitre       $oma
$ann          $col.lab      $fin          $lty          $omd
$ask          $col.main     $font         $lwd          $omi
$bg           $col.sub      $font.axis    $mai          $pch
$bty          $cra          $font.lab     $mar          $pin
$cex          $crt          $font.main    $mex          $plt
$cex.axis     $csi          $font.sub     $mfcol        $ps
$cex.lab      $cxy          $lab          $mfg          $pty
$cex.main     $din          $las          $mfrow        $smo
$cex.sub      $err          $lend         $mgp          $srt
```

`$tck`	`$usr`	`$xaxs`	`$xpd`	`$yaxs`
`$tcl`	`$xaxp`	`$xaxt`	`$yaxp`	`$yaxt`

Второй аргумент функции **par(..., no.readonly = FALSE)** — **no.readonly** — логический аргумент, если его значение **TRUE** и не заданы другие аргументы, то будут выведены только те параметры функции **par()**, которые установлены для текущего графического окна и могут быть установлены (изменены) в дальнейшем. Таких параметров — 65.

Таким образом, только 5 аргументов функции **par()** не могут быть заданы и (или) изменены пользователем. Это следующие аргументы:

- **"cin"** — размер (ширина, высота) символов в дюймах,
- **"cra"** — размер (ширина, высота) символов в растрах (пикселях) (некоторые графические устройства устанавливают произвольный размер пикселя, обычно около 1/72 дюйма),
- **"csi"** — высота символа в дюймах,
- **"cxy"** — размер символа в пользовательских единицах (единицы размера,определённые пользователем — **par("cxy") = par("cin") / par("pin")**) и, наконец,
- **"din"** — размеры графического устройства в дюймах.

Из оставшихся 65 аргументов (параметров графического окна) следующие (21) могут быть заданы только в функции **par()**:

- **"ask"** — логический аргумент (по умолчанию значение **FALSE**); при значении **TRUE** в интерактивном режиме запрашивает разрешение на построение нового изображения;
- **"fig"** — числовой вектор **c(x1, x2, y1, y2)** — задаёт координаты области выводимого рисунка на графическом объекте (по умолчанию строится новый объект, чтобы добавить создаваемый к уже существующему необходимо задать параметр **new=T**); **"fin"=c(width, height)** — параметры области, на которой строится изображение, в дюймах (по умолчанию строится новый объект с заданным параметром);
- **"lheight"** — числовой аргумент, задающий множитель высоты строки выводимого текста (высота строки = высота символа (по умолчанию) × множитель высоты символа × множитель высоты строки), используется в **text()**;

- **"mai"=c(вниз, влево, вверх, вправо)** — числовой вектор — параметры (в дюймах) отступа от области с изображением (по умолчанию **c(1.02, 0.82, 0.82, 0.42)**); **"mar"=c(вниз, влево, вверх, вправо)** — числовой вектор — параметры отступа (в линиях) от области с изображением (по умолчанию **c(5, 4, 4, 2) + 0.1**); **"mex"** — символьная переменная, описывающая параметры отступа от области с рисунком;

- **"mfcol"** и **"mfrow"** — числовые вектора вида **c(nr, nc)**, графическое окно разбивается на $nr \times nc$ подокон, в которых будут выведены изображения, порядок заполнения — по столбцам и строкам, соответственно (при этом уменьшаются размеры выводимого в изображениях текста); **"mfg"=c(i, j)** — числовой вектор, аргументы которого определяют выводимый следующим рисунок (из созданного с помощью **"mfcol"** или **"mfrow"** массива рисунков);

- **"new"** — логический параметр; при значении **TRUE** следующая высокоуровневая команда не очищает графическое окно перед построением графика; по умолчанию имеет значение **FALSE**;

- **"oma"** — числовой вектор **c(вниз, влево, вверх, вправо)** — задаёт в линиях размеры внешнего поля графического окна, **"omd"** — числовой вектор **c(x1, x2, y1, y2)** — задаёт размеры внешнего поля графического окна в долях всего графического окна (так называемые NDC координаты — нормированные координаты устройства (normalized device coordinates)); **"omi"** — числовой вектор **c(вниз, влево, вверх, вправо)** — задаёт в дюймах размеры внешнего поля графического окна;

- **"pin"** — текущие размеры (ширина и высота) изображения в дюймах, **"plt"** — вектор вида **c(x1, x2, y1, y2)** — координаты изображения в долях графического окна, **"ps"** — число —размер текста в точках (одна точка примерно равна 1/72 дюйма), **"pty"** — символьный параметр — определяет тип области, выделяемой под изображение, **"s"** — квадратная область, **"m"** — максимально возможная область;

- **"usr"** — числовой вектор вида **c(x1, x2, y1, y2)** — задаёт предельные пользовательские координаты области изображения (начальное и конечное значения по осям);

- **"xlog"** и **"ylog"** — логические аргументы — нужно ли осевую шкалу переводить в логарифмическую форму.

Остальные 44 аргумента могут задаваться как в **par()**, так и в других графических функциях (высокоуровневых и низкоуровневых), поэтому их будем рассматривать в следующих разделах этой главы.

Замечание. Во избежание возможных недоразумений, связанных с созданием графических окон, рекомендуется присваивать параметры нового графического окна некоторой переменной, чтобы в дальнейшем можно было вернуться к исходным параметрам (по умолчанию).

```
op = par(mfrow = c(3,3), mar = .1+ c(2,2,3,1))
.
.
.
par(op)
```

7.1.2 Функция plot()

Функция

```
plot(x, y, ...)
```

позволяет графически отображать зависимость **y** от **x**.

Аргументы функции:

- **x** — координаты точки на графике.
- **y** — координаты по оси **y**, если заданы соответствующие значения по оси **x**.
- ... — дополнительные аргументы. К ним относятся:
  - **type** — символьный аргумент, определяющий тип построения графика. Возможные значения: **"p"** — точки (круги), **"l"** — линии, **"b"** — линии и точки, **"c"** — строятся только линии из **"b"**, **"o"** — точки и линии пересекаются, **"h"** — гистограммо-подобные вертикальные линии, **"s"** —ступенчатая линия, **"S"** — другая ступенчатая линия (см. **Замечания** ниже), **"n"** — ничего не строится (но зато определяются области о осям). Результат различного задания параметра **type** приведён на рис.7.1.
  - **log** — символьный аргумент — задаётся название оси (**"x"**, **"y"** или **"xy"**), координаты по которой переводятся в логарифмическую шкалу.
  - **lty** — стиль линии. Возможные значения: **"solid"** (**1**) — сплошная линия; **"blank"** (**0**) — отсутствует линия; **"dashed"** (**2**) — штриховая линия; **"dotdash"** (**4**) — штрих-пунктир; **"dotted"** (**3**) — пунктирная линия; **"longdash"** (**5**) — длинный штрих; **"twodash"** (**6**) —

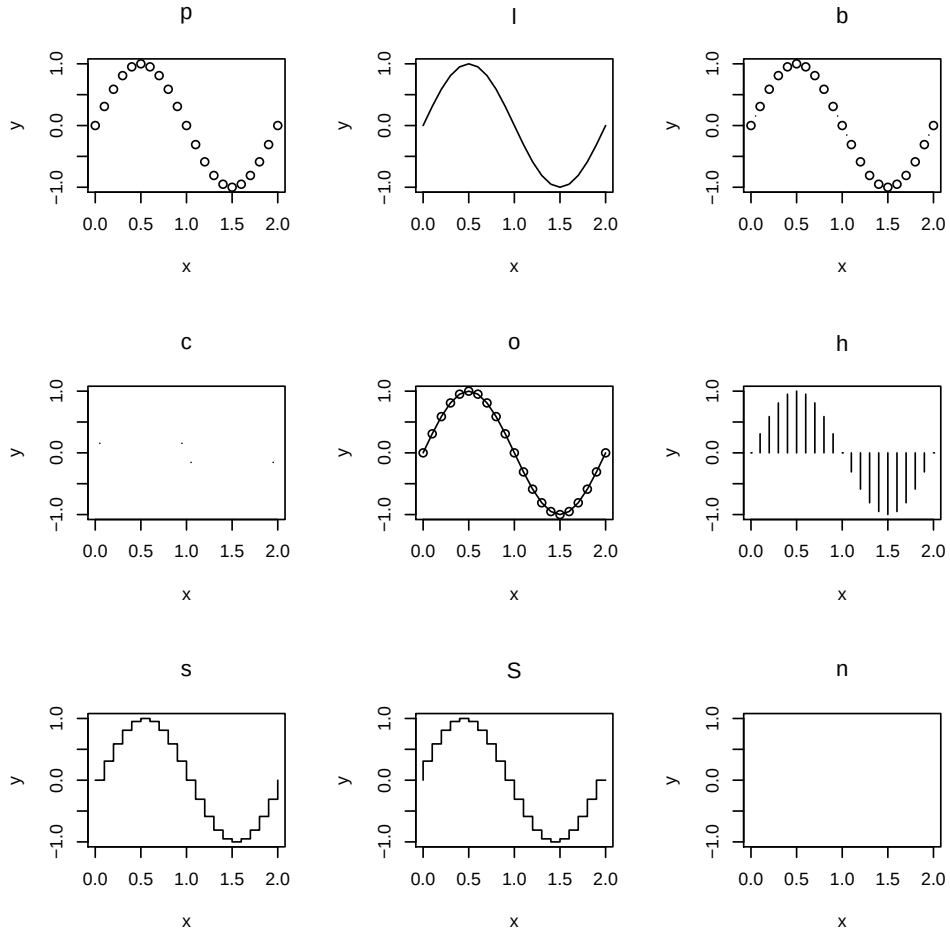

Рис. 7.1: $sin(x)$ при различных значениях аргумента **type**

двойной штрих (следует отметить, что должен быть задан параметр **type**=”**l**”). Результаты использования этого параметра представлены на рис. 7.2

– **xlab**, **ylab**, **main** и **sub** — символьные переменные — названия осей, основного и дополнительного заголовков графика (более подробно — см. разделы 7.2.2 и 7.2.3 этой главы).

– **col** — символьный или числовой аргумент — цвет графика. Подробнее — см. раздел 7.1.6 данной главы.

– **bg** — символьная переменная, отвечающая за цвет фона графического окна. По умолчанию белый — **bg**=”**white**”.

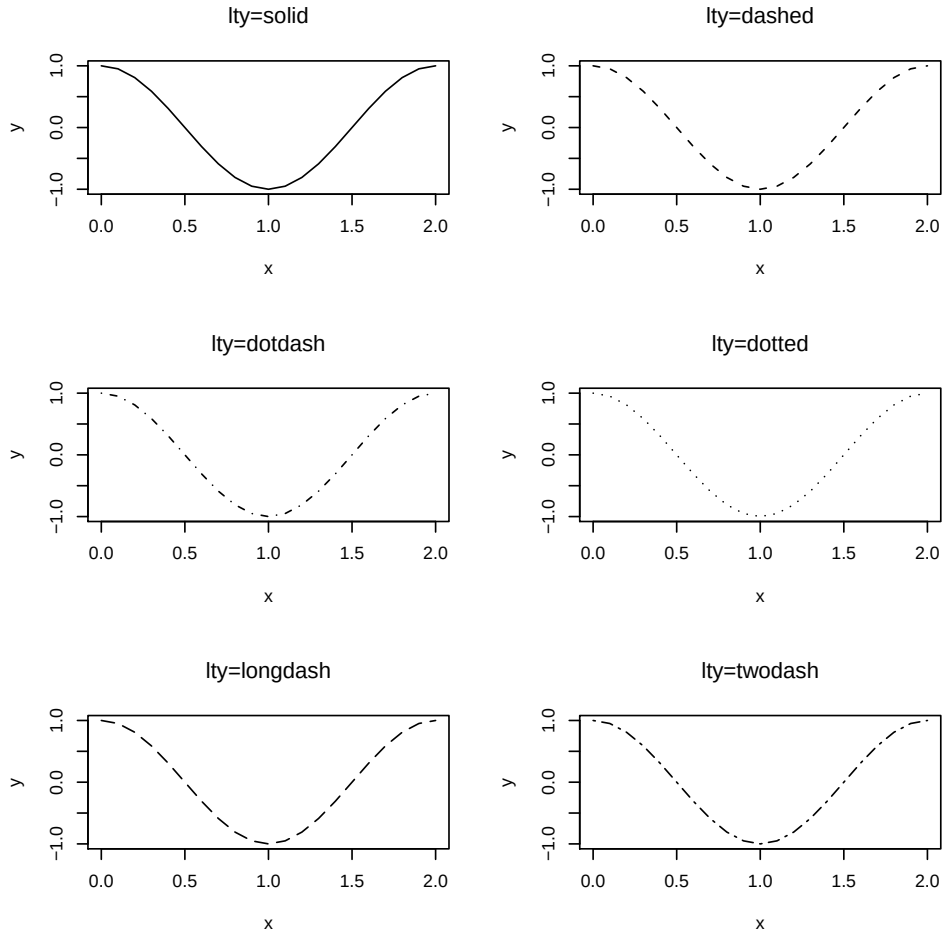

Рис. 7.2: $sin(x)$ при различных значениях аргумента **type**

– **col.axis** — символьная переменная — цвет осей. По умолчанию чёрный — **"black"**.

– **col.main** — символьная переменная — цвет основного заголовка. По умолчанию чёрный — **"black"**.

– **col.sub** — символьная переменная — цвет дополнительного заголовка. По умолчанию чёрный — **"black"**.

– **fg** — символьная переменная — цвет рамки вокруг графика.

– **font.axis**, **font.lab**, **font.main** и **font.sub** — числовые аргументы — тип шрифта для названия осей, основного и дополнительного заголовков. Возможные значения: **1** — простой (по умолчанию), **2** — жирный; **3** — курсив и **4** — жирный курсив.

– **lwd** — положительное целое число — толщина линии; по умолчанию **lwd=1**.

– **xlim** — числовой вектор **c(x1, x2)** — пределы по оси x.

– **ylim** — числовой вектор **c(y1, y2)** — пределы по оси y.

– **frame** — логический аргумент — нужно ли строить рамку вокруг графика.

Другой вариант использования функции **plot()**. В качестве первого аргумента задаётся имя функции, вычисляющей значения по оси **y**, второй и третий аргументы — начальное и конечное значение по оси **x**.

Пример 43. *Построим график функции* $y = \cos(x)$ *на отрезке* $[-2\pi; 2\pi]$ *двумя способами и выведем в едином графическом окне.*

Сначала создадим графическое окно и разделим его на два подокна.

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

Первый график построим, задавая массивы точек по оси **x** *и оси* **y**.

```
x=seq(-2*pi,2*pi,by=0.01)
y=cos(x)
```

Строим график:

```
plot(x,y,type='l',lty=1,col='black',ylab='cos(x)')
```

Теперь возъмём в качестве первого аргумента **plot()** *имя функции* — cos, *а также зададим начальное и конечное значения по оси* **x** — -2π *и* 2π, *соответственно.*

```
plot(cos,-2*pi,2*pi)
```

Результат представлен на рисунке 7.3.

Вид функции **plot()** по умолчанию:

```
plot(x, y = NULL, type = "p",  xlim = NULL, ylim = NULL,
     log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
     ann = par("ann"), axes = TRUE, frame.plot = axes,
     panel.first = NULL, panel.last = NULL, asp = NA, ...)
```

Для полной картины рассмотрим следующие аргументы:

- **ann** — логический параметр — нужно ли выводить название рисунка и осей; по умолчанию значение **TRUE**.

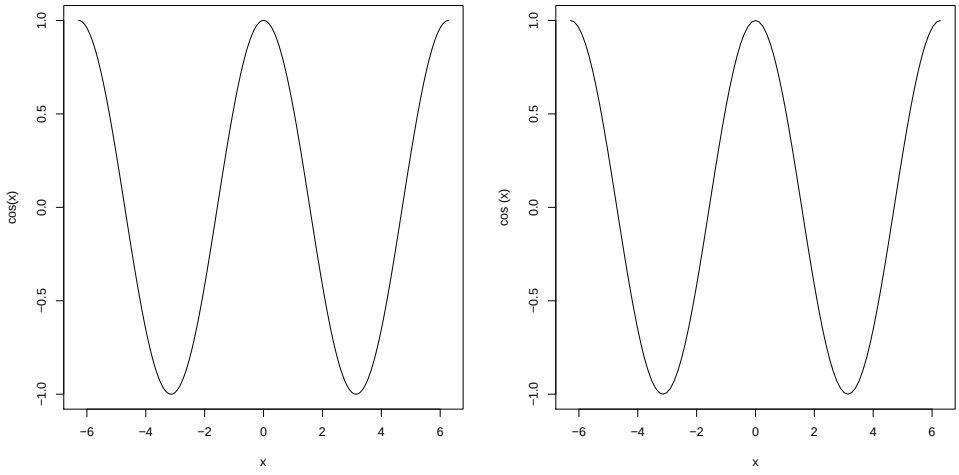

Рис. 7.3: График функции $y = \cos(x)$ на отрезке $[-2\pi; 2\pi]$

- **axes** — логический параметр — выводить обе оси или нет (значение **axes = NULL**также подавляет вывод осей).
- **xaxt** и **yaxt** — символьные аргументы — убирают ось **0x** (**xaxt='n'**) и ось **0y** (**yaxt='n'**).
- **frame.plot** — логический параметр; при значении **TRUE** строится рамка вокруг графика.
- **panel.first** — выражение — оценивается после построения осей на графике, но до построения самого графика. Полезно при построении сетки на графике.
- **panel.last** — выражение — оценивается после вывода графика.
- **asp** — соотношение y/x.
- **lab** — числовой вектор вида **c(x, y, len)**, где **x** и **y** — число меток на соответствующих осях, **len** — длина подписей к меткам (не используемый параметр). Значение аргумента по умолчанию **lab = c(5, 5, 7)**.
- **las** — числовой аргумент — стиль названия осей относительно самих осей. Возможные значения: **0** — названия выводятся параллельно осям (по умолчанию), **1** — горизонтально расположение, **2** — перпендикулярно, **3** — вертикально.

7.1.3 Управление графическим окном

Управление графическим окном (разбиение на несколько подокон) можно осуществлять как при помощи функции **par()** с аргументами **mfrow** и **mfcol**, так и при помощи следующих функций:

```
split.screen(figs, screen, erase = TRUE)
screen(n = , new = TRUE)
erase.screen(n = )
close.screen(n, all.screens = FALSE)
```

- **split.screen()** — графическое окно разбивается на заданное число окон, каждое из которых рассматривается как отдельное графическое окно; полезно при построении большого количества графиков в одном окне.
- **screen()** — выбор подокна, в котором будет построен график.
- **erase.screen()** — очистка определённого подокна (закрашивается цветом фона)
- **close.screen()** — удаление всех выбранных подокон.

Аргументы этих функций:

- **figs** — числовой вектор, состоящий из двух элементов — число подокон по строкам и по столбцам (**figs=c(2,3)** означает, что графическое окно разбивается на 2 строки, в каждой из которых 3 подокна).
- **screen** — номер разбиваемого подокна.
- **erase** — логический аргумент — нужно ли очищать данное окно.
- **n** — номер окна в котором строится график (функция **screen(n)**), либо которое очищается (**erase.screen()**) или закрывается (**close.screen()**).
- **new** — логический аргумент; при значении **TRUE** графическое окно будет очищено перед построением нового графика.
- **all.screens** — логический аргумент —следует ли закрыть все окна.

Замечания.

1. Первый вызов функции **split.screen()** переводит **R** в режим работы с несколькими окнами. Все остальные функции — **screen()**, **erase.screen()**, **close.screen()** и **split.screen()** (для дальнейшего разбиения) будут работать только в этом установленном режиме. Выход из режима разбиения окна осуществляется при помощи **close.screen(all = TRUE)**.

2. Первое созданное в результате разбиения окно является активным (т.е для работы в нём не надо обращаться к нему по номеру).

3. Рекомендуется полностью построить график и все дополнения к нему в текущем графическом окне, прежде чем переходить к следующему. Повторное обращение к окну с уже построенным изображением возможно, но результат внесения изменений непредсказуем.

4. Функции **screen()**, **erase.screen()**, **close.screen()** и **split.screen()** несовместимы с функциями **par(mfrow)** и **par(mfcol)**.

5. Функция **erase.screen** не будет работать,если фон графического устройства прозрачный (а в **R** это значение фона по умолчанию).

6. По окончанию работы в режиме **split.screen** обязательно нужно выйти из него — **close.screen(all = TRUE)**.

Рассмотрим работу функций на примере.

Пример 44. *Сначала установим фон графика (по умолчанию он прозрачный.)*

```
par(bg = "white")
```

Теперь разобьём графическое окно на две части:

```
split.screen(c(2,1))
```

Нижнюю часть снова разобьём — только теперь на три части

```
split.screen(c(1,3), screen = 2)
```

Проверим, сколько экранов всего получилось.

```
> split.screen()
[1] 1 2 3 4 5
```

Итого — 5 окон: два больших и три маленьких — части второго большого окна.

Выберем первое окно и построим в нём график.

```
screen(1)
plot(cos,-pi,pi)
```

Теперь построим график в 4 окне.

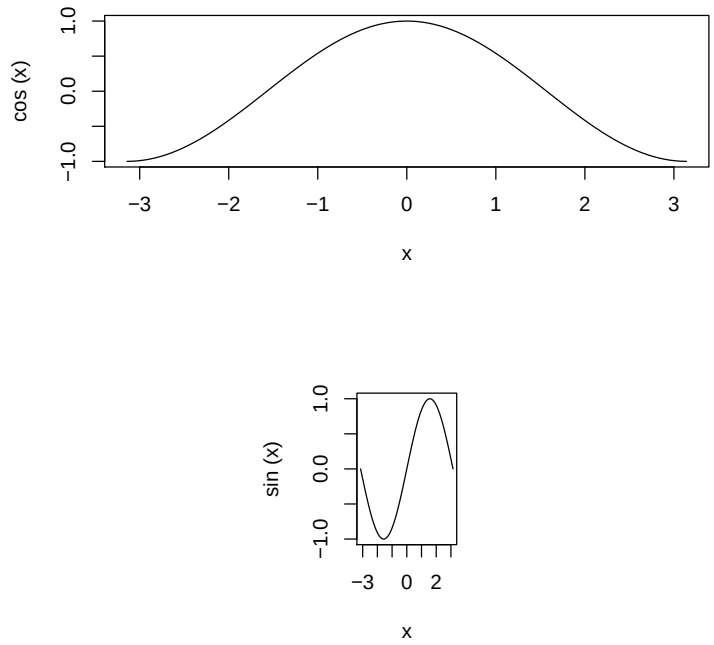

Рис. 7.4: Разбиение графического окна

```
screen(4)
plot(sin,-pi,pi)
```

Выходим из режима разбиения.

```
close.screen(all = TRUE)
```

Результат представлен на рис.7.4

Пример 45. *Снова изменяем фон графического окна, разбиваем сначала окно на две части, вторую из которых ещё на три.*

```
par(bg = "white")
split.screen(c(2,1))
split.screen(c(1,2),2)
```

Активным является первое из трёх окон, созданное в результате второго разбиения. Построим в нём график.

```
plot(1:10)
```

Хотим добавить название оси **y**. *Для этого стираем созданный график и снова его строим.*

```
erase.screen()
plot(1:10, ylab='')
```

Переходим к первому окну, а затем к четвёртому.

```
screen(1)
plot(1:10)
screen(4)
plot(1:10, ylab="ylab 4")
```

Возвращаемся к первому окну, но не стираем его.

```
screen(1, FALSE)
plot(10:1, axes=FALSE, lty=2, ylab="")
```

Выходим из режима разбиения.

```
close.screen(all = TRUE)
```

Результат представлен на рис.7.5

7.1.4 Функция contour()

Функция **contour()** — создаёт график с контурными линиями либо добавляет контурные линии к уже существующему графику.

Полный вид функции:

```
contour(x = seq(0, 1, length.out = nrow(z)),
        y = seq(0, 1, length.out = ncol(z)),
        z, nlevels = 10, levels = pretty(zlim, nlevels),
        labels = NULL,
        xlim = range(x, finite = TRUE),
        ylim = range(y, finite = TRUE),
        zlim = range(z, finite = TRUE),
        labcex = 0.6, drawlabels = TRUE, method = "flattest",
        vfont, axes = TRUE, frame.plot = axes,
        col = par("fg"), lty = par("lty"), lwd = par("lwd"),
        add = FALSE, ...)
```

Аргументы:

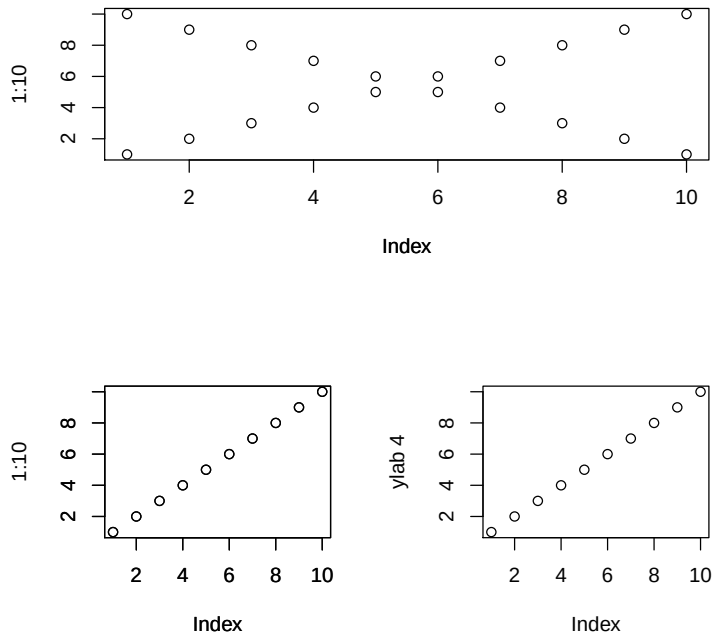

Рис. 7.5: Разбиение окна

- **x, y** — числовые вектора — координаты линий сетки, в которых определяются значения **z**. Значения векторов **x** и **y** должны конечны, не должно быть **NA** и должны располагаться в возрастающем порядке. По умолчанию эти вектора принимают значения между 0 и 1. Если **x** —список, то его компоненты **x$x** и **x$y** задают вектора **x** и **y**. Если в списке есть компонента **x$z**, то её значения определяют аргумент **z**.
- **z** — числовая матрица — значения в точках **(x,y)**: $z = (f(x_i, y_j))_{ij}$.
- **nlevels** — число контурных уровней, если не задан аргумент **levels**.
- **levels** — числовой вектор уровней, в которых строятся контурные линии.
- **labels** — символьный вектор — метки контурных линий.
- **labcex** — размер контурных линий.
- **drawlabels** — логический аргумент — вывод меток контурных линий.
- **method** — символьный аргумент — определение места расположения меток. Возможные значения: **'simple'**, **'edge'** и **'flattest'** (по умолчанию).

'simple' — метки выводятся на краю графика и накладываются на контурную линию. **'edge'** — метки выводятся на краю графика, вписываются в контурную линию, различные метки не пересекаются. **'flattest'** — метки встраиваются в плоской части контурной линии, метки не пересекаются. Задание второго или третьего метода может привести к тому, что не все метки будут выведены.

- **vfont** — задание семейства шрифтов для меток. Если значение **NULL** — используется семейство шрифтов по умолчанию.
- **xlim**, **ylim** и **zlim** — предельные значения для **x**, **y** и **z**.
- **axes** и **frame.plot** — логические аргументы — нужно ли строить оси и рамку вокруг графика.
- **col** — символьный аргумент — цвет контурных линий.
- **lty** — тип линии.
- **lwd** — толщина линии.
- **add** — логический аргумент — нужно ли добавить контурные линии к уже существующему графику (**add**=**TRUE**) или надо создать новый.
- ... — дополнительные аргументы, отвечающие за вид осей, заголовков и пр.

Пример 46. *Изобразим «узор персидского ковра» (рис.7.6).*
Зададим значения векторов **x** *и* **y**

```
x <- y <- seq(-4*pi, 4*pi, len = 27)
```

и определим на них функцию.

```
r <- sqrt(outer(x^2, y^2, "+"))
```

Разобьём графическое окно на 4 части и зададим размеры отступов от графиков.

```
opar <- par(mfrow = c(2, 2), mar = rep(0, 4))
```

Теперь для различных значений параметра $f = i\pi, \quad \pi = \overline{0,3}$ *построим график функции* $z = \cos(r^2) \cdot e^{-r/f}$, *где* $r = \sqrt{x^2 + y^2}$.

```
for(f in pi^(0:3))
  contour(cos(r^2)*exp(-r/f),
          drawlabels = FALSE, axes = FALSE, frame = TRUE)
```

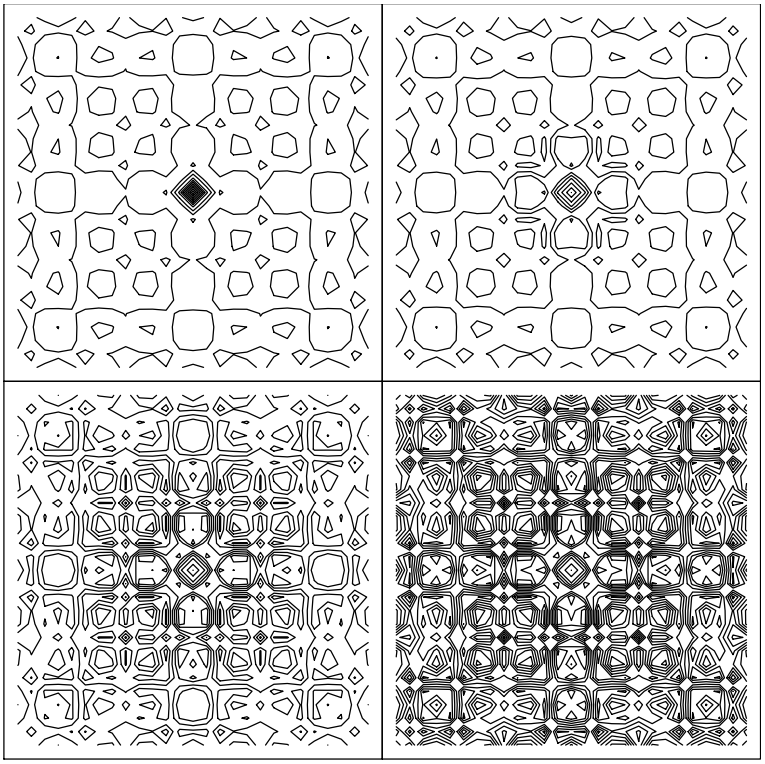

Рис. 7.6: Узор персидского ковра

Пример 47. *Построим контурный график при различных значениях аргумента* **method**. *Сначала зададим элементы вектора* **x**.

```
x <- -6:16
```

Разбиваем графическое окно на 4 части.

```
op <- par(mfrow = c(2, 2))
```

Первый контурный график:

```
contour(outer(x, x), method = "edge", vfont = c("sans serif", "plain"))
```

Для второго графика определим $z = x/\sqrt{|x|}$. *Построим сначала закрашенную сетку при помощи функции* **image(x, x, z)**, *на которой уже зададим контурные линии.*

```
z <- outer(x, sqrt(abs(x)), FUN = "/")
image(x, x, z)
contour(x, x, z, col = "pink", add = TRUE, method = "edge",
        vfont = c("sans serif", "plain"))
```

Третий график:

```
contour(x, x, z, ylim = c(1, 6), method = "simple", labcex = 1)
```

и четвёртый.

```
contour(x, x, z, ylim = c(-6, 6), nlev = 20, lty = 2, method = "simple")
```

Результат представлен на рис.7.7.

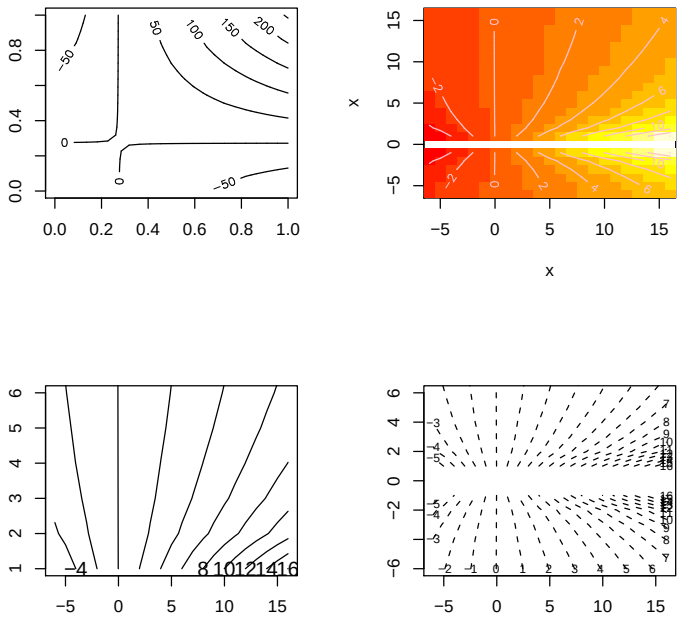

Рис. 7.7: Контурные линии при различных значениях аргумента **method**

7.1.5 Функция curve()

Функция **curve( )** строит график заданной функции на некотором интервале.

```
curve(expr, from = NULL, to = NULL, n = 101, add = FALSE,
      type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)
```

Аргументы функции:

- **expr** — выражение как функция от x или имя функции, график которой нужно построить.

- **from** и **to** — числовые аргументы — границы отрезка, на котором строится график. Если не заданы эти параметры, то границы определяются значением аргумента x заданного выражения. Если же и x не определён, то график строится на отрезке $[0; 1]$.

- **n** — числовой аргумент — число точек, в которых определяется значение **expr**.

- **add** — логический аргумент — нужно ли добавлять строящийся график к уже существующему (**add**=**TRUE**) или нужно создать новое графическое окно (**add**=**FALSE**).

Пример 48. *Построим графики нескольких функций* — $f_1(x) = x^2 - 3x$, $f_2(x) = x^2 - 2$, $f_3(x) = \cos(x)$ *и* $f_4(x) = \sin\left(\cos(x) \cdot e^{-x/2}\right)$ *при помощи* **curve()**, *результат представлен на рис.7.8.*

```
par(mfrow=c(2,2))
curve(x^3-3*x, -2, 2)
curve(x^2-2, add = TRUE, col = "violet")
plot(cos, -pi,  3*pi)
plot(cos, xlim = c(-pi,3*pi), n = 1001, col = "blue", add=TRUE)
f1 <- function(x) sin(cos(x)*exp(-x/2))
curve(f1, -8, 7, n=2001)
plot (f1, -8, -5)
```

7.1.6 Задание цвета в R

В **R** реализовано несколько возможностей задания цвета.

Первая — задать символьную переменную — название нужного цвета. Со списком доступных цветов можно ознакомиться при помощи функций **colors()** или **colours()** — будет выведен вектор из 657 названий цветов. Если результат вызова функции **colors()** (или **colours()**) присвоить некоторой переменной (т.е. создать вектор цветов), то можно будет использовать элементы этого вектора.

```
> color=colors()
> color[10:30]
 [1] "aquamarine2"    "aquamarine3"    "aquamarine4"    "azure"
 [5] "azure1"         "azure2"         "azure3"         "azure4"
```

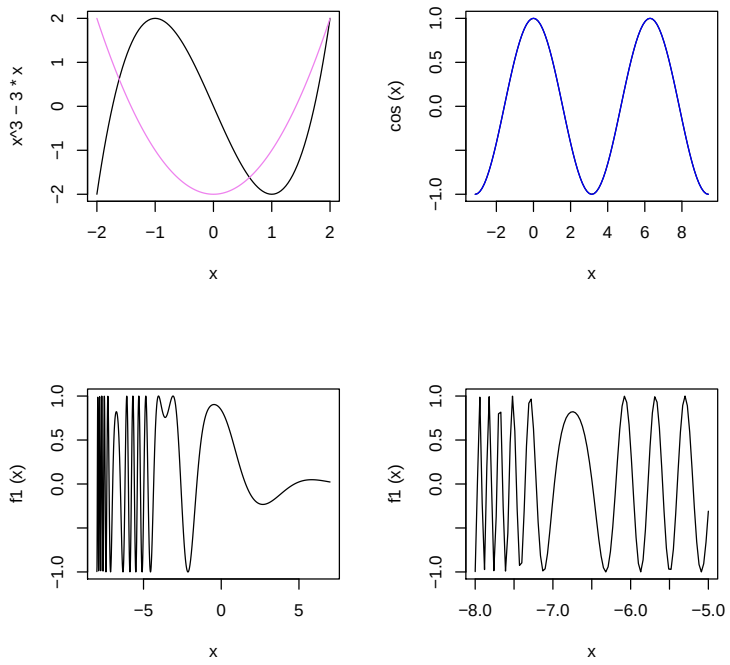

Рис. 7.8: Использование функции **curve()**

```
 [9] "beige"           "bisque"          "bisque1"         "bisque2"
[13] "bisque3"         "bisque4"         "black"           "blanchedalmond"
[17] "blue"            "blue1"           "blue2"           "blue3"
[21] "blue4"
```

Или проще

```
> colors()[1:5]
[1] "white"          "aliceblue"      "antiquewhite"   "antiquewhite1"
[5] "antiquewhite2"
```

```
> colors()
  [1] "white"               "aliceblue"           "antiquewhite"
  [4] "antiquewhite1"       "antiquewhite2"       "antiquewhite3"
  [7] "antiquewhite4"       "aquamarine"          "aquamarine1"
 [10] "aquamarine2"         "aquamarine3"         "aquamarine4"
 [13] "azure"               "azure1"              "azure2"
 [16] "azure3"              "azure4"              "beige"
 [19] "bisque"              "bisque1"             "bisque2"
 [22] "bisque3"             "bisque4"             "black"
 [25] "blanchedalmond"      "blue"                "blue1"
```

```
 [28] "blue2"              "blue3"              "blue4"
 [31] "blueviolet"         "brown"              "brown1"
 [34] "brown2"             "brown3"             "brown4"
 [37] "burlywood"          "burlywood1"         "burlywood2"
 [40] "burlywood3"         "burlywood4"         "cadetblue"
 [43] "cadetblue1"         "cadetblue2"         "cadetblue3"
 [46] "cadetblue4"         "chartreuse"         "chartreuse1"
 [49] "chartreuse2"        "chartreuse3"        "chartreuse4"
 [52] "chocolate"          "chocolate1"         "chocolate2"
 [55] "chocolate3"         "chocolate4"         "coral"
 [58] "coral1"             "coral2"             "coral3"
 [61] "coral4"             "cornflowerblue"     "cornsilk"
 [64] "cornsilk1"          "cornsilk2"          "cornsilk3"
 [67] "cornsilk4"          "cyan"               "cyan1"
 [70] "cyan2"              "cyan3"              "cyan4"
 [73] "darkblue"           "darkcyan"           "darkgoldenrod"
 [76] "darkgoldenrod1"     "darkgoldenrod2"     "darkgoldenrod3"
 [79] "darkgoldenrod4"     "darkgray"           "darkgreen"
 [82] "darkgrey"           "darkkhaki"          "darkmagenta"
 [85] "darkolivegreen"     "darkolivegreen1"    "darkolivegreen2"
 [88] "darkolivegreen3"    "darkolivegreen4"    "darkorange"
 [91] "darkorange1"        "darkorange2"        "darkorange3"
 [94] "darkorange4"        "darkorchid"         "darkorchid1"
 [97] "darkorchid2"        "darkorchid3"        "darkorchid4"
[100] "darkred"            "darksalmon"         "darkseagreen"
[103] "darkseagreen1"      "darkseagreen2"      "darkseagreen3"
[106] "darkseagreen4"      "darkslateblue"      "darkslategray"
[109] "darkslategray1"     "darkslategray2"     "darkslategray3"
[112] "darkslategray4"     "darkslategrey"      "darkturquoise"
[115] "darkviolet"         "deeppink"           "deeppink1"
[118] "deeppink2"          "deeppink3"          "deeppink4"
[121] "deepskyblue"        "deepskyblue1"       "deepskyblue2"
[124] "deepskyblue3"       "deepskyblue4"       "dimgray"
[127] "dimgrey"            "dodgerblue"         "dodgerblue1"
[130] "dodgerblue2"        "dodgerblue3"        "dodgerblue4"
[133] "firebrick"          "firebrick1"         "firebrick2"
[136] "firebrick3"         "firebrick4"         "floralwhite"
[139] "forestgreen"        "gainsboro"          "ghostwhite"
[142] "gold"               "gold1"              "gold2"
[145] "gold3"              "gold4"              "goldenrod"
[148] "goldenrod1"         "goldenrod2"         "goldenrod3"
[151] "goldenrod4"         "gray"               "gray0"
[154] "gray1"              "gray2"              "gray3"
[157] "gray4"              "gray5"              "gray6"
[160] "gray7"              "gray8"              "gray9"
[163] "gray10"             "gray11"             "gray12"
[166] "gray13"             "gray14"             "gray15"
[169] "gray16"             "gray17"             "gray18"
[172] "gray19"             "gray20"             "gray21"
```

```
[175] "gray22"          "gray23"          "gray24"
[178] "gray25"          "gray26"          "gray27"
[181] "gray28"          "gray29"          "gray30"
[184] "gray31"          "gray32"          "gray33"
[187] "gray34"          "gray35"          "gray36"
[190] "gray37"          "gray38"          "gray39"
[193] "gray40"          "gray41"          "gray42"
[196] "gray43"          "gray44"          "gray45"
[199] "gray46"          "gray47"          "gray48"
[202] "gray49"          "gray50"          "gray51"
[205] "gray52"          "gray53"          "gray54"
[208] "gray55"          "gray56"          "gray57"
[211] "gray58"          "gray59"          "gray60"
[214] "gray61"          "gray62"          "gray63"
[217] "gray64"          "gray65"          "gray66"
[220] "gray67"          "gray68"          "gray69"
[223] "gray70"          "gray71"          "gray72"
[226] "gray73"          "gray74"          "gray75"
[229] "gray76"          "gray77"          "gray78"
[232] "gray79"          "gray80"          "gray81"
[235] "gray82"          "gray83"          "gray84"
[238] "gray85"          "gray86"          "gray87"
[241] "gray88"          "gray89"          "gray90"
[244] "gray91"          "gray92"          "gray93"
[247] "gray94"          "gray95"          "gray96"
[250] "gray97"          "gray98"          "gray99"
[253] "gray100"         "green"           "green1"
[256] "green2"          "green3"          "green4"
[259] "greenyellow"     "grey"            "grey0"
[262] "grey1"           "grey2"           "grey3"
[265] "grey4"           "grey5"           "grey6"
[268] "grey7"           "grey8"           "grey9"
[271] "grey10"          "grey11"          "grey12"
[274] "grey13"          "grey14"          "grey15"
[277] "grey16"          "grey17"          "grey18"
[280] "grey19"          "grey20"          "grey21"
[283] "grey22"          "grey23"          "grey24"
[286] "grey25"          "grey26"          "grey27"
[289] "grey28"          "grey29"          "grey30"
[292] "grey31"          "grey32"          "grey33"
[295] "grey34"          "grey35"          "grey36"
[298] "grey37"          "grey38"          "grey39"
[301] "grey40"          "grey41"          "grey42"
[304] "grey43"          "grey44"          "grey45"
[307] "grey46"          "grey47"          "grey48"
[310] "grey49"          "grey50"          "grey51"
[313] "grey52"          "grey53"          "grey54"
[316] "grey55"          "grey56"          "grey57"
[319] "grey58"          "grey59"          "grey60"
```

```
[322] "grey61"              "grey62"              "grey63"
[325] "grey64"              "grey65"              "grey66"
[328] "grey67"              "grey68"              "grey69"
[331] "grey70"              "grey71"              "grey72"
[334] "grey73"              "grey74"              "grey75"
[337] "grey76"              "grey77"              "grey78"
[340] "grey79"              "grey80"              "grey81"
[343] "grey82"              "grey83"              "grey84"
[346] "grey85"              "grey86"              "grey87"
[349] "grey88"              "grey89"              "grey90"
[352] "grey91"              "grey92"              "grey93"
[355] "grey94"              "grey95"              "grey96"
[358] "grey97"              "grey98"              "grey99"
[361] "grey100"             "honeydew"            "honeydew1"
[364] "honeydew2"           "honeydew3"           "honeydew4"
[367] "hotpink"             "hotpink1"            "hotpink2"
[370] "hotpink3"            "hotpink4"            "indianred"
[373] "indianred1"          "indianred2"          "indianred3"
[376] "indianred4"          "ivory"               "ivory1"
[379] "ivory2"              "ivory3"              "ivory4"
[382] "khaki"               "khaki1"              "khaki2"
[385] "khaki3"              "khaki4"              "lavender"
[388] "lavenderblush"       "lavenderblush1"      "lavenderblush2"
[391] "lavenderblush3"      "lavenderblush4"      "lawngreen"
[394] "lemonchiffon"        "lemonchiffon1"       "lemonchiffon2"
[397] "lemonchiffon3"       "lemonchiffon4"       "lightblue"
[400] "lightblue1"          "lightblue2"          "lightblue3"
[403] "lightblue4"          "lightcoral"          "lightcyan"
[406] "lightcyan1"          "lightcyan2"          "lightcyan3"
[409] "lightcyan4"          "lightgoldenrod"      "lightgoldenrod1"
[412] "lightgoldenrod2"     "lightgoldenrod3"     "lightgoldenrod4"
[415] "lightgoldenrodyellow" "lightgray"          "lightgreen"
[418] "lightgrey"           "lightpink"           "lightpink1"
[421] "lightpink2"          "lightpink3"          "lightpink4"
[424] "lightsalmon"         "lightsalmon1"        "lightsalmon2"
[427] "lightsalmon3"        "lightsalmon4"        "lightseagreen"
[430] "lightskyblue"        "lightskyblue1"       "lightskyblue2"
[433] "lightskyblue3"       "lightskyblue4"       "lightslateblue"
[436] "lightslategray"      "lightslategrey"      "lightsteelblue"
[439] "lightsteelblue1"     "lightsteelblue2"     "lightsteelblue3"
[442] "lightsteelblue4"     "lightyellow"         "lightyellow1"
[445] "lightyellow2"        "lightyellow3"        "lightyellow4"
[448] "limegreen"           "linen"               "magenta"
[451] "magenta1"            "magenta2"            "magenta3"
[454] "magenta4"            "maroon"              "maroon1"
[457] "maroon2"             "maroon3"             "maroon4"
[460] "mediumaquamarine"    "mediumblue"          "mediumorchid"
[463] "mediumorchid1"       "mediumorchid2"       "mediumorchid3"
[466] "mediumorchid4"       "mediumpurple"        "mediumpurple1"
```

```
[469] "mediumpurple2"       "mediumpurple3"       "mediumpurple4"
[472] "mediumseagreen"      "mediumslateblue"     "mediumspringgreen"
[475] "mediumturquoise"     "mediumvioletred"     "midnightblue"
[478] "mintcream"           "mistyrose"           "mistyrose1"
[481] "mistyrose2"          "mistyrose3"          "mistyrose4"
[484] "moccasin"            "navajowhite"         "navajowhite1"
[487] "navajowhite2"        "navajowhite3"        "navajowhite4"
[490] "navy"                "navyblue"            "oldlace"
[493] "olivedrab"           "olivedrab1"          "olivedrab2"
[496] "olivedrab3"          "olivedrab4"          "orange"
[499] "orange1"             "orange2"             "orange3"
[502] "orange4"             "orangered"           "orangered1"
[505] "orangered2"          "orangered3"          "orangered4"
[508] "orchid"              "orchid1"             "orchid2"
[511] "orchid3"             "orchid4"             "palegoldenrod"
[514] "palegreen"           "palegreen1"          "palegreen2"
[517] "palegreen3"          "palegreen4"          "paleturquoise"
[520] "paleturquoise1"      "paleturquoise2"      "paleturquoise3"
[523] "paleturquoise4"      "palevioletred"       "palevioletred1"
[526] "palevioletred2"      "palevioletred3"      "palevioletred4"
[529] "papayawhip"          "peachpuff"           "peachpuff1"
[532] "peachpuff2"          "peachpuff3"          "peachpuff4"
[535] "peru"                "pink"                "pink1"
[538] "pink2"               "pink3"               "pink4"
[541] "plum"                "plum1"               "plum2"
[544] "plum3"               "plum4"               "powderblue"
[547] "purple"              "purple1"             "purple2"
[550] "purple3"             "purple4"             "red"
[553] "red1"                "red2"                "red3"
[556] "red4"                "rosybrown"           "rosybrown1"
[559] "rosybrown2"          "rosybrown3"          "rosybrown4"
[562] "royalblue"           "royalblue1"          "royalblue2"
[565] "royalblue3"          "royalblue4"          "saddlebrown"
[568] "salmon"              "salmon1"             "salmon2"
[571] "salmon3"             "salmon4"             "sandybrown"
[574] "seagreen"            "seagreen1"           "seagreen2"
[577] "seagreen3"           "seagreen4"           "seashell"
[580] "seashell1"           "seashell2"           "seashell3"
[583] "seashell4"           "sienna"              "sienna1"
[586] "sienna2"             "sienna3"             "sienna4"
[589] "skyblue"             "skyblue1"            "skyblue2"
[592] "skyblue3"            "skyblue4"            "slateblue"
[595] "slateblue1"          "slateblue2"          "slateblue3"
[598] "slateblue4"          "slategray"           "slategray1"
[601] "slategray2"          "slategray3"          "slategray4"
[604] "slategrey"           "snow"                "snow1"
[607] "snow2"               "snow3"               "snow4"
[610] "springgreen"         "springgreen1"        "springgreen2"
[613] "springgreen3"        "springgreen4"        "steelblue"
```

```
[616] "steelblue1"          "steelblue2"          "steelblue3"
[619] "steelblue4"          "tan"                 "tan1"
[622] "tan2"                "tan3"                "tan4"
[625] "thistle"             "thistle1"            "thistle2"
[628] "thistle3"            "thistle4"            "tomato"
[631] "tomato1"             "tomato2"             "tomato3"
[634] "tomato4"             "turquoise"           "turquoise1"
[637] "turquoise2"          "turquoise3"          "turquoise4"
[640] "violet"              "violetred"           "violetred1"
[643] "violetred2"          "violetred3"          "violetred4"
[646] "wheat"               "wheat1"              "wheat2"
[649] "wheat3"              "wheat4"              "whitesmoke"
[652] "yellow"              "yellow1"             "yellow2"
[655] "yellow3"             "yellow4"             "yellowgreen"
```

На рис.7.9 приведены все цвета в **R**.

```
x=1:657
y1=rep(1,length(x))
plot(x,y1,type='l',lty=1,col=colors()[5],xlim=c(0,658),ylim=c(0,658))
for (i in 2:657) abline(h=i,col=colors()[i])
```

Также цвет графика можно задать в формате RGB — символьная переменная вида "#**RRGGBB**", где пары RR, GG и BB состоят из двух шестнадцатеричных цифр и, следовательно, могут принимать значения от 00 до FF.

Третий вариант — цвет можно задать как элемент вектора-палитры с помощью функции **palette()**

```
> palette()
[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
[8] "gray"
> z=palette()
> z[3]
[1] "green3"
```

Четвёртый вариант — цвет можно за задать как функцию **rgb(red, green, blue)**, где **red, green, blue** $= \overline{0,1}$.

```
> rgb(0,0,1)
[1] "#0000FF"
> rgb(0,0,0.2)
[1] "#000033"
> rgb(0.1,0.1,0.2)
[1] "#1A1A33"
```

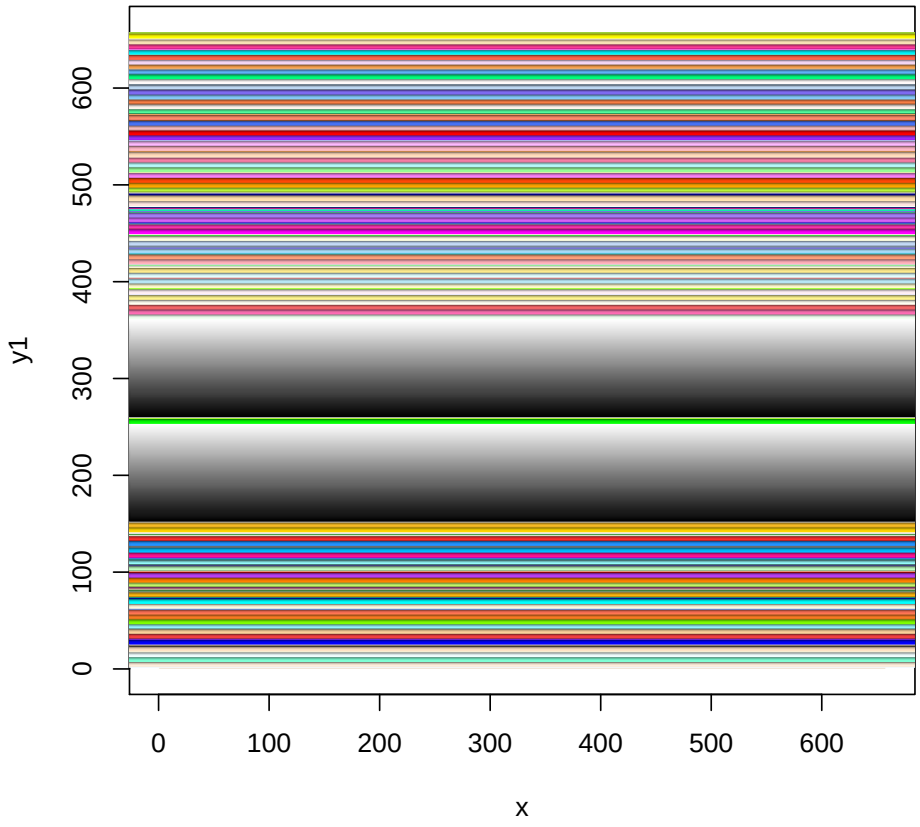

Рис. 7.9: Цветовая палитра в **R**

7.2 Функции низкого уровня

К функциям низкого уровня можно отнести:

- **abline()** — построение прямых линий в уже существующем графическом окне;
- **arrows()** — рисование стрелок;
- **axis()** — построение оси графика;
- **box()** — построение рамки вокруг графика;
- **grid()** — задание прямоугольной сетки на графике;

- **legend()** — задание различных легенд к графику;
- **lines()** — построение линий, соединяющих заданные точки;
- **mtext()** — вывод надписей в соответствующей области;
- **points()** — добавление точек на график;
- **polygon()** — построение многоугольников;
- **rect()** — построение прямоугольников;
- **segments()** — соединение точек прямыми отрезками;
- **symbols()** — построение одного из шести видов фигур (круг, квадрат, прямоугольник, звезда, термометр, **boxplot**) на графике;
- **text()** — добавление текста к графику;
- **title()** — добавление заголовков;
- **xspline()** — построение сплайна относительно заданных контрольных точек.

7.2.1 Добавление новых объектов на график — функции abline(), lines(), arrows(), points(), polygon(), rect(), segments(), symbols()

В этой части рассмотрим низкоуровневые функции, позволяющие добавлять различные графические объекты к уже построенному графику.

Функция abline()

Вид функции:

```
abline(a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
       coef = NULL, untf = FALSE, ...)
```

добавляет одну или несколько прямых линий вида $y = a + bx$ к созданному ранее графику.

Аргументы функции:

- **a** и **b** — числовые аргументы — точка пересечения с осью **y** и коэффициент наклона, соответственно.

- **h** — числовой аргумент — значение по оси **y** для построения горизонтальной линии.
- **v** — числовой аргумент — значение по оси **x** для построения вертикальной линии.
- **coef** — числовой вектор вида **c(a,b)** — задаёт аргументы **a** и **b**.
- **reg** — числовой аргумент — определяет вид прямой. Если это
  - числовой вектор длины 1 — задан коэффициент наклона **b** прямой $y = bx$, проходящей через начало координат;
  - числовой вектор длины 2 — заданы аргументы **a** и **b**.
- **untf** — логический аргумент — отвечает за приведение строящегося графика в соответствие с заданной системой координат. Если **untf=TRUE**, а одна из осей (или обе) были переведены в логарифмический формат (аргументы **xlog** и **ylog**), то прямая всё равно строится в исходной (не преобразованной) системе координат, в противном случае прямая строится согласно модифицированным координатам.
- `...` — дополнительные аргументы:
  - **col** — цвет линии.
  - **lty** — тип линии.
  - **lwd** — ширина линии.

Возможные варианты использования:

```
abline(a, b, untf = FALSE, ...)
abline(h=, untf = FALSE, ...)
abline(v=, untf = FALSE, ...)
abline(coef=, untf = FALSE, ...)
abline(reg=, untf = FALSE, ...)
```

Пример 49. *На примере покажем различные способы построения прямых линий в* **R**. *Результат приведён на рис.7.10.*

```
x=seq(-pi,pi,by=0.01)
y=cos(x)
plot(x,y, type='n',xlab="x", ylab="y", xlim=c(-pi-0.1,pi+0.1),
ylim=c(-1.1,1.1))
abline(a=0,b=1,lty=1)
```

```
abline(a=0,b=-1,lty=2)
abline(coef=c(0,0.5),lty=3)
abline(reg=c(0,-0.5),lty=4)
abline(v = -3:3, col=colors()[451:457])
abline(h=seq(-1,1,by=0.5), col=colors()[21:25])
abline(h=0,v=0,lwd=2,col='black')
```

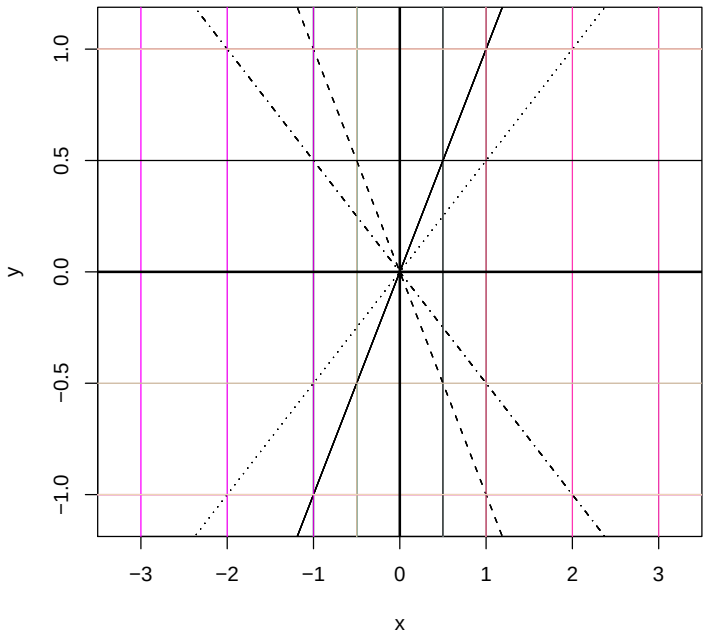

Рис. 7.10: Построение прямых линий

Функция lines()

Функция **lines()** соединяет заданные точки отрезками. Её вид:

```
lines(x, y = NULL, type = "l", ...)
```

Единственным аргументом функции является аргумент **x** — список из двух компонент (первая компонента — координаты по оси **0x**, вторая — координаты по оси **0y**) или матрица из двух столбцов (первый столбец — координаты по оси **0x**, второй — координаты по оси **0y**).

Если аргумент **x** — числовой вектор — координаты по оси **0x**, то должен быть задан аргумент **y** — координаты по оси **0y**.

Среди значений аргументов **x** и (или) **y** могут быть и **NA**. В таком случае к точке с такой координатой (координатами) (или от неё) прямая просто не строится — создаются разрывы в линиях.

Если **type = 'h'**, то параметр **col**, отвечающий за цвет линий, можно задать как вектор.

Пример 50. *Приведём несколько примеров задания координат точек, которые будут соединены линиями.*

```
par(bg='white')
x=seq(-pi,pi,by=0.01)
y=sqrt(abs(cos(x)))
x1=seq(-3,3,by=0.5)
y1=sqrt(abs(cos(x1)))
split.screen(c(2,1))
plot(x,y, type='l',xlab="x", ylab="y", xlim=c(-pi-0.1,pi+0.1),
ylim=c(-0.1,1.1))
lines(x1,y1,col='blue',lwd=2)
lines(x1,y1,type='h',col='green')
split.screen(c(1,2),2)
plot(x,y, type='l',xlab="x", ylab="y", xlim=c(-pi-0.1,pi+0.1),
ylim=c(-0.1,1.1))
z=list(x1,y1)
lines(x1,y1,col='darkred',lwd=2)
screen(4)
plot(x,y, type='l',xlab="x", ylab="y", xlim=c(-pi-0.1,pi+0.1),
ylim=c(-0.1,1.1))
z=cbind(x1,y1)
lines(x1,y1,col='lightgoldenrod',lwd=2)
close.screen(all = TRUE)
```

Функция points()

Функция **points()** рисует точки заданного типа (по умолчанию - круги) в заданных координатах. Вид функции:

```
points(x, y = NULL, type = "p", ...)
```

Снова, только один обязательный аргумент — **x**, если **x** — список или матрица с координатами строящихся точек. Если **x** – числовой вектор — координаты по оси **0x**, то дополнительно задаются координаты по оси **0y** — аргумент **y**.

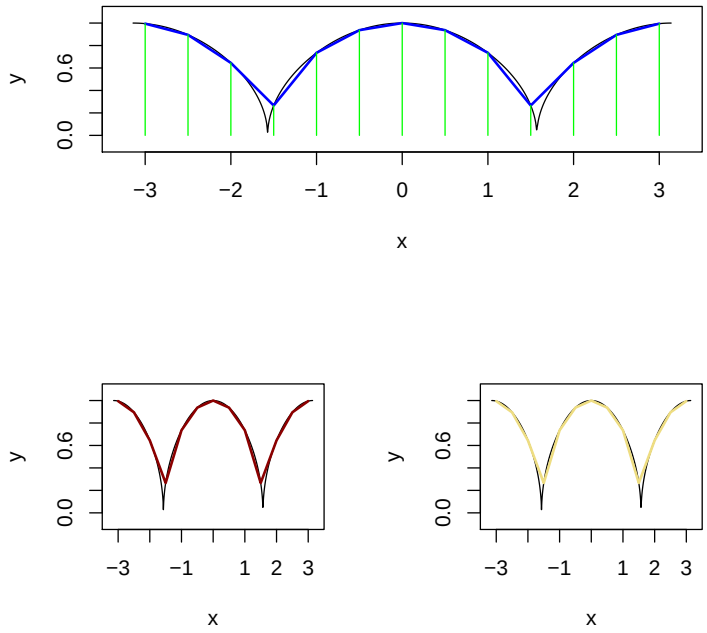

Рис. 7.11: Соединение точек линиями

Отметим ряд дополнительных параметров:

- **pch** — числовой аргумент — тип рисуемого символа в точке с заданными координатами. Возможные значения — от 0 до 255, из них: **0** — □, **1** — ○, **2** — △, **3** — +, **4** — ×, **5** — ◇, **6** — ▽, **7** — ⊠, **8** — ∗, **9** — ромб с крестом внутри, **10** — ⊕, **11** — два пересекающихся треугольника, **12** — ⊞, **13** — ⊗, **14** — квадрат со вписанным треугольником; символы **15**–**25** — закрашиваемые геометрические фигуры; **26**–**31** — не используются и игнорируются при задании; **32**–**127** — ASCII символы; **128**–**255** — различные символы. Часть из них (**0**–**127**) представлены на рис.7.12.

- **cex** — числовой аргумент — во сколько раз увеличить размер выводимого символа.

- **bg** — символьный или числовой аргумент — цвет фона, которым можно закрасить символы **15**–**25**.

- **col** — символьный или числовой аргумент — цвет символа.

Пример 51. *Выведем все символы для функции* **points()**.

```
x=0:20
y=0:20
plot(x,y,type='n')
for (i in 0:14)
points(x[i+1],y[20],pch=i,cex = 1.5)
for (i in 15:25)
points(x[i+1-15],y[18],pch=i,cex = 1.5,col=i,bg=i)
for (i in 32:51)
points(x[i+1-32],y[16],pch=i,cex = 1.5)
for (i in 52:71)
points(x[i+1-52],y[14],pch=i,cex = 1.5)
for (i in 72:91)
points(x[i+1-72],y[12],pch=i,cex = 1.5)
for (i in 92:111)
points(x[i+1-92],y[10],pch=i,cex = 1.5)
for (i in 112:127)
points(x[i+1-112],y[8],pch=i,cex = 1.5)
```

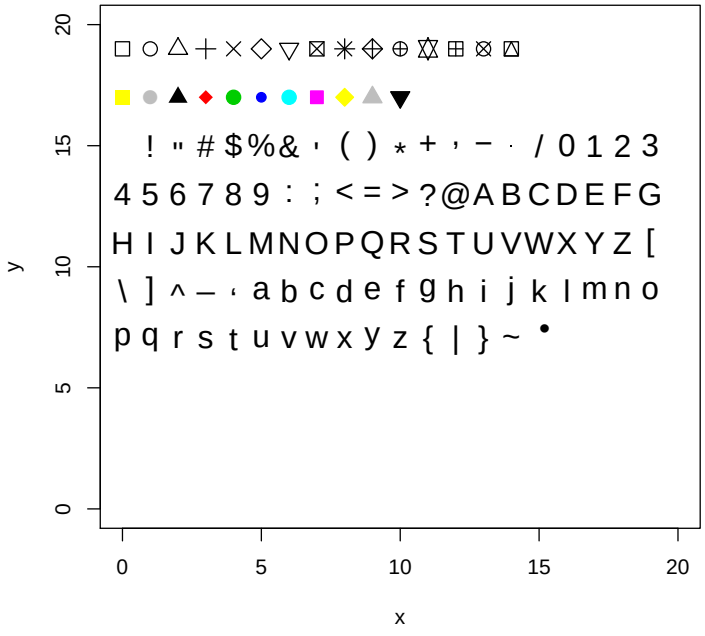

Рис. 7.12: Примеры различных символов, выводимых функцией **points()**

Функция arrows()

Функция

```
arrows(x0, y0, x1 = x0, y1 = y0, length = 0.25, angle = 30, code = 2,
       col = par("fg"), lty = par("lty"), lwd = par("lwd"),...)
```

позволяет рисовать стрелки на уже созданном графике.

Её аргументы:

- **x0** и **y0** — координаты точки, откуда стрела выходит (начала стрелы). Могут быть векторами.
- **x1** и **y1** — координаты точки, куда стрелка входит (конец стрелы). Могут быть векторами.
- **length** — длина стрелки в дюймах.
- **angle** — угол между древком стрелы и наконечником (в градусах).
- **code** — параметр, определяющий тип стрелы: **1** — острие стрелы в точке (x_1, y_1), **2** — острие стрелы в точке (x_0, y_0), **3** — острие стрелы в точках (x_1, y_1) и (x_0, y_0).
- **col**, **lty** и **lwd** — параметры, определяющие цвет стрелы, тип её древка и толщину стрелы.

Как рисовать стрелы показано в примере 52.

Функция segments()

Функция

```
segments(x0, y0, x1 = x0, y1 = y0,
         col = par("fg"), lty = par("lty"), lwd = par("lwd"),...)
```

соединяет прямыми отрезками заданные точки. Аргументы:

- **x0** и **y0** — начальные координаты — откуда отрезок выходит. Могут быть векторами.
- **x1** и **y1** — конечные координаты — куда входит входит. Могут быть векторами.
- **col**, **lty** и **lwd** — параметры, определяющие цвет стрелы,тип линии и толщину.

Пример 52. *Нарисуем на графике стрелы.*

```
x=1:12;y=c(12,10,11,9,6,7,5,8,1,3,2,4)
i = order(x,y); x <- x[i]; y <- y[i]
par(mfrow=c(1,2))
plot(x,y)
s = seq(length(x)-1)# one shorter than data
arrows(x[s], y[s], x[s+1], y[s+1], col= 1:3,angle=10)
```

и сегменты

```
plot(x,y)
## draw arrows from point to point :
s <- seq(length(x)-1)# one shorter than data
segments(x[s], y[s], x[s+1], y[s+1], col= 1:3)
```

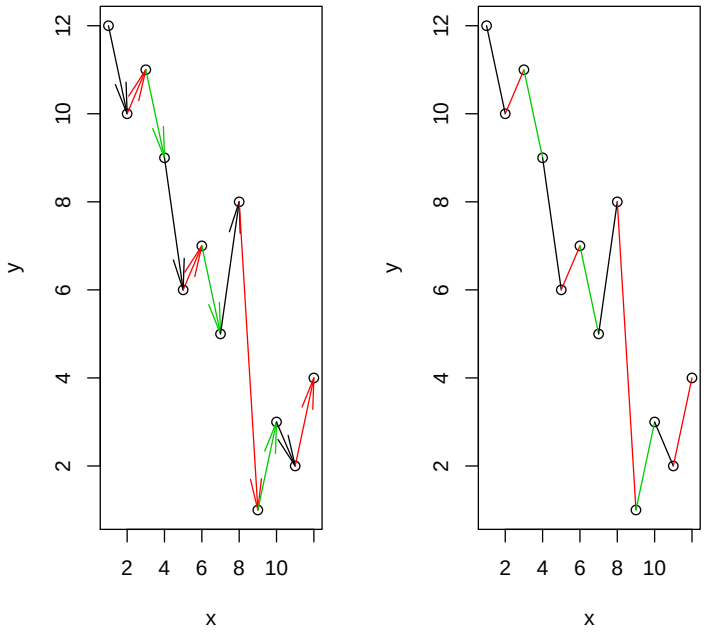

Рис. 7.13: Построение стрел и сегментов

Функция polygon()

Функция

```
polygon(x, y = NULL, density = NULL, angle = 45,
        border = NULL, col = NA, lty = par("lty"),
        ..., fillOddEven = FALSE)
```

строит многоугольник с заданными вершинами и заданными параметрами. Аргументы функции:

- **x** и **y** — координаты вершин многоугольника. Если **x** — список из двух компонент или матрица из двух столбцов, то достаточно только этого аргумента. Если среди координат есть хотя бы одно значение **NA**, то многоугольник разбивается на несколько.

- **density** — числовой аргумент — плотность штриховки многоугольника, число линий на один дюйм. По умолчанию **density = NULL** — отсутствие штриховки. Значение **density = 0** — не будет ни штриховки, ни закраски многоугольника. **density = NA** или отрицательное число — штриховки не будет, но можно закрашивать.

- **angle** — угол наклона (в градусах) линии штриховки.

- **col** — цвет внутренней части многоугольника.Если **col=NA**, то многоугольник не закрашивается, если не определена штриховка. Если **density** — положительное число, то аргумент **col** задаёт цвет штриховки. Аргумент **col** может быть вектором в случае построения нескольких многоугольников.

- **border** — цвет границы многоугольника. По умолчанию — **border=NULL** — цвет границы совпадает с цветом фона графического окна. Значение **border=NA** означает отсутствие границ. Значение **border=FALSE** эквивалентно **border=NA**, а **border=TRUE** — **border=NULL**. Аргумент **border** может быть вектором в случае построения нескольких многоугольников.

- **lty** — тип используемой линии.

- **fillOddEven** — аргумент, отвечающий за окрас и штриховку пересекающихся частей многоугольников.

Несколько вариантов построения многоугольников рассмотрено в примере 53.

Функция rect()

Частный случай многоугольника — прямоугольник — можно построить с помощью функции

```
rect(xleft, ybottom, xright, ytop, density = NULL, angle = 45,
     col = NA, border = NULL, lty = par("lty"), lwd = par("lwd"),...)
```

Так как большинство аргументов совпадает с аргументами функции **polygon()**, рассмотрим только отличные:

- **xleft** — левая координата по оси **0x** (скаляр или вектор).
- **ybottom** — нижняя координата по оси **0y** (скаляр или вектор).
- **xright** — правая координата по оси **0x** (скаляр или вектор).
- **ytop** — верхняя координата по оси **0y** (скаляр или вектор).

Пример 53. *Построим несколько многоугольников и прямоугольников. Сначала разобьём графическое окно на 4 части.*

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

В первой построим многоугольник с заданными вершинами.

```
plot(c(1,9), 1:2, type="n",xlab="", ylab="")
polygon(1:9, c(2,1,2,1,1,2,1,2,1),
        col=c("red", "blue"),
        border=c("green", "yellow"),
        lwd=3, lty=c("dashed", "solid"))
```

На втором представлен случай, когда одна из координат есть **NA**.

```
plot(c(1,9), 1:2, type="n",xlab="", ylab="")
polygon(1:9, c(2,1,2,1,NA,2,1,2,1),
        col=c("red", "blue"),
        border=c("green", "yellow"),
        lwd=3, lty=c("dashed", "solid"))
```

Третья часть — построение многоугольников.

```
plot(c(100, 250), c(300, 450), type = "n", xlab="", ylab="")
i = 4*(0:10)
rect(100+i, 300+i, 150+i, 380+i, col=rainbow(11, start=.7,end=.1))
```

На последней части — примеры закрашивания и штриховки.

```
plot(c(100, 200), c(300, 450), type= "n", xlab="", ylab="")
rect(100, 300, 125, 350) # transparent
rect(100, 400, 125, 450, col="green", border="blue")
rect(115, 375, 150, 425, col=par("bg"), border="transparent")
rect(150, 300, 175, 350, density=10, border="red")
rect(150, 400, 175, 450, density=30, col="blue",
     angle=-30, border="transparent")
par(op)
```

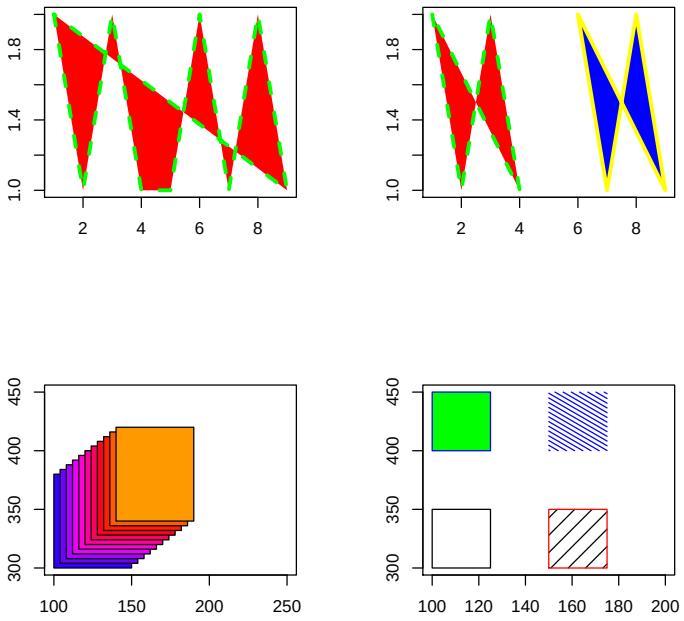

Рис. 7.14: Многоугольники

Функция symbols()

При помощи функции

```
symbols(x, y = NULL, circles, squares, rectangles, stars,
        thermometers, boxplots, inches = TRUE, add = FALSE,
        fg = par("col"), bg = NA,
        xlab = NULL, ylab = NULL, main = NULL,
        xlim = NULL, ylim = NULL, ...)
```

можно рисовать на графике круги, квадраты, прямоугольники, звёзды, термометры и **boxplot**.

Аргументы:

- **x, y** — координаты создаваемых фигур.
- **circles** — вектор радиусов создаваемых окружностей.
- **squares** — вектор длин сторон квадратов.
- **rectangles** — матрица из двух столбцов: первый столбец — длины оснований прямоугольников, второй — высоты прямоугольников.
- **stars** — матрица из трёх и более столбцов - длин лучей создаваемых звёзд.
- **thermometers** — матрица из трёх или четырёх столбцов. Первые два столбца — ширина и высота создаваемых символов. Если в матрице только три столбца, то третий столбец задаёт высоту закрашиваемой части (цветом параметра **fg**) термометра (доля от общей). Если в матрице четыре столбца, то третий и четвёртый столбцы задают высоты закрашиваемых частей (цветом параметра **fg**) термометров (снизу и сверху, соответственно) как доли общей высоты (оставшаяся часть закрашивается цветом **bg**).
- **boxplots** — матрица из пяти столбцов. Первые два столбца определяют ширину и высоту рамки основной части **boxplots**, третий и четвёртый — верхний и правый «ус», пятый столбец — задаёт медиану как долю высоты основной части **boxplots**.
- **inches** — логический или числовой аргумент — высота создаваемых символов. Если **inches=TRUE** (по умолчанию), то высота рисуемых фигур определяется таким образом, чтобы максимальная равнялась одному дюйму. Если **inches=FALSE**, то размеры создаваемых символов определяются по размерности оси **0x**. Если аргумент **inches** —число, то задаётся высота в дюймах.
- **add** — логический аргумент — добавить ли фигуры к уже существующему графику или построить новый.
- **fg** — цвет окрашивания элементов (в основном рамка).
- **bg** — цвет закрашивания фигур.

- **xlab, ylab, main, xlim, ylim** — аргументы, задаваемые в случае построения нового графика (**add=FALSE**).

Пример 54. *Изобразим на графике различные символы при помощи функции* **symbols()**.

```
x=1:100
y=1:100
plot(x,y,type='n')
x1=seq(10,90,by=10);x1
y1=rep(90,9);y1
A=seq(1,5,length=9);A
symbols(x1,y1,circles=A,bg=1:9,add=T,inches=FALSE)
B=seq(5,1,length=9)
symbols(x1,y1-10,squares=B,bg=1:9,add=T,inches=FALSE)
C=cbind(seq(1,5,length=9),seq(5,1,length=9))
symbols(x1,y1-20,rectangles=C,bg=1:9,add=T,inches=FALSE)
D=matrix(3,nrow=9,ncol=3)
symbols(x1,y1-30,stars=D,bg=1:9,add=T,inches=FALSE)
E=matrix(3,nrow=9,ncol=4)
symbols(x1,y1-40,stars=E,fg=1:9,add=T,inches=FALSE)
FF=matrix(3,nrow=9,ncol=5)
symbols(x1,y1-50,stars=FF,bg=1:9,add=T,inches=FALSE)
G=cbind(rep(3,9),rep(6,9),seq(0.1,0.9,by=0.1));G
symbols(x1,y1-60,thermometers=G,fg=1:9,add=T,inches=FALSE)
H=cbind(rep(3,9),rep(9,9),seq(0.1,0.45,length=9),seq(0.1,0.45,length=9))
symbols(x1,y1-70,thermometers=H,fg=1:9,bg=5,add=T,inches=FALSE)
```

7.2.2 Оформление графика — функции axis(), grid() и box()

Рассмотрим функции, позволяющие оформлять график — добавлять (изменять) оси, дополнительно выводить координатную сетку, рисовать рамку вокруг графика.

Функция axis()

Функция

```
axis(side, at = NULL, labels = TRUE, tick = TRUE, line = NA,
     pos = NA, outer = FALSE, font = NA, lty = "solid",
```

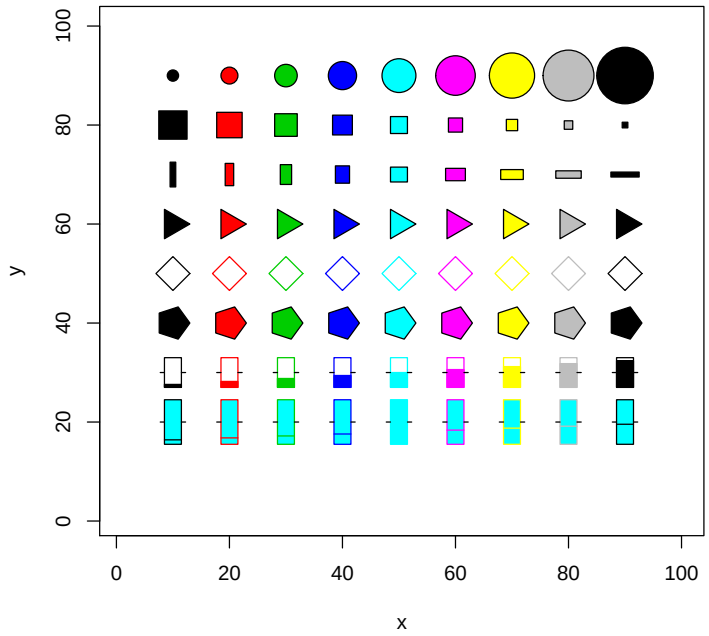

Рис. 7.15: Функция **symbols()**

```
lwd = 1, lwd.ticks = lwd, col = NULL, col.ticks = NULL,
hadj = NA, padj = NA, ...)
```

позволяет создавать оси по отдельности. Аргументы:

- **side** — число, определяющее где относительно графика будет построена ось. Значения: **1** — снизу , **2** — слева, **3** — сверху, **4** — справа.
- **at** — числовой вектор — деления на осях (от наименьшего до наибольшего значения). Значения **Inf**, **NaN** и **NA** игнорируются.
- **labels** — логический или символьный аргумент — метки созданных делений. Если **labels** = **TRUE**, то в качестве меток используются значения аргумента **at**. Символьный вектор задаёт названия меток делений, определённых аргументом **at** (поэтому длины векторов должны совпадать).
- **tick** — логический аргумент — должны ли рисоваться метки (штрихи) на самой оси.

- **line** — число линий, на которое построенная ось может выходить за область построения графика. Значение **line = NA**, означает, что ось строится только в области графика.

- **pos** — координата графического окна, до которой будет строиться ось. Если значение аргумента отлично от **NA**, то отключает аргумент **line**. Значение **line = NA**, означает, что ось строится только в области графика.

- **outer** — логический аргумент — должна ли ось выходить за стандартную область графика.

- **font** — семейство шрифтов, используемое для текста. Возможные значения: **1** — простой (по умолчанию), **2** — жирный; **3** — курсив и **4** — жирный курсив.

- **lty** — тип линии для оси и делений на оси. Возможные значения: **"solid"** (**1**) — сплошная линия; **"blank"** (**0**) — отсутствует линия; **"dashed"** (**2**) — штриховая линия; **"dotdash"** (**4**) — штрих-пунктир; **"dotted"** (**3**) — пунктирная линия; **"longdash"** (**5**) — длинный штрих; **"twodash"** (**6**) — двойной штрих (следует отметить, что должен быть задан параметр **type="l"**).

- **lwd** — толщина оси. Отрицательное или нулевое значение подавляет построение оси.

- **lwd.ticks** — толщина линий для делений на оси. Отрицательное или нулевое значение подавляет построение делений.

- **col** — цвет оси.

- **col.ticks** — цвет меток на оси.

- **hadj** — число или числовой вектор — расположение подписей к меткам на осях. Все подписи параллельны (горизонтальны) направлению чтения. Возможные значения: **0** — подписи ставятся слева от метки, **0.5** — по центру метки, **1** — справа от метки.

- **padj** — число или числовой вектор — расположение подписей к меткам на осях. Все подписи располагаются перпендикулярно направлению чтения. Возможные значения: **0** — название справа или выше метки, **1** — название слева или ниже метки.

Различные варианты задания осей приведены в примере 55 и рис.7.16.

Функция grid()

Функция

```
grid(nx = NULL, ny = nx, col = "lightgray", lty = "dotted",
     lwd = par("lwd"), equilogs = TRUE)
```

задаёт прямоугольную сетку на графике. Аргументы функции:

- **nx** и **ny** — количество ячеек сетки по осям **0x** и **0y**, соответственно. Если **nx** = **NULL**, то сетка строится на основе делений оси. Если **nx** = **NA** или **ny** = **NA**, то сетка в соответствующем направлении не строится.
- **col** — цвет линий сетки.
- **lty** — тип линии сетки.
- **lwd** — толщина линий сетки.
- **equilogs** — логический аргумент. Используется, если оси логарифмические и сетка строится в привязке к осям.

Стоит заметить, что подобную сетку можно построить и при помощи функции **abline()** (см. раздел 7.2.1 данной главы).

Варианты различного задания сетки приведены в примере 55 и представлены на рис.7.16.

Функция box()

И последняя функция в данном разделе — функция

```
box(which = "plot", lty = "solid", ...)
```

строящая прямоугольник (рамку) вокруг графика (заданной области) нужного цвета и типа линии. Аргументы:

- **which** — символьный аргумент. Возможные значения: **'plot'** — рамка вокруг графика, **'figure'** — рамка вокруг всей графической области (окна или подокна), **'inner'** и **'outer'** — рамка по краю всего графического окна.
- **lty** — тип линии.
- **...** — дополнительные параметры — цвет рамки, её толщина.

Пример 55. *Приведём несколько примеров построения осей, задания сетки и рамки. Представим графическое окно в виде нескольких*

```
op=par(mfrow=c(3,2))
```

Первый вариант задания осей

```
plot(1:4, rnorm(4), axes = FALSE)
axis(1, at=1:4, labels=LETTERS[1:4])
axis(2)
```

Иной вариант:

```
plot(1:7, rnorm(7), axes=FALSE,
     type = "s", frame = FALSE, col = "red")
axis(4, col = "violet", col.axis="dark violet", lwd = 2)
axis(3, col = "gold", lty = 2, lwd = 0.5)
```

Ещё вариант

```
plot(1:4, rnorm(4), axes = FALSE)
axis(1, at=1:4, labels=LETTERS[1:4],col='darkblue',
col.ticks='violet',lwd.ticks=2)
axis(2,col='red')
```

Сетка только по оси **0y**.

```
plot(1:3)
grid(NA, 5, lwd = 2)
```

По обоим направлениям:

```
plot(cos,-pi,pi)
grid(7, 5, lwd = 1, lty = 1) # grid only in y-direction
```

Различные варианты построения рамки (последние два варианта строят рамку вокруг всего графического окна)

```
plot(1:7, abs(rnorm(7)), type = 'h', axes = FALSE)
axis(1, at = 1:7, labels = letters[1:7],col='blue')
box(which='plot',lty = 'solid', col = 'red')
box(which='inner',lty = 4, col = 'blue',lwd=2)
box(which='outer',lty = 5, col = 'gold',lwd=3)
box(which='figure',lty = 3, col = 'green')
```

Выход из данного режима графического окна

```
par(op)
```

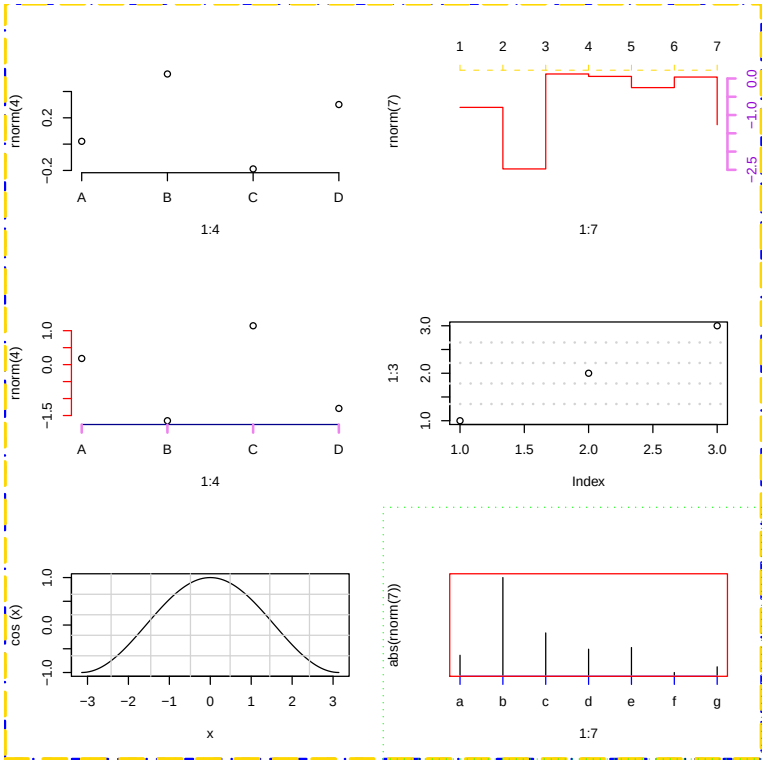

Рис. 7.16: Функции **axis**(), **grid**() и **box**()

7.2.3 Текст в графическом окне — функции expression(), text(), legend(), mtext() и title()

Эта часть главы посвящена выводу дополнительной текстовой информации к построенному графику — заголовки, легенды и текст.

Функция expression()

Сначала разберём функцию **expression**()[1], позволяющую создавать текст с использованием математических символов, греческих букв.

Обращение к функции:

```
expression(выражение)
```

С помощью

[1]Функция **expression**() создаёт особую структуру в **R** — «выражение», используемую при символьном дифференцировании в частности

```
demo(plotmath)
```

можно ознакомиться с записью различных математических символов. Так же они приведены на рис.7.17–7.21

Arithmetic Operators		Radicals	
x + y	$x + y$	sqrt(x)	$\sqrt{x}$
x − y	$x - y$	sqrt(x, y)	$\sqrt[y]{x}$
x * y	xy	Relations	
x/y	x/y	x == y	$x = y$
x %+−% y	$x \pm y$	x != y	$x \neq y$
x%/%y	$x \div y$	x < y	$x < y$
x %*% y	$x \times y$	x <= y	$x \leq y$
x %.% y	$x \cdot y$	x > y	$x > y$
−x	$-x$	x >= y	$x \geq y$
+x	$+x$	x %~~% y	$x \approx y$
Sub/Superscripts		x %=~% y	$x \cong y$
x[i]	x_i	x %==% y	$x \equiv y$
x^2	x^2	x %prop% y	$x \propto y$
Juxtaposition		Typeface	
x * y	xy	plain(x)	x
paste(x, y, z)	xyz	italic(x)	*x*
Lists		bold(x)	**x**
list(x, y, z)	x, y, z	bolditalic(x)	***x***
		underline(x)	$\underline{x}$

Рис. 7.17: Различные математические операторы и символы

Ellipsis		Arrows	
list(x[1], ..., x[n])	$x_1, \ldots, x_n$	x %<–>% y	$x \leftrightarrow y$
x[1] + ... + x[n]	$x_1 + \cdots + x_n$	x %–>% y	$x \rightarrow y$
list(x[1], cdots, x[n])	$x_1, \cdots, x_n$	x %<–% y	$x \leftarrow y$
x[1] + ldots + x[n]	$x_1 + \ldots + x_n$	x %up% y	$x \uparrow y$
Set Relations		x %down% y	$x \downarrow y$
x %subset% y	$x \subset y$	x %<=>% y	$x \Leftrightarrow y$
x %subseteq% y	$x \subseteq y$	x %=>% y	$x \Rightarrow y$
x %supset% y	$x \supset y$	x %<=% y	$x \Leftarrow y$
x %supseteq% y	$x \supseteq y$	x %dblup% y	$x \Uparrow y$
x %notsubset% y	$x \not\subset y$	x %dbldown% y	$x \Downarrow y$
x %in% y	$x \in y$	Symbolic Names	
x %notin% y	$x \notin y$	Alpha − Omega	$A - \Omega$
Accents		alpha − omega	$\alpha - \omega$
hat(x)	$\hat{x}$	phi1 + sigma1	$\varphi + \varsigma$
tilde(x)	$\tilde{x}$	Upsilon1	Υ
ring(x)	$\mathring{x}$	infinity	∞
bar(xy)	$\overline{xy}$	32 * degree	32°
widehat(xy)	$\widehat{xy}$	60 * minute	$60'$
widetilde(xy)	$\widetilde{xy}$	30 * second	$30''$

Рис. 7.18: Различные математические операторы и символы

Style	
displaystyle(x)	x
textstyle(x)	x
scriptstyle(x)	x
scriptscriptstyle(x)	x
Spacing	
x ~ ~y	$x\ \ y$

x + phantom(0) + y	$x + + y$
x + over(1, phantom(0))	$x + \frac{1}{}$
Fractions	
frac(x, y)	$\frac{x}{y}$
over(x, y)	$\frac{x}{y}$
atop(x, y)	$\begin{matrix} x \\ y \end{matrix}$

Рис. 7.19: Различные математические операторы и символы

Функция text()

Функция

Big Operators	
sum(x[i], i = 1, n)	$\sum_{1}^{n} x_i$
prod(plain(P)(X == x), x)	$\prod_{x} P(X = x)$
integral(f(x) * dx, a, b)	$\int_{a}^{b} f(x)dx$
union(A[i], i == 1, n)	$\bigcup_{i=1}^{n} A_i$
intersect(A[i], i == 1, n)	$\bigcap_{i=1}^{n} A_i$
lim(f(x), x %->% 0)	$\lim_{x \to 0} f(x)$
min(g(x), x >= 0)	$\min_{x \geq 0} g(x)$
inf(S)	$\inf S$
sup(S)	$\sup S$

Рис. 7.20: Различные математические операторы и символы

Grouping	
(x + y) * z	$(x+y)z$
x^y + z	$x^{y}+z$
x^(y + z)	$x^{(y+z)}$
x^{y + z}	x^{y+z}
group("(", list(a, b), "]")	$(a, b]$
bgroup("(", atop(x, y), ")")	$\binom{x}{y}$
group(lceil, x, rceil)	$\lceil x \rceil$
group(lfloor, x, rfloor)	$\lfloor x \rfloor$
group("|", x, "|")	$\|x\|$

Рис. 7.21: Различные математические операторы и символы

```
text (x, y = NULL, labels = seq_along(x), adj = NULL,
```

```
pos = NULL, offset = 0.5, vfont = NULL,
cex = 1, col = NULL, font = NULL, ...)
```

позволяет выводить дополнительную текстовую информацию к построенному графику. Аргументы функции:

- **x** и **y** — числовые вектора – координаты вывода текста (выводимый текст по умолчанию центрируется относительно заданной координаты). Желательно, чтобы длины векторов совпадали, в противном случае меньший вектор дублируется до длины большего.
- **labels** — либо символьный вектор, либо вектор, элементами которого являются **expression** — задают выводимый текст. Желательно, чтобы длина вектора **labels** совпадала с длинами векторов, задающих координаты.
- **adj** — одно или два значения из $[0; 1]$, определяющие положение выводимого текста относительно координат **x** и **y**.
- **pos** — альтернативное задание положения текста относительно координат. Возможные значения: **1** — ниже заданной координаты, **2** — слева от неё, **3** — сверху, **4** — справа. Подавляет аргумент **adj**.
- **offset** — данный аргумент задаёт смещение выводимого текста от заданной координаты (смещение в долях ширины символа).
- **vfont** — дополнительный аргумент, отвечает за тип шрифта семейства **Hershey**. По умолчанию значение **vfont = NULL** — стандартный шрифт.
- **cex** — числовой параметр — множитель стандартного размера символа. Значение этого аргумента, умноженное на стандартный размер символа **par('cex')**, определяет окончательный размер выводимых символов. Значения **cex = NULL** и **cex = NA** эквивалентны **cex = 1.0**.
- **col** — цвет текста.
- **font** — тип шрифта. Возможные значения: **1** — обычный текст (по умолчанию), **1** — жирный шрифт, **3** — курсив, **4** — жирный курсив.

Функция mtext()

Функция

```
mtext(text, side = 3, line = 0, outer = FALSE, at = NA,
      adj = NA, padj = NA, cex = NA, col = NA, font = NA, ...)
```

отвечает за вывод текста снизу или сверху, или слева, или справа от графика. Рассмотрим её аргументы (часть из них уже знакомы):

- **text** — символьный вектор или вектор, элементами которого являются **expression** — выводимый текст.
- **side** — сторона графика, относительно которой выводится текст. Значения: **1** — снизу, **2** — слева, **3** — сверху, **4** — справа.
- **line** — числовой параметр — количество линий, на которое надо отступить от границы области рисунка (отрицательное значение — переход внутрь области рисунка).
- **outer** — логический аргумент — может ли текст выходить за область графического окна.
- **at** — расположение текста в пользовательских координатах (от 0 до 1).
- **adj** — положение символьной строки, параллельной направления чтения. **adj=0** — текст находится снизу или слева, **adj=1** — справа или сверху.
- **padj** — положение символьной строки, перпендикулярной направления чтения. **padj=0** — текст находится справа или сверху, **padj=1** — слева или снизу.
- **cex** — размер выводимого текста (может быть вектором).
- **col** — цвет выводимого текста (может быть вектором).
- **font** — тип шрифта (может быть вектором).

Функция legend()

Функция

```
legend(x, y = NULL, legend, fill = NULL, col = par("col"),
       border="black", lty, lwd, pch,
       angle = 45, density = NULL, bty = "o", bg = par("bg"),
       box.lwd = par("lwd"), box.lty = par("lty"), box.col = par("fg"),
       pt.bg = NA, cex = 1, pt.cex = cex, pt.lwd = lwd,
       xjust = 0, yjust = 1, x.intersp = 1, y.intersp = 1,
       adj = c(0, 0.5), text.width = NULL, text.col = par("col"),
```

```
merge = do.lines && has.pch, trace = FALSE,
plot = TRUE, ncol = 1, horiz = FALSE, title = NULL,
inset = 0, xpd, title.col = text.col)
```

позволяет добавлять легенды к графику. Аргументы:

- **x** и **y** — координаты, задающие положение легенды. Если **x** и **y** — числа, то задана координата верхнего левого угла прямоугольника с легендой. Если **x** и **y** задают две точки, то тем самым определены противоположные углы легенды. Также положение легенды можно задать при помощи следующих символьных переменных: **'bottomright'** — в нижней правой части области построения графика, **'bottom'** — в нижней части, **'bottomleft'** — в нижней левой части, **'left'** — в левой части, **'topleft'** — в верхней левой части, **'top'** — в верхней части, **'topright'** — в верхней правой части, **'right'** — в правой части, **'center'** — в центральной части области графика.

- **legend** — текст выводимой легенды — вектор, символьный или **expression**.

- **fill** — задаёт цвет фона легенды или цвет штриховки легенды.

- **col** — вектор — задаёт цвета точек, соответствующие записям в аргументе **legend**.

- **border** — цвет рамки легенды. Используется, если задан аргумент **fill**.

- **lty** — тип линии в легенде.

- **lwd** — толщина линии легенды.

- **pch** — тип символов в легенде. Используется, если график построен при помощи символов. Подробно об этом параметре см.7.2.1 (функция **points()**) и пример 51.

- **angle** — наклон линий штриховки (в градусах).

- **density** — плотность штриховки — положительное число. Если значение **NULL**, **NA** или отрицательное, то область легенды закрашивается сплошным цветом.

- **bty** — символьный аргумент — рамка вокруг области с легендой. Два значения: **'o'** — рамка строится (по умолчанию) и **'n'** — рамка не строится.

- **bg** — фон области с легендой (используется, если **bty** не принял значение **'n'**).
- **box.lwd** — толщина рамки легенды (используется, если **bty** = **'o'**).
- **box.lty** — тип линии рамки легенды (используется, если **bty** = **'o'**).
- **box.col** —цвет рамки легенды (используется, если **bty** = **'o'**).
- **pt.bg** — цвет закраски символов (см.7.2.1, функция **points()**).
- **cex** — параметр изменения размера выводимого текста.
- **pt.cex** — параметр изменения размера используемых символов (см.7.2.1, функция **points()**).
- **pt.lwd** — толщина линий в выводимых символах (см.7.2.1, функция **points()**).
- **xjust** — числовой параметр — расположение легенды относительно координаты **x**. Возможные значения: **0** — слева от заданной точки, **0.5** — центрируется относительно **x**, **1** — справа от **x**.
- **yjust** — числовой параметр — расположение легенды относительно координаты **y**. Возможные значения: **0** — сверху от заданной точки, **0.5** — центрируется относительно **y**, **1** — снизу от **y**.
- **x.intersp** — расположение текста относительно координаты **x**.
- **y.intersp** — расположение текста относительно координаты **y**.
- **adj** — числовой вектор из двух элементов — расположение текста относительно координат **x** и **y**.
- **text.width** — ширина выводимого текста в пользовательских координатах (т.е. координатах **xy**). Только положительное значение.
- **text.col** — цвет текста легенды.
- **merge** — логический аргумент — соединяет точки и линии, но не закрашенные символы.
- **trace** — логический аргумент — показывает ход построения легенды (выводит используемые числовые значения).
- **plot** — логический — нужно ли строить легенду. При значении **FALSE** легенда не строится,.

- **ncol** — числовой аргумент — количество столбцов (колонок), в которых выводится легенда. По умолчанию **ncol = 1** (вертикальная легенда).
- **horiz** — логический аргумент — нужно ли легенду делать горизонтальной.
- **title** — символьный аргумент — заголовок легенды.
- **inset** — числовой аргумент — отступ от границы области графика. Значения от 0до 1 — доля от всей области графика.
- **xpd** — логический аргумент — может ли легенда выходить за область самого рисунка (графика).
- **title.col** — цвет заголовка легенды.

Функция title()

Наконец, последняя функция в этом разделе, это функция

```
title(main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
      line = NA, outer = FALSE, ...)
```

выводящая основной и дополнительный заголовки, а также названия осей. Аргументы:

- **main** — символьный аргумент — основной заголовок. Располагается над графиком.
- **sub** — символьный аргумент — дополнительный заголовок. Располагается под графиком.
- **xlab** — символьный аргумент — название оси **0x**.
- **ylab** — символьный аргумент — название оси **0y**.
- **line** — числовой аргумент — на сколько линий заголовок должен отступить от края области с рисунком.
- **outer** — логический аргумент — нужно ли заголовки выносить во внешнюю часть графического окна.
- **...** —

Пример 56. *В данном примере показано несколько вариантов вывода текста к графику, построения легенд и заголовков.*

```
op=par(mfrow=c(3,2))

plot(-1:1,-1:1, type = "n", xlab = "Re", ylab = "Im")
K <- 16
text(exp(1i * 2 * pi * (1:K) / K), col = 2)

plot(1:10, (-4:5)^2, main="Parabola", xlab="",ylab='')
mtext("10 of them")
for(s in 1:4)
    mtext(expression(y=x^2), line = (-1)^s,
          side=s, col=s, font=s, cex= (1+s)/2)

x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type = "l", ylim = c(-1.2, 1.8), col = 3, lty = 2)
points(x, cos(x), pch = 3, col = 4)
lines(x, tan(x), type = "b", lty = 1, pch = 4, col = 6)
legend(-1, 1.9, c("sin", "cos", "tan"), col = c(3,4,6),
       text.col = "green4", lty = c(2, -1, 1), pch = c(-1, 3, 4),
       merge = TRUE, bg = 'gray90')

x <- seq(-pi, pi, len = 65)
plot(x, sin(x), type="l", col = 2, xlab = expression(phi),
     ylab = expression(f(phi)))
abline(h=-1:1, v=pi/2*(-6:6), col="gray90")
lines(x, cos(x), col = 3, lty = 2)
ex.cs1 <- expression(plain(sin) * phi,  paste("cos", phi))
legend(-3, .9, ex.cs1, lty=1:2, col=2:3,    adj = c(0, .6))

x <- seq(-pi, pi, len = 65)
y=cos(x)
plot(x, y, type='n')
legend("bottomright", "(x,y)", pch=1, title="bottomright")
legend("bottom", "(x,y)", pch=1, title="bottom")
legend("bottomleft", "(x,y)", pch=1, title="bottomleft")
legend("topleft", "(x,y)", pch=1, title="topleft, inset = .05",
       inset = .05)
legend("topright", "(x,y)", pch=1, title="topright, inset = .02",
       inset = .02)

x <- seq(-4, 4, len = 101)
```

```
y <-sin(x)
plot(x, y, type = "l", xaxt = "n", xlab='',ylab='')

title(main = expression(paste(plain(sin) * phi, "  and  ",
                              plain(cos) * phi)),
        ylab = expression("sin" * phi, "cos" * phi), # only 1st is taken
        xlab = expression(paste("Phase Angle ", phi)),
        col.main = "blue")
curve(cos,from=-4,to=4,add=T)
axis(1, at = c(-pi, -pi/2, 0, pi/2, pi),
     labels = expression(-pi, -pi/2, 0, pi/2, pi))
abline(h = 0, v = pi/2 * c(-1,1), lty = 2, lwd = .1, col = "gray70")

par(op)
```

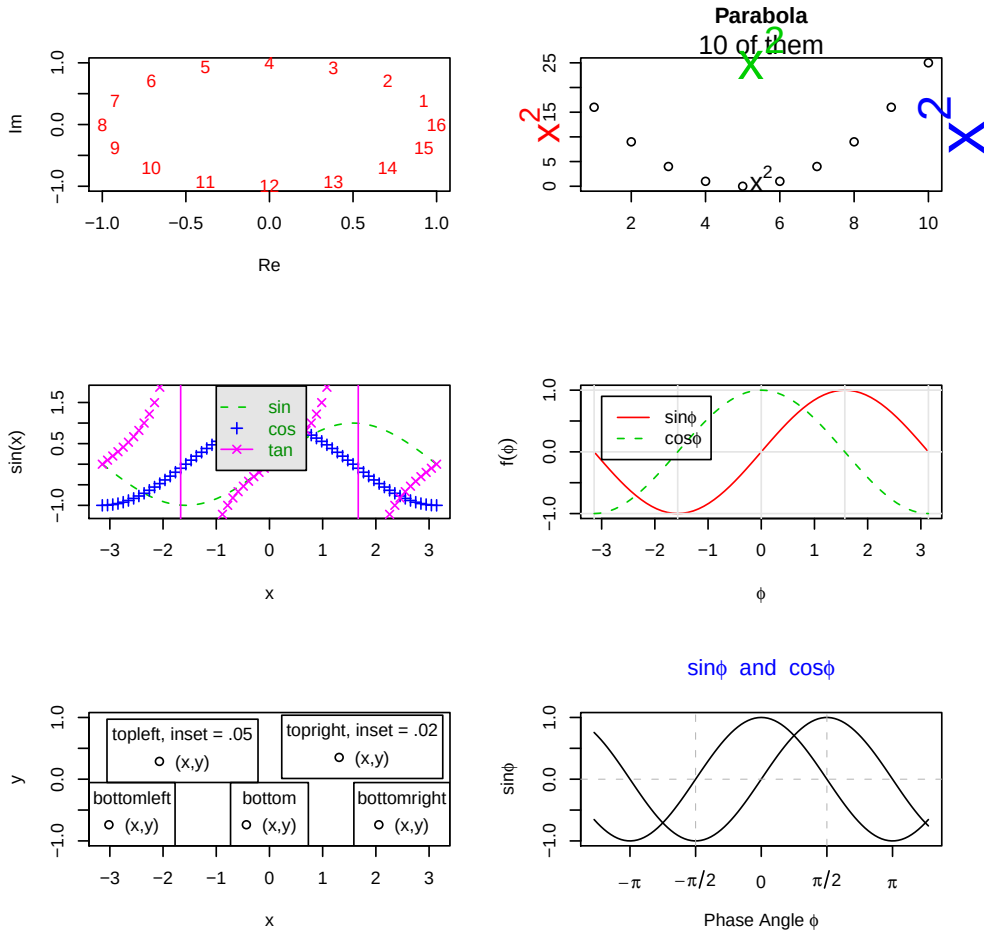

Рис. 7.22: Вывод текста на графике

Глава 8

Решение нелинейных уравнений и систем нелинейных уравнений. Интегрирование и дифференцирование. Экстремумы функций

В этой главе будут рассмотрены функции, позволяющие решать нелинейные уравнения и системы уравнения, вычислять интегралы и находить производные (в том числе и частные) функций, значения производных в заданных точках, а также экстремумы функций.

8.1 Решение нелинейных уравнений и систем нелинейных уравнений

Здесь будут рассмотрены три функции: функция **uniroot** базового пакета **base** и функции **uniroot.all** и **multiroot** пакета **rootSolve**.

8.1.1 Функция uniroot

```
uniroot(f, interval, ...,
lower = min(interval), upper = max(interval),
f.lower = f(lower, ...), f.upper = f(upper, ...),
tol = .Machine$double.eps^0.25, maxiter = 1000)
```

Аргументы:

- **f** — функция, нуль которой (т.е. корень) вычисляется. Отметим, что нуль функции **f** ищется **только** по её первому аргументу.
- **interval** — числовой вектор — интервал, на котором ищется корень (необходимо, чтобы значения функции на концах этого интервала имели разные знаки).
- **lower** и **upper** — альтернативное задание интервала поиска **interval** через его начало и конец.
- **f.lower** и **f.upper** — граничные значения функции (по умолчанию значения функции **f** на границах интервала поиска).
- **tol** — желаемая точность.
- **maxiter** — максимальное число итераций.
- **...** — дополнительные аргументы.

Решение считается найденным, либо если значение функции в найденной точке x^* равняется нулю ($f(x^*) == 0$), либо если изменение значения x^* на следующей итерации меньше заданной точности **tol**. Если достигнут максимум итераций, а решение не найдено, то выдаётся предупреждение.

Результатом функции **uniroot** является список из четырёх компонент: искомое решение x^* — *root*, значение функции в найденной точке $f(x^*)$ — *f.root*, число итераций *iter* и точность решения *estim.prec*. Если x^* совпадает с одним из концов заданного интервала поиска, то тогда значение *estim.prec* — **NA**.

Существенный недостаток функции **uniroot** — это то, что ищется только одно решение на заданном интервале. Если существует несколько нулей функции, то будет выводиться только первый найденный.

Пример 57. *Найдём решение уравнения* $5x^2 - 10x = 0$ *сначала на отрезке* $[1; 3]$*, а потом на* $[0; 3]$.

```
> f=function(x,a=5,b=10){f=a*x^2-b*x}
> uniroot(f,c(1,3))
$root
[1] 2.000000

$f.root
[1] -2.678252e-06
```

```
$iter
[1] 6

$estim.prec
[1] 6.535148e-05

> uniroot(f,c(0,3))
$root
[1] 0

$f.root
[1] 0

$iter
[1] 0

$estim.prec
[1] 0
```

Во втором случае в качестве решения находится только $x = 0$.

8.1.2 Функция uniroot.all

Функция **uniroot.all**[1] устраняет недостатки **uniroot**, позволяя вычислять несколько корней на заданном интервале.

```
uniroot.all(f, interval, lower=min(interval), upper=max(interval),
  tol=.Machine$double.eps^0.2, maxiter=1000, n=100, ...)
```

Отличие заключается в введении аргумента **n** — число подинтервалов, на которые делится исходный интервал, и на каждом из этих подинтервалов ищется нуль функции. Результат вызова функции – вектор с найденными решениями.

Пример 58. *Вернёмся к предыдущему примеру и найдём решение уравнения* $5x^2 - 10x = 0$ *на отрезке* $[0; 3]$.

```
> f=function(x,a=5,b=10){f=a*x^2-b*x}
> uniroot.all(f,c(0,3),n=10)
[1] 0.000000 2.000000
```

Найдены оба решения. Отметим, что если задать слишком большое число подинтервалов поиска, то возможны погрешности в решении.

[1] Пакет **rootSolve**

```
> uniroot.all(f,c(0,3))
[1] 0.000000 1.999999
```

Здесь $n = 100$ *(по умолчанию).*

8.1.3 Функция multiroot

Функция **multiroot** позволяет найти решение (если оно существует) системы из n уравнений с n неизвестными.

```
multiroot(f, start, maxiter=100,
  rtol=1e-6, atol=1e-8, ctol=1e-8,
  useFortran=TRUE, positive=FALSE,
  jacfunc=NULL, jactype="fullint",
  verbose=FALSE, bandup=1, banddown=1, ...)
```

Рассмотрим только часть аргументов, необходимых для работы:

- **f** — функция, в которой задана система уравнений. Должна возвращать вектор той же длины, что и **start**.
- **start** — вектор начальных значений для искомых неизвестных.Если элементам **start** присвоены имена, то эти имена будут использоваться при выводе решения.
- **maxiter** — максимальное допустимое число итераций.
- **rtol** и **atol** — относительная и абсолютная погрешности — скаляр или вектор. Если вектора, то каждый их элемент задаёт погрешности при поиске соответствующей неизвестной.
- **ctol** — текущая погрешность — скаляр. Если между двумя итерациями изменение неизвестной меньше этого заданного параметра, то считается, что решение найдено.
- **positive** — логический аргумент. Если **positive=TRUE**, то предполагается, что решения системы — положительны.

Результатом вызова функции является список из четырёх компонент: *root* — решения системы, *f.root* — значения уравнений системы в найденных точках, *iter* — число итераций, *estim.precis* — точность полученного решения.

8.2 Интегрирование и дифференцирование. Экстремумы функций

В **R** реализовано несколько функций, позволяющих вычислять интегралы, находить производные как в символьном виде, так и значения производных в заданных точках, определять экстремумы функций, решать оптимизационные задачи.

8.2.1 Вычисление интегралов и производных от функций и значения производных в заданных точках

Пакет **R** позволяет вычислять интегралы по конечным или бесконечным пределам. Для этого используется функция

```
integrate(f, lower, upper, ..., subdivisions=100,
rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
stop.on.error = TRUE, keep.xy = FALSE, aux = NULL)
```

с аргументами:

- **f** — определённая в **R** функция, интегрируемая по первому аргументу. Результатом интегрирования функции должно быть конечное число, иначе выводится сообщение об ошибке.
- **lower** и **upper** — нижний и верхний пределы интегрирования. Могут быть бесконечными. Если известно, что в результате интегрирования на бесконечном (полу-бесконечном) интервале должно получиться конечное число, то в качестве пределов (предела) следует задавать **Inf**, а не большие числа (см. пример 59).
- **...** — дополнительные аргументы, относящиеся к **f**. Они должны располагаться в том же порядке, что и в задаваемой интегрируемой функции.
- **subdivisions** — максимальное число интервалов, на которое разбивается интервал интегрирования.
- **rel.tol** — требуемая относительная точность, не может быть меньше $0.5 \cdot 10^{-28}$.
- **abs.tol** — требуемая абсолютная точность.

- **stop.on.error** — логический аргумент. При возникновении ошибки прекращает вычисление, в противном случае выдаётся результат с предупреждениями.
- последние два аргумента: **keep.xy** и **aux**, — не используются, введены для совместимости с языком **S**.

Результатом работы будет список. состоящий из следующих полей:

- **value** — результат интегрирования.
- **abs.error** — оценка модуля абсолютной ошибки.
- **subdivisions** — число подинтервалов интегрирования.
- **message** — вывод сообщения **«OK»** либо символьная строка с сообщением об ошибке.

Рассмотрим несколько примеров вычисления интегралов.

Пример 59. *Вычислим вероятность попадания нормально распределённой случайной величины с параметрами* $\mu = 0$ *и* $\sigma = 1$ *на интервал* $[0; \infty)$.

```
> integrate(dnorm,0,Inf)
0.5 with absolute error < 4.7e-05
```

Теперь в качестве верхнего предела рассмотрим конечные значения: 2, 20, 200, 2.000 и 20.000.

```
> integrate(dnorm,0,2)
0.4772499 with absolute error < 5.3e-15
> integrate(dnorm,0,20)
0.5 with absolute error < 3.7e-05
> integrate(dnorm,0,200)
0.5 with absolute error < 1.6e-07
> integrate(dnorm,0,2000)
0.5 with absolute error < 4.4e-06
> integrate(dnorm,0,20000)
0 with absolute error < 0
```

Последний вариант (верхний предел интегрирования равен 20000) приводит к ошибке.

В качестве другого примера найдём интеграл от функции $f(x) = \frac{1}{(x+1)\sqrt{x}}$. *Предел интегрирования — от 0 до* ∞. *Сначала зададим саму подинтегральную функцию.*

```
integrand = function(x) {1/((x+1)*sqrt(x))}
```

Теперь вычисли интеграл

```
> integrate(integrand, lower = 0, upper = Inf)
3.141593 with absolute error < 2.7e-05
```

Наконец, в качестве последнего примера рассмотрим случай, когда у интегрируемой функции несколько параметров: $f(x) = \frac{1}{a(1+(x-b)^2)}$, *где* $a = 2, b = 2$.

```
> integrand = function(x,a,b) {1/(a*(1+(x-b)^2))}
> integrate(integrand, lower = 0, upper = Inf,a=2,b=2)
1.338973 with absolute error < 5.5e-08
```

Кроме того, в базовой установке **R** реализованы три функции, которые в символьном виде вычисляют производные (в том числе и частные) заданных выражений.

Вид функций:

```
D (expr, name)
deriv(expr, namevec, function.arg = NULL, tag = ".expr",
hessian = FALSE, ...)
deriv3(expr, namevec, function.arg = NULL, tag = ".expr",
hessian = TRUE, ...)
```

Функция **D()** позволяет вычислять производную функции по одному аргументу. Функции **deriv()** и **deriv3()** позволяют вычислить частные производные.

Аргументы:

- **expr** — либо выражение (**expression**) или (за исключением функции **D**) формула (**formula**).
- **name**, **namevec** — символьный вектор, задающий имена переменных (только одна переменная для **D()**), по которым берутся производные.
- **function.arg** — если данный аргумент определён и не равен **NULL** — символьный вектор аргументов результирующей функции или функция (с пустым телом), или логический аргумент **TRUE**, определяющий использование функции с аргументами, имена которых определены **namevec**.
- **tag** — символьный аргумент — префикс, используемый для обозначения созданных локальных переменных при выводе результата.

- **hessian** — логический аргумент — нужно ли вычислять вторые производные и должны ли они быть включены в выводимые результаты.

- **...** — дополнительные аргументы, определяемые используемыми методами.

Результатом работы функций, как было уже упомянуто выше, будут символьные выражения, значения которых могут быть найдены в конкретных точках.

Стоит отметить, что функция **D()** может применяться к своим результатам, т.е. можно вычислять производные (по одному аргументу) любого порядка.

Примеры работы с функциями.

Пример 60. *В качестве примера найдём первую и вторую производные функции* $f(x, y) = \sin(\cos(x + y^2))$ *по переменным* x *и* y*, а также вычислим значения найденных производных в точках* $x = \frac{\pi}{4}$ *и* $y = \frac{\pi}{6}$.

```
> trig = expression(sin(cos(x + y^2)))
> ( D.x = D(trig, "x") )
-(cos(cos(x + y^2)) * sin(x + y^2))
> ( D.y = D(trig, "y") )
-(cos(cos(x + y^2)) * (sin(x + y^2) * (2 * y)))
```

Вторые производные:

```
> ( D.x.x = D(D.x, "x") )
-(sin(cos(x + y^2)) * sin(x + y^2) * sin(x + y^2) + cos(cos(x +
    y^2)) * cos(x + y^2))
> ( D.x.y = D(D.x, "y") )
-(sin(cos(x + y^2)) * (sin(x + y^2) * (2 * y)) * sin(x + y^2) +
    cos(cos(x + y^2)) * (cos(x + y^2) * (2 * y)))
> ( D.y.y = D(D.y, "y") )
-(sin(cos(x + y^2)) * (sin(x + y^2) * (2 * y)) * (sin(x + y^2) *
    (2 * y)) + cos(cos(x + y^2)) * (cos(x + y^2) * (2 * y) *
    (2 * y) + sin(x + y^2) * 2))
> ( D.y.x = D(D.y, "y") )
-(sin(cos(x + y^2)) * (sin(x + y^2) * (2 * y)) * (sin(x + y^2) *
    (2 * y)) + cos(cos(x + y^2)) * (cos(x + y^2) * (2 * y) *
    (2 * y) + sin(x + y^2) * 2))
```

Теперь найдём значения найденных производных в заданных точках. Для этого воспользуемся функцией **eval()**. *Для этого надо присвоить неизвестным аргументам функции (в нашем случае — это x и y) конкретные значения, затем в качестве аргумента функции* **eval()** *задать имена полученных выражений.*

```
> x=pi/4; y=pi/6
> eval(D.x)
[1] -0.7698184
> eval(D.y)
[1] -0.806152
> eval(D.x.x)
[1] -0.7893344
> eval(D.x.y)
[1] -0.8265891
> eval(D.y.y)
[1] -2.405239
> eval(D.y.x)
[1] -2.405239
```

Теперь при помощи функций **deriv()** *и* **deriv3()** *вычислим частные производные.*

```
> deriv(trig, c("x","y"), func = TRUE)
function (x, y)
{
    .expr2 <- x + y^2
    .expr3 <- cos(.expr2)
    .expr5 <- cos(.expr3)
    .expr6 <- sin(.expr2)
    .value <- sin(.expr3)
    .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
        "y")))
    .grad[, "x"] <- -(.expr5 * .expr6)
    .grad[, "y"] <- -(.expr5 * (.expr6 * (2 * y)))
    attr(.value, "gradient") <- .grad
    .value
}
```

Получен градиент - частные производные, **.value** *— исходная (дифференцируемая) функция.*

```
> deriv3(trig, c("x","y"), func = TRUE)
function (x, y)
{
    .expr2 <- x + y^2
    .expr3 <- cos(.expr2)
    .expr4 <- sin(.expr3)
    .expr5 <- cos(.expr3)
    .expr6 <- sin(.expr2)
    .expr14 <- 2 * y
    .expr15 <- .expr6 * .expr14
    .expr16 <- .expr4 * .expr15
    .expr18 <- .expr3 * .expr14
    .value <- .expr4
    .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
        "y")))
    .hessian <- array(0, c(length(.value), 2L, 2L), list(NULL,
        c("x", "y"), c("x", "y")))
    .grad[, "x"] <- -(.expr5 * .expr6)
    .hessian[, "x", "x"] <- -(.expr4 * .expr6 * .expr6 + .expr5 *
        .expr3)
    .hessian[, "x", "y"] <- .hessian[, "y", "x"] <- -(.expr16 *
        .expr6 + .expr5 * .expr18)
    .grad[, "y"] <- -(.expr5 * .expr15)
    .hessian[, "y", "y"] <- -(.expr16 * .expr15 + .expr5 * (.expr18 *
        .expr14 + .expr6 * 2))
    attr(.value, "gradient") <- .grad
    attr(.value, "hessian") <- .hessian
    .value
}
Получены частные производные и матрица вторых производных - матрица Гессе (гессиан)
```

Используя встроенные функции, создадим функцию для вычисления производных старшего порядка (по одной переменной).

```
DD = function(expr,name, order = 1) {
if(order < 1) stop("'order' must be >= 1")
if(order == 1) D(expr,name)
else DD(D(expr, name), name, order - 1)
}
```

Вызов функции и результат:

```
> DD(expression(sin(cos(x + y^2))),'x',3)
-((sin(cos(x + y^2)) * cos(x + y^2) - cos(cos(x + y^2)) * sin(x +
    y^2) * sin(x + y^2)) * sin(x + y^2) + sin(cos(x + y^2)) *
    sin(x + y^2) * cos(x + y^2) + (sin(cos(x + y^2)) * sin(x +
    y^2) * cos(x + y^2) - cos(cos(x + y^2)) * sin(x + y^2)))
```

8.2.2 Нахождение экстремумов функции. Решение задач оптимизации

В **R**, как правило, решается задача поиска минимума функции на некотором интервале с некоторыми условиями или без.

Функция **optim()** является основной в решении задач нахождения экстремумов.

```
optim(par, fn, gr = NULL, ...,
method = c("Nelder-Mead", "BFGS", "CG", "L-BFGS-B", "SANN"),
lower = -Inf, upper = Inf,
control = list(), hessian = FALSE)
```

Её аргументы:

- **par** — первоначальные значения параметров, относительно которых проводится оптимизация функции.
- **fn** — минимизируемая (или максимизируемая) функция, первый аргумент которой — вектор параметров, относительно которых проводится оптимизация. Результатом вызова функции должен быть скаляр.
- **gr** — функция, возвращающая градиент для методов **'BFGS'**, **'CG'** и **"L-BFGS-B'**.
- **...** — дополнительные аргументы, используемые **fn** и **gr**.
- **method** — используемый метод. Для функции **optim()** реализованы следующие методы:
  - **'Nelder-Mead'** — базовый метод. Устойчивый, используется только сама функция, но медленно сходится. Применим для недифференцируемых функций.
  - **'BFGS'** — квази-Ньютоновский метод, использующий функцию и её градиент.
  - **'CG'** — метод сопряжённых градиентов, менее устойчив по сравнению с предыдущими двумя, но также и менее ресурсозатратен.

  - **'L-BFGS-B'** — модификация метода **'BFGS'**, использующая ограничения на переменные.
  - **'SANN'** — стохастический оптимизационный метод. Использует только саму функцию, но медленно сходится. Применим для недифференцируемых функций. Сильно зависит от контрольных параметров.

- **lower** и **upper** — границы для переменных, используются только при методе **L-BFGS-B**.

- **control** — список, в который могут входить следующие управляющие параметры:

  - **trace** — неотрицательное целое число. Если положительное, то выводится информация о ходе оптимизации. Чем больше значение принимает контрольный параметр **trace**, тем более подробная информация выводится.
  - **fnscale** — масштабирующий параметр для оптимизируемой функции. Если принимает отрицательное значение, то решается задача максимизации. Таким образом, оптимизация происходит для **fn(par)/fnscale**.
  - **parscale** — числовой вектор, масштабирующий параметры оптимизации. Таким образом, оптимизация производится по **par/parscale** параметрам.
  - **ndeps** — вектор, определяющий размер шага оптимизации, по умолчанию — 10^{-3}.
  - **maxit** — максимальное число итераций. Значения по умолчанию: 100 — для методов **'BFGS'**, **'CG'** и **'L-BFGS-B'**, 500— **'Nelder-Mead'**, 10000 — для **'SANN'**.
  - **abstol** — абсолютная точность сходимости, применима только для неотрицательных функций.
  - **reltol** — относительная точность сходимости. Значение по умолчанию 10^{-8}.
  - **alpha** , **beta**, **gamma** — масштабирующие параметры для метода **'Nelder-Mead'**.
  - **REPORT** — частота выводимых сообщений для методов **'BFGS'**, **'L-BFGS-B'** и **'SANN'**, если контрольный параметр **trace** положителен. По умолчанию одно сообщение выводится на каждые 10 итераций для методов **'BFGS'** и **'L-BFGS-B'** или на каждые 100 итераций для метода **'SANN'**.

- **type** — контрольный параметр для метода сопряжённых градиентов. Выбор разновидности метода. Значение 1 — вариант **Fletcher–Reeves**, 2 — **Polak–Ribiere**, 3 — **Beale–Sorenson**.
- **lmm** — целое число — вариант модификации метода bf'L-BFGS-B'. По умолчанию — значение 5.
- **factr** — числовой параметр, контролирующий сходимость bf'L-BFGS-B' метода. Метод сходится, если изменение оптимизируемой функции меньше либо равно значению машинной точности (10^{-15}), умноженному на **factr**. По умолчанию — 10^7.
- **pgtol** — параметр, управляющий сходимостью метода bf'L-BFGS-B'.
- **temp** и **tmax** — управляющие параметры для метода **'SANN'**.

.

- **hessian** — логический аргумент — нужно ли выводить численные значения матрицы вторых производных.

Функция **optim()** может использоваться несколько раз подряд (как относительно одного параметра, так и нескольких).

В результате работы функции **optim()** создаётся список со следующими полями:

- **par** — оптимальные найденные значения параметров (точка минимума или максимума).
- **value** — значение оптимизируемой функции в найденной точке.
- **counts** — целочисленный вектор из двух компонент, описывающий, сколько раз использовалась функция и её градиент при оптимизации.
- **convergence** — целое число - сообщение о типе завершения оптимизации:
  - 0 — удачное завершение (практически всегда для **'SANN'**).
  - 1 — достигнуто максимальное число итераций.
  - 10 — расхождение метода **'Nelder-Mead'**.
  - 51 — ошибка в методе bf'L-BFGS-B'.
  - 52 — ошибка в методе bf'L-BFGS-B'..
- **message** — дополнительно выводимая информация об оптимизации (либо **NULL**).

- **hessian** — матрица — оценка матрицы вторых производных в найденной точке (выводится только если аргумент **hessian** функции **optim()** принимает значение **TRUE**).

Пример 61. *В качестве примера рассмотрим функцию Розенберга* $f(x, y) = 100(y - x^2) + (1 - x)^2$ *и найдём ее минимум.*

Сначала зададим саму функцию ($x[1] = x, x[2] = y$)*:*

```
fr <- function(x) {
x1 <- x[1]
x2 <- x[2]
100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
```

Функция, вычисляющая частные производные (градиенты):

```
grr <- function(x) {
x1 <- x[1]
x2 <- x[2]
c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
200 * (x2 - x1 * x1))
}
```

Результаты поиска точки минимума для различных методов:

```
> optim(c(-1.2,1), fr)
$par
[1] 1.000260 1.000506
$value
[1] 8.825241e-08
$counts
function gradient
     195       NA
$convergence
[1] 0
$message
NULL

> optim(c(-1.2,1), fr, grr, method = "BFGS")
$par
[1] 1 1
$value
[1] 9.594955e-18
```

```
$counts
function gradient
     110       43
$convergence
[1] 0
$message
NULL

> optim(c(-1.2,1), fr, NULL, method = "BFGS", hessian = TRUE)
$par
[1] 0.9998044 0.9996084
$value
[1] 3.827383e-08
$counts
function gradient
     122       38
$convergence
[1] 0
$message
NULL
$hessian
          [,1]      [,2]
[1,]  801.6881 -399.9218
[2,] -399.9218  200.0000

> optim(c(-1.2,1), fr, grr, method = "CG")
$par
[1] -0.7648373  0.5927588
$value
[1] 3.106579
$counts
function gradient
     402      101
$convergence
[1] 1
$message
NULL

> optim(c(-1.2,1), fr, grr, method = "CG", control=list(type=2))
$par
```

```
[1] 1.018684 1.037829
$value
[1] 0.0003491976
$counts
function gradient
     389      101
$convergence
[1] 1
$message
NULL

> optim(c(-1.2,1), fr, grr, method = "L-BFGS-B")
$par
[1] 0.9999997 0.9999995
$value
[1] 2.267597e-13
$counts
function gradient
      47       47
$convergence
[1] 0
$message
[1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"

> optim(c(-1.2,1), fr, method = "SANN")
$par
[1] 0.9945856 0.9895046
$value
[1] 3.856351e-05
$counts
function gradient
   10000       NA
$convergence
[1] 0
$message
NULL
```

Как видно из приведённых результатов метод сопряжённых градиентов дал неверный результат.

Следующие две функции одинаковы и различаются только написанием. Позволяют определить точку минимума (максимума) функции на заданном

интервале.

```
optimize(f = , interval = , ..., lower = min(interval),
upper = max(interval), maximum = FALSE,
tol = .Machine$double.eps^0.25)

optimise(f = , interval = , ..., lower = min(interval),
upper = max(interval), maximum = FALSE,
```

Их аргументы:

- **f** — оптимизируемая функция. В зависимости от значения аргумента **maximum** ищется либо минимум, либо максимум функции **f**.
- **interval** — числовой вектор, задающий интервал, на котором ищется экстремум функции.
- **...** — дополнительные аргументы для **f**.
- **lower** — нижняя граница интервала оптимизации.
- **upper** — верхняя граница интервала оптимизации.
- **maximum** — логический аргумент. Нужно искать минимум (по умолчанию) или максимум функции?
- **tol** — желаемая точность.

Функции **optimize** и **optimise** применимы (только) для непрерывных функций. Если оптимизируемая функция унимодальна, то буде найден (при правильном задании интервала оптимизации) глобальный минимум (максимум), если же оптимизируемая функция не унимодальна, то, скорее всего, будет найден локальный минимум (максимум).

Результат вызова **optimize** или **optimise** — список из двух компонентов: **minimum** — найденной точки минимума (максимума) и значения функции в этой точке — **objective**.

Рассмотрим пример.

Пример 62. *Найдём минимум и максимум функции* $f(x) = \frac{(x-5)^2 \sin x}{(x+2)^2}$ *на интервалах* $[-1; 4]$, $[0; 3]$ *и* $[1, 3]$.

```
> f = function (x,a,b) {(x-a)^2*sin(x)/((x+b)^2)}
> xmin = optimize(f, c(-1, 4), tol = 0.0001, a = 5, b=2)
> xmin
$minimum
```

```
[1] 3.642265
$objective
[1] -0.02779574

> xmax = optimize(f, c(-1, 4), tol = 0.0001, a = 5, b=2, maximum=T)
> xmax
$maximum
[1] 0.691799
$objective
[1] 1.634088

> xmin = optimize(f, c(0, 3), tol = 0.0001, a = 5, b=2)
> xmin
$minimum
[1] 2.999958
$objective
[1] 0.02258725

> xmax = optimize(f, c(0, 3), tol = 0.0001, a = 5, b=2, maximum=T)
> xmax
$maximum
[1] 0.6917868
$objective
[1] 1.634088

> xmin = optimize(f, c(1, 3), tol = 0.0001, a = 5, b=2)
> xmin
$minimum
[1] 2.999952
$objective
[1] 0.02258839

> xmax = optimise(f, c(1, 3), tol = 0.0001, a = 5, b=2, maximum=T)
> xmax
$maximum
[1] 1.000048
$objective
[1] 1.495910
```

Функция **nlm()** находит минимум заданной функции f при помощи метода Ньютона. Используются частные производные и вторые производные. Вид

функции

```
nlm(f, p, ..., hessian = FALSE, typsize = rep(1, length(p)),
fscale = 1, print.level = 0, ndigit = 12, gradtol = 1e-6,
stepmax = max(1000 * sqrt(sum((p/typsize)^2)), 1000),
steptol = 1e-6, iterlim = 100, check.analyticals = TRUE)
```

и её аргументы:

- **f** — минимизируемая функция. Если среди атрибутов функции заданы градиент и гессиан, то они будут использованы при вычислении. Иначе будут использованы результаты численного дифференцирования. К примеру, **deriv** позволяет получить функцию с градиентом в качестве атрибута.
- **p** — начальные значения для аргументов, по которым ищется минимум.
- **...** — дополнительные для **f** аргументы.
- **hessian** — логический аргумент. Если значение **TRUE**, то выводится значение матрицы вторых производных в найденной точке минимума.
- **typsize** — оценка размера каждого параметра в точке минимума.
- **fscale** — оценка размерности **f** в точке минимума.
- **print.level** — числовой аргумент — определяет объём информации, выводимой в процессе минимизации.
  - 0 (значение по умолчанию) — информация не выводится,
  - 1 — данные о первом и последнем этапам минимизации выводятся.
  - 2 — полностью вся информация о процессе оптимизации выводится.
- **ndigit** — число значащих знаков для функции **f**.
- **gradtol** — положительное число, достигая которое, градиент считается равным нулю и процесс оптимизации прерывается.
- **stepmax** — положительное число — отвечает за шаг оптимизации.
- **steptol** — положительное число — минимальный допустимый шаг оптимизации.
- **iterlim** — максимальное число итераций.

- **check.analyticals** — логический аргумент — нужно ли сравнивать аналитические градиенты и вторые производные с результатами численного дифференцирования. Помогает выявить некорректно сформулированные аналитические градиенты и вторые производные.

Результатом работы функции **nlm()** является список со следующими полями:

- **minimum** — значение **f** в точке минимума.
- **estimate** — точка, в которой достигает своего минимума функция **f**.
- **gradient** — значение градиента в найденной точке минимума.
- **hessian** — значение матрицы вторых производных в найденной точке минимума.
- **code** — код завершения процесса оптимизации. 1 и 2 - найденные значения являются (скорее всего) решением, 3–5 — сообщения об ошибках.
- **iterations** — число выполненных итераций.

Рассмотрим пример.

Пример 63. *Снова найдём минимум функции* $f(x) = \frac{(x-5)^2 \sin x}{(x+2)^2}$ *на интервалах* $[-1; 4]$, $[0; 3]$ *и* $[1, 3]$.

```
> f = function (x,a,b) {(x-a)^2*sin(x)/((x+b)^2)}
> nlm(f,0,a=5,b=2)
$minimum
[1] -1.635157e+15
$estimate
[1] -2
$gradient
[1] 8.10962e+20
$code
[1] 2
$iterations
[1] 17
```

Поменяем начальное значение x

```
> f = function (x,a,b) {(x-a)^2*sin(x)/((x+b)^2)}
> nlm(f,4,a=5,b=2)
$minimum
```

```
[1] -0.02779574
$estimate
[1] 3.642274
$gradient
[1] 4.610341e-09
$code
[1] 1
$iterations
[1] 7
```

Как видно, меняя начальные значения, можно найти разные точки минимума.

Функция **nlminb()** используется в решения задач оптимизации (поиск минимума функции) при наличии (отсутствии) ограничений на параметры с помощью процедур **PORT** (подробная информация - см. http://netlib.bell-labs.com/cm/cs/cstr/153.pdf).

```
nlminb(start, objective, gradient = NULL, hessian = NULL, ...,
scale = 1, control = list(), lower = -Inf, upper = Inf)
```

Её аргументы:

- **start** — числовой вектор — начальные значения аргументов функции, относительно которых проводится оптимизация.
- **objective** — функция, минимум которой ищется. Первым аргументом функции должен быть вектор параметров, относительно которых проводится оптимизация. В качестве своего значения функция должна возвращать скаляр. (возможные значения — **Inf** и **NA**). Вспомогательные аргументы функции **objective** могут определены быть далее (см. **...**).
- **gradient** — дополнительный аргумент (функция), вычисляющий градиент функции.
- **hessian** — дополнительный аргумент, позволяющий вычислить значения вторых производных минимизируемой функции (тоже функция).
- **...** — дополнительные аргументы для минимизируемой функции **objective**.
- **scale** — масштабирующий параметр (см. документацию для **PORT** — http://netlib.bell-labs.com/cm/cs/cstr/153.pdf).
- **control** — список управляющий параметров:

- **eval.max** — максимальное допустимое число вызовов целевой функции. Значение по умолчанию — 200.
- **iter.max** — максимально допустимое число итераций. По умолчанию — 150.
- **trace** — вывод значения целевой функции и парметров оптимизации на каждой итерации. Значение **0** означает, что информация выводиться не будет.
- **abs.tol** – абсолютная точность. По умолчанию — 0. АЕсли целевая функция неотрицательная, то лучше использовать значение **1e-20**.
- **rel.tol** — относительная точность. По умолчанию – **1e-10**.
- **x.tol** — точность относительно аргументов. По у молчанию — **1.5e-8**.
- **step.min** — минимальный размер шага итерации. По у молчанию — **2.2e-14**.

- **lower** и **upper** — вектора — нижняя и верхняя границы для аргументов оптимизации.

Результатом работы функции является список, включающий в себя следующие поля:

- **par** — оптимальный набор аргументов функции (т.е. точка минимума).
- **objective** — значение функции в найденной точке минимума.
- **convergence** — код завершения работы функции. **0** — успешное завершение.
- **message** — строка с дополнительной информацией (либо **NULL**).
- **iterations** — число итераций. N
- **evaluations** — число вызовов целевой функции и функции градиентов.

Пример 64. *Снова найдём минимум функции* $f(x) = \frac{(x-5)^2 \sin x}{(x+2)^2}$ *на интервалах* $[-1; 4]$, $[0; 3]$ *и* $[1, 3]$.

```
> f = function (x,a,b) {(x-a)^2*sin(x)/((x+b)^2)}
> nlminb(4,f,a=5,b=2,)
$par
[1] 3.642276
$objective
```

```
[1] -0.02779574
$convergence
[1] 0
$message
[1] "relative convergence (4)"
$iterations
[1] 7
$evaluations
function gradient
       9       10
```

Зададим ограничения относительно x,а именно $x \geq 0$:

```
> nlminb(4,f,a=5,b=2,lower=0,upper=Inf)
$par
[1] 3.642276
$objective
[1] -0.02779574
$convergence
[1] 0
$message
[1] "relative convergence (4)"
$iterations
[1] 7
$evaluations
function gradient
       8       10
```

Последняя функция, которая будет рассмотрена в этом разделе — это функция **constrOptim**, для которой можно задавать ограничения в виде линейных неравенств.

```
constrOptim(theta, f, grad, ui, ci, mu = 1e-04, control = list(),
method = if(is.null(grad)) "Nelder-Mead" else "BFGS",
outer.iterations = 100, outer.eps = 1e-05, ...)
```

Её аргументы:

- **theta** — числовой вектор начальных значения аргументов оптимизируемой функции длины p (начальные значения должны находиться в области, определённой ограничениями в виде неравенств).
- **f** — минимизируемая функция .

- **grad** — градиент функции **f** (тоже функция) либо **NULL**.
- **ui** — матрица ограничений размера $k \times p$.
- **ci** — вектор ограничений длины k.
- **mu** — параметр отладки оптимизации.
- **control**, **method** и **hessian** — парметры, относящиеся к функции **optim**.
- **outer.iterations** — число итераций, совершенных алгоритмом оптимизации.
- **outer.eps** — неотрицательное число - относительная точность сходимости алгоритма оптимизации.
- **...** — другие аргументы, относящиеся к **f** и **grad**.

Область, в которой ищется точка минимума, определяется неравенством (матричная форма) $\mathbf{ui}\theta - \mathbf{ci} \geq \mathbf{0}$.

Результат работы функции - список, аналогичный создаваемому функцией **optim()**, но с двумя дополнительными компонентами: **barrier.value** (значение созданной специальной функции в найденной точке минимума) и **outer.iterations**.

Рассмотрим пример.

Пример 65. *Снова рассмотрим функцию Розенберга* $f(x, y) = 100(y - x^2) + (1 - x)^2$ *и найдём ее минимум.*

Сначала зададим саму функцию ($x[1] = x, x[2] = y$)*:*

```
fr <- function(x) {
x1 <- x[1]
x2 <- x[2]
100 * (x2 - x1 * x1)^2 + (1 - x1)^2
}
```

Функция, вычисляющая частные производные (градиенты):

```
grr <- function(x) {
x1 <- x[1]
x2 <- x[2]
c(-400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1),
200 * (x2 - x1 * x1))
}
```

Теперь найдём её минимум в области $\{x \leq 0.9, y - x > 0.1\}$ *с помощью функции* **constrOptim()**

```
> constrOptim(c(.5,0), fr, grr, ui=rbind(c(-1,0),c(1,-1)), ci=c(-0.9,0.1))
$par
[1] 0.8891335 0.7891335
$value
[1] 0.01249441
$counts
function gradient
     256       48
$convergence
[1] 0
$message
NULL
$outer.iterations
[1] 4
$barrier.value
[1] -7.399944e-05
```

А теперь минимум этой же функции, но уже с другими ограничениями $\{x \leq 1, y \leq 1\}$

```
> constrOptim(c(-1.2,0.9), fr, grr, ui=rbind(c(-1,0),c(0,-1)), ci=c(-1,-1))
$par
[1] 0.9999761 0.9999521
$value
[1] 5.734117e-10
$counts
function gradient
     297       94
$convergence
[1] 0
$message
NULL
$outer.iterations
[1] 12
$barrier.value
[1] -0.0001999195
```

Литература

[1] R Development Core Team (2009). **R: A language and environment for statistical computing.** R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

[2] Brian S. Everitt and Torsten Hothorn (2005). **A Handbook of Statistical Analyses Using R**, London and Erlangen, http://www.R-project.org.

[3] Michael J. Crawley (2007). **The R Book**, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester. ISBN-13: 978-0-470-51024-7.

[4] John M. Chambers (2008). **Software for Data Analysis. Programming with R**, Springer Science+Business Media, LLC, USA, ISBN: 978-0-387-75935-7

[5] Phil Spector (2008). **Data Manipulation with R**, Springer Science+Business Media, LLC, USA, ISBN 978-0-387-74730-9

[6] Yosef Cohen and Jeremiah Y. Cohen (2008). **Statistics and Data with R**, John Wiley & Sons Ltd,The Atrium, Southern Gate, Chichester, ISBN 978-0-470-75805-2

[7] J. H. Maindonald (2001). **Using R for Data Analysis and Graphics An Introduction.** Statistical Consulting Unit of the Graduate School, Australian National University.

[8] John Verzani. **simpleR — Using R for Introductory Statistics**, http://cran.R-project.org/other-docs.html.

[9] Jim Lemon. **Kickstarting R**, http://cran.R-project.org/other-docs.html.

[10] Золотых Н. Ю., Половинкин А.Н. (2007). **Машинное обучение. Лабораторный практикум**, http://www.uic.unn.ru/ zny/ml/

[11] Меретилов М.А. (2006). **Методические указания к лабораторным работам по курсу «Методы анализа данных»**, http://gis-lab.info/docs/r-metoda-2006.10.23.pdf

[12] Шипунов А.Б. **R - объектно-ориентированная статистическая среда**, http://herba.msu.ru/shipunov/software/r/r-ru.htm