МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Национальный исследовательский Нижегородский государственный
университет им. Н.И. Лобачевского

В.Н. Якимов

ОСНОВЫ АНАЛИЗА БИОМЕДИЦИНСКИХ И ЭКОЛОГИЧЕСКИХ ДАННЫХ В СРЕДЕ R
ЧАСТЬ 1

Учебное пособие

Рекомендовано ученым советом Института биологии и биомедицины для
студентов ННГУ, обучающихся по направлениям подготовки
06.03.01 «Биология», 05.03.06 «Экология и природопользование»,
30.05.01 «Медицинская биохимия», 30.05.02 «Медицинская биофизика»,
30.05.03 «Медицинская кибернетика»

Нижний Новгород
2019

УДК 57.087.1:519.2(075.8)
ББК 28с:22.172я73
Я 45

Я 45 Якимов В.Н. ОСНОВЫ АНАЛИЗА БИОМЕДИЦИНСКИХ И ЭКОЛОГИЧЕСКИХ ДАННЫХ В СРЕДЕ R. ЧАСТЬ 1: Учебное пособие – Н. Новгород: Нижегородский госуниверситет, 2019. – 97 с.

Рецензенты: д.б.н. **Шитиков В.К.**
д.б.н. **Воденеев В.А.**
д.ф.-м.н., д.б.н. **Иудин Д.И.**

Настоящее учебное пособие имеет целью ознакомление студентов с основами синтаксиса языка программирования R, а также с основами работы в интегрированной среде разработки RStudio; изучение основных понятий, принципов и методов анализа данных в биомедицинских и экологических исследованиях; приобретение практических навыков анализа данных в среде вычислений R.

Учебное пособие предназначено для студентов очной формы обучения, обучающихся по направлениям подготовки 06.03.01 «Биология», 05.03.06 «Экология и природопользование», 30.05.01 «Медицинская биохимия», 30.05.02 «Медицинская биофизика», 30.05.03 «Медицинская кибернетика».

Ответственный за выпуск:
председатель методической комиссии ИББМ ННГУ
к.б.н. Водеенеева Е.Л.

УДК 57.087.1:519.2(075.8)
ББК 28с:22.172я73

© Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского, 2019
© В.Н. Якимов, 2019

Оглавление

Введение 5
Глава 1. Технологии анализа данных 7
Глава 2. Основы работы с R и RStudio 13
2.1. Установка R 13
2.2. Установка RStudio 14
2.3. Основы интерфейса RStudio 14
2.4. R как калькулятор 15
2.5. Рабочее пространство и рабочая директория 20
2.6. Работа со скриптами 22
2.7. Функциональность R и пакеты 24
Глава 3. Типы данных 28
3.1. Числовые данные 28
3.2. Комплексные данные 30
3.3. Логические данные 30
3.4. Текстовые данные 32
3.5. Необработанные данные 32
3.6. Специальные типы 33
3.7. Приведение типов 34
Глава 4. Структуры данных 38
4.1. Вектор 38
4.2. Матрица 40
4.3. Массив 42
4.4. Список 43
4.5. Фрейм данных 44
4.6. Фактор 47
Глава 5. Обращение к данным 49
5.1. Индексирование векторов 49
5.2. Индексирование матриц и массивов 52
5.3. Индексирование списков 54
5.4. Индексирование фреймов 56
Глава 6. Управляющие структуры 59
6.1. Условный оператор if else 59
6.2. Цикл for 60
6.3. Цикл while 61
6.4. Бесконечный цикл repeat 62
Глава 7. Функции 64
7.1. Определение функции 64
7.2. Аргументы функции 64
7.3. Возвращаемое значение 67

Глава 8. Векторизованные вычисления 69
Глава 9. Манипулирование данными 73
9.1. Создание и удаление переменных 73
9.2. Сортировка и ранжирование 74
9.3. Формирование подвыборок 76
Глава 10. Основы графической системы 78
10.1. Основные графические подсистемы R 78
10.2. Функция plot() и ее основные аргументы 79
10.3. Аннотирование графиков и добавление элементов 81
10.4. Использование функции par() 83
10.5. Графические устройства 94

Введение

Настоящее учебное пособие предназначено для студентов, обучающихся по направлениям подготовки 06.03.01 «Биология», 05.03.06 «Экология и природопользование», 30.05.01 «Медицинская биохимия», 30.05.02 «Медицинская биофизика», 30.05.03 «Медицинская кибернетика». Пособие имеет целью ознакомление студентов с основами синтаксиса языка программирования R, а также с основными понятиями, принципами и методами анализа данных в биомедицинских и экологических исследованиях, которые изучаются в рамках дисциплин «Математические методы в биологии» (6 семестр), «Математические методы в экологии» (4-5 семестры), «Математические методы в биологии и медицине» (8 семестр). Указанные дисциплины читаются студентам разных направлений подготовки и разных курсов, они имеют свою предметную специфику, но их объединяет необходимость формирования практических навыков обработки, описания, анализа и визуализации данных научных исследований.

В настоящее время в сфере биомедицинских и экологических исследований стандартным инструментом анализа данных стал язык программирования R и среда разработки RStudio. R позволяет импортировать данные практически из любых источников, проводить их описание, систематизацию, выполнять анализ как стандратными, так и постоянно разрабатываеми новыми методами, формировать иллюстрации любой степени сложности.

С ростом популярности R в мире постоянно растет количество учебной литературы, посвященной этой среде. По нашим оценкам, библиография книг по R перевалила за тысячу наименований. Однако подавляющее большинство этой литературы опубликовано на английском языке и русскоязычным студентам практически недоступно. В конце 2000-х начали появляться первые учебные пособия и методические рекомендации на русском языке, выпускаемые университетами для собственных нужд (Савельев и др., 2007; Буховец и др., 2010; Зарядов, 2010). В 2012 году была издана книга А.Б. Шипунова и соавторов «Наглядная статистика. Используем R!», описывающая среду R и предоставляемые ею возможности. В 2014–2015 годах были изданы два пособия для самостоятельного изучения R (Кабаков Р.И. «R в действии. Анализ и визуализация данных на языке R»; Мастицкий С.Э., Шитиков В.К. «Статистический анализ и визуализация данных с помощью R»). Упомянем также специализированную монографию В.К. Шитикова и Г.С. Розенберга «Рандомизация и бутстреп: статистический анализ в биологии и экологии с использованием R». На этом перечень имеющейся на русском языке литературы

по R практически исчерпывается. Настоящее учебное пособие призвано в некоторой степени восполнить недостаток учебной литературы.

Учебное пособие состоит из двух частей. Первая часть представляет собой краткое, но по возможности полное описание языка программирования R, а также программной среды, в которой предстоит работать студентам в рамках освоения перечисленных выше дисциплин. Вторая часть посвящена описанию наиболее часто используемых на практике статистических методов анализа данных. Уровень изложения не предполагает знакомство читателя ни с анализом данных, ни с программированием (хотя таковое знакомство, несомненно, окажется полезным). Для иллюстрации различных методов анализа используются примеры из реальных биомедицинских и экологических исследований. В частности, для иллюстрации применения большей части статистических методов, изложенных во второй части настоящего пособия, используется набор данных по морфологической изменчивости медоносных пчел (*Apis mellifera* L.). Эти данные были любезно предоставлены сотрудником ННГУ, к.б.н. А.А. Брагазиным. Необходимые для работы с пособием данные можно загрузить с официальной интернет-страницы данного пособия[1]. Каждой главе соответсвует скрипт, содержащий все примеры кода, содержащиеся в тексте. Читателю рекомендуется работать параллельно с текстом и скриптом, выполняя все примеры на компьютере в R и внося изменения в примеры для понимания работы обсуждаемых в тексте конструкций языка программирования.

[1] http://eco.365site.ru/book/R/

Глава 1. Технологии анализа данных

В ходе научной деятельности порождается большое количество данных, требующих обработки и анализа. Независимо от формы исследования (контролируемый эксперимент, опрос, наблюдения в природе) первичная информация представляет собой набор переменных, описывающих множество объектов исследования. Сбор первичных данных – это только один из этапов, за которым обязательно следует этап их обработки и анализа. Довольно редко при взгляде на первичные данные можно сделать однозначные выводы. Есть ли связь между изучаемыми переменными? Можно ли с уверенностью говорить о действии изучаемого фактора или воздействия (например, метода лечения)? Существуют ли отличия между изучаемыми группами, и если существуют, то между какими именно? Можно ли построить систему классификации изучаемых объектов? Для решения вопросов такого рода привлекается математическая статистика – раздел математики, позволяющий по имеющимся данным построить теоретико-вероятностную модель изучаемого явления. Статистические методы позволяют давать количественную оценку изучаемых явлений, на основании которой принимаются либо отвергаются определенные гипотезы и делаются выводы.

В биологии, медицине и экологии вопросы корректного статистического анализа данных имеют особое значение в связи с высокой изменчивостью живых организмов и формируемых ими сообществ, а также условий окружающей среды. На фоне высокой изменчивости изучаемых объектов надежные выводы могут быть сделаны только на основе статистического анализа данных.

В структуре научного исследования можно условно выделить несколько типичных этапов, которые составляют своего рода «жизненный цикл» данных: сбор и фиксация, хранение, описание, статистический анализ, визуализация, архивирование. На разных этапах этого цикла применяются различные технические средства, призванные облегчить работу исследователя.

Для первичной фиксации данных могут быть использованы как традиционные нецифровые средства (записи в полевой или лабораторный дневник), так и компьютерные технологии (введение данных непосредственно в специализированные программы, запись результатов измерений с компьютеризированных срeств измерений). Хранение данных в современных условиях осуществляется в оцифрованном виде. Данные, производимые в результате научных исследований, обычно имеют табличную форму, что определяет формат их хранения. Наиболее распространенными вариантами являются табличный документ и база данных. Первый вариант разумно выбрать

для хранения относительно небольших наборов данных, второй – для хранения больших массивов информации. Базы данных создаются также в случае необходимости работать с системой сложно структурированных показателей (например, когда принципиально необходимо организовать данные в виде связанных таблиц). Вопросы, связанные с выбором, организацией и эксплуатацией баз данных мы оставляем за рамками настоящего пособия. Наш опыт показывает, что подавляющее большинство данных, фигурирующих в биомедицинских и экологических исследованиях, могут быть организованы в виде табличного документа.

Для работы с табличными документами служат программы, называемые электронными таблицами. Наиболее распространенной программой такого типа является Microsoft Excel. В качестве альтернативных вариантов упомянем Numbers (входит в пакет iWork, работающий под операционной системой Mac OS), а также Calc (программы с таким названием входят в свободно распространяемые пакеты LibreOffice и OpenOffice.org). Несомненным достоинством Microsoft Excel является его высокая функциональность и интеграция с другими продуктами семейства Microsoft Office. Очевидным недостатком является высокая стоимость этого продукта. Мы рекомендуем использовать Microsoft Excel для введения и хранения данных, а для их анализа и визуализации применять более специализированные программы.

Конечными форматами хранения табличных данных являются различные варианты текстовых файлов (.txt, .csv), а также форматы Microsoft (.xls и .xlsx). В текстовых файлах данные хранятся построчно, определение столбцов данных осуществляется либо по ширине (каждому столбцу соответствует строго определенное число символов), либо с использованием разделителей – символов, обозначающих переход к следующему столбцу (чаще всего используются пробелы, символы табуляции и запятые). Microsoft Excel позволяет легко конвертировать данные, хранящиеся во внутреннем формате .xls, в любые текстовые форматы с использованием функций экспорта.

В ходе исследования после формирования набора данных возникает необходимость их анализа. Виды анализа конкретных данных определяются целями и задачами выполняемой работы, структурой исследования, а также в некоторой степени структурой самих данных. Тем не менее, подавляющее большинство видов анализа относится к методам статистического анализа, позволяющим получать количественно обоснованные выводы об изучаемых системах или явлениях. Простейшие виды статистического анализа реализованы в Microsoft Excel в виде стандартных функций, а также в виде специальной надстройки «Пакет анализа». Из множества других распространенных

специализированных программных продуктов для статистического анализа данных упомянем StatSoft Statistica, IBM SPSS Statistics, StataCorp Stata. Программные продукты этого класса характеризуются наличием пользовательского интерфейса, то есть для спецификации и настройки необходимого анализа можно воспользоваться разветвленной системой контекстных меню и форм, которые заполняются вручную. Результаты анализа также выводятся в виде специальных форм и графиков. Недостатками программ такого типа являются высокая стоимость и ограниченность набора реализованных методов анализа. Если в программе отсутствует необходимый метод анализа, то добавить его могут только разработчики. Несмотря на то, что производители постоянно совершенствуют свои программные продукты, скорость внедрения современных методов анализа оставляет желать лучшего, к тому же существует тенденция к расширению функциональности путем создания специализированных модулей, которые реализуются за отдельную плату.

Другой класс программного обеспечения, который используется для анализа данных, – программные среды, которые объединяют в едином продукте среду разработки на высокоуровневом языке программирования, его интерпретатор, а также набор инструментов для удобного манипулирования данными и их визуализации. Анализ данных в таких средах осуществляется путем выполнения выражений, написанных на языке программирования, которые можно вводить вручную, но чаще всего удобнее сохранить в виде текстового файла («скрипта»), который можно выполнять поэтапно (построчно или блоками), либо запустить целиком (при этом выражения все равно будут выполняться строго последовательно). Наиболее распространенными продуктами этого класса являются MathWorks Matlab, GNU Octave, TIBCO Software S-PLUS и R. Отметим также, что в последние годы в целях анализа данных широко используется язык программирования общего назначения Python, а также относительно новый язык Julia.

Matlab (сокращение от **mat**rix **lab**oratory – матричная лаборатория) является универсальной вычислительной средой, предназначенной в первую очередь для матричных вычислений, и поэтому широко распространен как система для научных и инженерных приложений. Недостатками Matlab являются его высокая стоимость, а также относительно небольшой набор статистических методов, реализованных в пакете расширения Statistics Toolbox. Бесплатным аналогом Matlab, использующим одноименный язык программирования, является свободно распространяемая среда Octave.

S-PLUS представляет собой современную реализацию языка программирования S, разработанного в конце 1970-х – начале 1980-х годов в исследовательском центре Bell Laboratories (на тот момент – подразделение корпорации AT&T) под руководством Джона Чемберса (John Chambers) специально для проведения статистических вычислений. В самом синтаксисе этого языка заложено много специальных возможностей, облегчающих статистическое моделирование. В течение 1990-х годов S и его позднейшие реализации заняли доминирующее положение на рынке программного обеспечения для статистического анализа. На сегодняшний день коммерческие права на язык S принадлежат компании TIBCO Software Inc., которая и занимается выпуском и поддержкой среды S-PLUS. Главным и практически единственным недостатком S на протяжении всей его истории является платность.

Некоммерческим свободно распространяемым аналогом S является язык программирования и программная среда R, использованию которой для анализа данных и посвящено настоящее пособие. История проекта R стартует в начале 1990-х годов, когда двое молодых исследователей Росс Ихака (Ross Ihaka) и Роберт Джентлмен (Robert Gentleman) из Оклендского университета (Новая Зеландия) решили написать собственную реализацию языка S. Первая версия была представлена в 1993 году, а в 1995 году проект стал принципиально открытым и до сих пор распространяется под универсальной общедоступной лицензией (GNU GPL), подразумевающей неограниченную возможность использования, распространения и модификации. Постепенно R приобретал все большее распространение, пользователи выявляли недостатки, которые исправлялись разработчиками. С определенного момента поток предложений о доработке стал настолько плотным, что отцы-основатели проекта Ихака и Джентлмен не успевали его обрабатывать, и в результате в 1997 году была сформирована The R Core Group – группа разработчиков, занимающихся поддержкой и доработкой исходного кода R. В конечном итоге к этой группе присоединился и создатель языка S Джон Чемберс. В 2000 году была выпущена среда R версии 1.0.0. Разработка R ведется постоянно, новые версии выпускаются регулярно. На момент последней редакции настоящего пособия актуальной является версия 3.5.2, выпущенная 20 декабря 2018 года.

Синтаксис R как языка программирования соответствует синтаксису языка S, они полностью совместимы. Несмотря на то, что детали реализации и внутренней логики языков несколько отличаются, для конечного пользователя это значения практически не имеет. Функциональность R организована в виде блоков, которые называются пакетами (packages). Пакеты представляют собой

наборы функций, учебных наборов данных и соответствующей документации. Разработчиками пакетов могут быть как члены The R Core Group, так и рядовые пользователи R. Ввиду абсолютной открытости популярность R постоянно растет, он используется как инструмент анализа данных практически во всех ведущих мировых университетах и корпорациях. Вокруг R сформировалось сообщество пользователей и разработчиков, активно внедряющих современные разработки в сфере статистического анализа и моделирования. На сегодняшний день R стал de facto стандартом как в среде профессиональных статистиков, разрабатывающих методы анализа, так и в среде пользователей анализа, включая биологов, медиков и экологов. В научных публикациях считается хорошим тоном приложить скрипты, реализующие описываемые методы анализа, либо сразу реализовать их в виде пакета. Еще одним несомненным достоинством R является кросс-платформенность: существуют версии R, работающие во всех основных операционных системах (Windows, Mac OS, Linux, Unix), что позволяет разработчикам и пользователям не заботиться о деталях реализации для разных платформ.

Несмотря на практически неограниченные возможности, которые предоставляет R при анализе и визуализации данных, эта среда не лишена своих недостатков. R идеально подходит для анализа данных, которые помещаются в оперативную память компьютера. Если же анализируемые данные настолько масштабны, что не помещаются в оперативную память, для их обработки лучше воспользоваться специализированным программным обеспечением, работающим непосредственно с базами данных, в которых хранятся исходные материалы. В последние годы разработано множество пакетов для R, позволяющих работать практически с любыми базами данных, однако их использование сопряжено со значительными сложностями и может не являться оптимальным решением.

Главным же недостатком R является сложность его освоения. В основе использования среды R лежит одноименный алгоритмический язык, поэтому для полноценного использования R необходимо освоение принципов программирования хотя бы на минимальном уровне. Для пользователей, которые в подавляющем большинстве случаев используют программные продукты, снабженные графическим интерфейсом (GUI), необходимость собственноручного написания выражений для выполнения является крайне непривычной. Тем не менее, такой программный интерфейс предоставляет невообразимую свободу в манипулировании данными и в настройке различных видов анализа под каждый конкретный случай. Усилия, затраченные на освоение

простейших навыков программирования, окупятся сторицей при проведении анализа данных собственных исследований.

Глава 2. Основы работы с R и RStudio

Среда разработки R представляет собой свободно распространяемое средство анализа и визуализации данных, которое доступно для основных операционных систем. В настоящем пособии мы будем рассматривать работу с R в операционной системе Windows. Интерфейс базовой среды R отличается значительным минимализмом и включает несколько стандартных для Windows меню (Файл, Вид, Помощь…), окно консоли выражений оператора, простейшие текстовый редактор и редактор данных, а также окно графического устройства (для вывода графиков). Для более удобной работы со средой R мы рекомендуем использовать RStudio – интегрированную среду разработки, которая является своего рода надстройкой над базовой средой R. RStudio, аналогично R, является свободно распространяемым продуктом (лицензия AGPL v3), доступным для Windows, Mac Os и Linux.

2.1. Установка R

Главный сайт проекта R в сети Internet располагается по адресу http://www.r-project.org/. Здесь можно найти общую информацию о проекте, полную документацию, ответы на часто задаваемые вопросы, ссылки на различные R-ориентированные ресурсы и сетевые сообщества, а также дистрибутивы R и пакетов. Дистрибутивы и пакеты хранятся в CRAN – Comprehensive R Archive Network – всеобъемлющей сети архивов R. Эта сеть сайтов с одинаковым содержимым (зеркал) обеспечивает стабильное хранилище ресурсов и, если какой-либо из сайтов сети будет недоступен, то необходимые материалы легко можно скачать с любого другого зеркала. Основной адрес CRAN http://cran.r-project.org/. В разделе mirrors можно выбрать географически ближайшее зеркало. Главная страница CRAN на любом зеркале предлагает скачать и установить R: нужно пройти по ссылке Download R for Windows и на следующей странице выбрать версию дистрибутива base, после чего вы попадете на страницу с дистрибутивом актуальной версии R, с которой необходимо скачать исполняемый файл дистрибутива R-3.5.2-win.exe (номер версии актуален на момент последней редакции пособия). Дистрибутив необходимо запустить (например, двойным кликом по исполняемому файлу) и следовать стандартным для операционной системы Windows диалогам по установке. По умолчанию R будет установлен в папку C:\Program Files\R\, а ярлык R будет помещен на рабочий стол.

2.2. Установка RStudio

Главный сайт проекта RStudio находится по адресу http://www.rstudio.com/. На главной странице нужно выбрать RStudio в выпадающем списке Products, на следующей странице выбрать версию для отдельных компьютеров (RStudio Desktop) и кликнуть по кнопке Download RStudio Desktop, после чего выбрать версию дистрибутива для необходимой операционной системы (в нашем случае Windows) и скачать соответствующий файл актуальной версии (на момент написания пособия RStudio-1.1.463.exe). Дистрибутив необходимо запустить и следовать стандартным для операционной системы Windows диалогам по установке. По умолчанию R будет установлен в папку C:\Program Files\RStudio\, а соответствующий ярлык будет помещен на рабочий стол.

При использовании RStudio нет необходимости отдельно запускать базовую среду R. После открытия RStudio, она самостоятельно найдет необходимую версию R и активизирует ее, поэтому ярлыки базовой среды R можно удалить.

2.3. Основы интерфейса RStudio

RStudio запускается файлом rstudio.exe, который можно найти в подпапке bin той папки, куда была установлена программа. На этот файл указывает ярлык, помещаемый на рабочий стол. Основной интерфейс RStudio состоит из четырех блоков, которые располагаются под стандартными панелями меню и инструментов (рис. 1). В верхнем левом углу располагается панель источников, в которой происходит просмотр и редактирование текстовых файлов (скриптов, файлов документации и т.п.), а также таблиц данных. Если открыто несколько файлов и таблиц, то перемещаться между ними можно с помощью вкладок (аналогично вкладкам интернет-браузера). Если не открыто ни одного файла, а именно так будет при первом запуске RStudio, панель источников скрыта и всю левую половину занимает блок консоли. Консоль (Console) – это основной элемент взаимодействия R с пользователем; сюда вводятся выражения для исполнения, сюда же выводится результат выполнения (если он предусмотрен выражением). При запуске новой сессии в консоли выдается стандартный для R набор информации: текущая версия, лицензия, инструкции о том, как получить сведения о разработчиках, цитировании, помощи, а также о том, как завершить сессию. После этого стандартного набора следует приглашение: символ “>”. В одном блоке с консолью на дополнительной вкладке расположен текстовый терминал для ввода системных команд Windows.

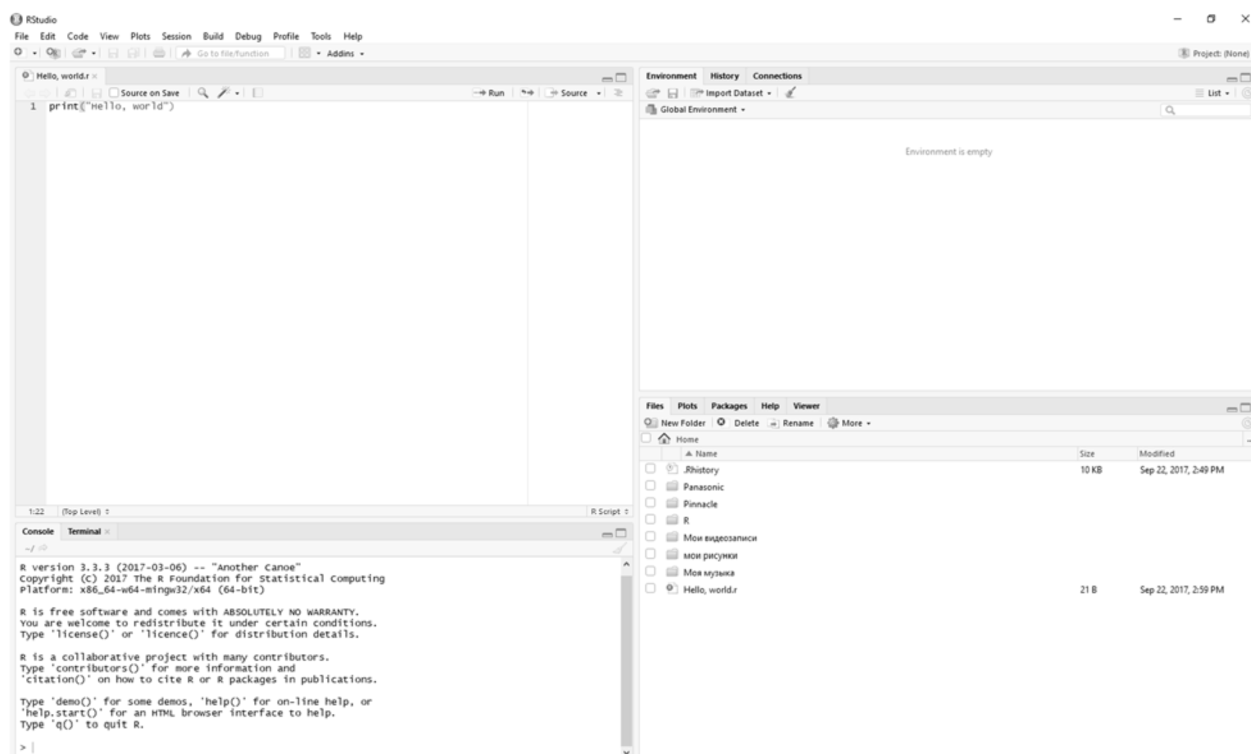

Рис. 1. Интерфейс RStudio

В верхнем правом углу располагается блок, содержащий на отдельных вкладках рабочее пространство (Environment), историю выполненных выражений (History), а также панель соединений с базами данных (Connections). При первом запуске все они будут пустыми. В нижнем левом углу располагается блок, содержащий на отдельных вкладках интерфейсы для взаимодействия с файловой системой (Files), графикой (Plots), пакетами (Packages), системой помощи (Help), а также внутренний браузер (Viewer) для просмотра локальных веб-страниц и приложений.

2.4. R как калькулятор

Итак, мы предполагаем, что R и RStudio успешно установлены и запущена новая сессия RStudio. Начнем знакомство с языком программирования R с ознакомления с его простейшими математическими возможностями. Попробуем воспользоваться консолью R в качестве калькулятора.

Комбинации констант, переменных, операторов и функций языка R, вводимые в консоль и выполняемые интерпретатором, называются *выражениями*. При работе с консолью выражения вводятся в строку после символа приглашения “>”, они будут выполнены после нажатия клавиши Enter.

В этом пособии будет приведено огромное количество таких выражений, которые можно ввести в консоль R (читателю настоятельно рекомендуется это сделать), а также соответствующих ответов системы. Мы будем следовать следующему соглашению: выражения на языке R будут приводиться моноширинным шрифтом после символа приглашения, соответствующий ответ системы будет приводиться в следующей строке после выражения.

Итак, попробуем ввести простейшее арифметическое выражение:

```
> 34 + 57
[1] 91
```

Это означает, что в консоль R было введено выражение “34 + 57” и система ответила, напечатав в консоли “[1] 91”. Чтобы воспроизвести этот результат, нужно набрать в консоли выражение “34 + 57” и нажать клавишу Enter, символ приглашения “>” вручную набирать не нужно.

Легко видеть, что R выполнил сложение двух чисел и в качестве ответа выдал результат сложения. “[1]” в ответе означает, что число 91 является первым (и в данном случае единственным) элементом результата вычислений. Аналогично можно выполнять другие арифметические операции:

```
> 13 - 7
[1] 6
> 15 * 6
[1] 90
> 45 / 4
[1] 11.25
```

Обратите внимание на последний пример с делением. *Оператором* деления в R служит косая черта “/”. Результатом деления оказалось нецелое число, причем при его отображении для отделения целой части от дробной используется точка.

Другие арифметические операции: возведение в степень (“^”), целочисленное деление (“%/%”), остаток от целочисленного деления (“%%”):

```
> 2 ^ 10
[1] 1024
> 13 %/% 5
[1] 2
> 13 %% 5
[1] 3
```

Арифметические операции легко комбинируются друг с другом, при этом соблюдается обычный порядок выполнения: сначала возведение в степень, затем умножение и деление, затем сложение и вычитание. Порядок операторов можно изменить, используя круглые скобки:

```
> 8 / 2 ^ 2 + 4 * 5
[1] 22
> 8 / 2 ^ (2 + 4) * 5
[1] 0.625
> ((15 + 5) / 10) ^ 2 * 3
[1] 12
```

При нескольких последовательных возведениях в степень эти операции выполняются справа налево:

```
> 3 ^ 2 ^ 2
[1] 81
```

В данном примере сначала 2 возводится в степень 2, и только после этого 3 возводится в степень 4 (результат первого возведения в степень).

Помимо базовых арифметических операторов в вычислениях часто используются простейшие математические преобразования, например извлечение корня или взятие модуля или логарифма. Такие преобразования реализованы в R в виде *функций*. Функция в R представляет собой набор выражений, которые необходимо выполнить. Каждая функция имеет собственное уникальное имя. Чтобы воспользоваться функцией, необходимо ввести ее имя, после которого должны следовать круглые скобки, в которые заключается перечень аргументов, передаваемых функции. В простейшем случае аргументом будет число либо объект, над которым необходимо произвести какую-либо операцию. Подробнее функции и их аргументы разбираются в главе 7. Например, для извлечения квадратного корня можно воспользоваться функцией `sqrt()`:

```
> sqrt(9)
[1] 3
> sqrt(5*6)
[1] 5.477226
```

В первом примере функции `sqrt()` передается число 9, а функция возвращает квадратный корень. Во втором примере функции передается результат выражения “`5*6`”, а сама функция вычисляет уже квадратный корень из 30.

Наиболее употребительные математические функции приведены в Таблице 1. Функции можно комбинировать с арифметическими операторами и между собой, можно также передавать функциям результаты вычисления других функций. При использовании арифметических операторов и функций приоритет выполнения всегда за функциями, то есть в первую очередь выполняются функции:

Таблица 1

Наиболее употребительные математические функции

Функция	Описание	Пример
`sqrt()`	Квадратный корень	`> sqrt(9)` `[1] 3`
`abs()`	Абсолютное значение	`> abs(-3)` `[1] 3`
`exp()`	Экспонента	`> exp(1)` `[1] 2.718282`
`log()`	Натуральный логарифм	`> log(10)` `[1] 2.302585`
`log2()`	Двоичный логарифм	`> log2(512)` `[1] 9`
`log10()`	Десятичный логарифм	`> log10(1000)` `[1] 3`
`cos()`	Косинус	`> cos(pi)` `[1] -1`
`sin()`	Синус	`> sin(pi/2)` `[1] 1`
`tan()`	Тангенс	`> tan(pi/4)` `[1] 1`
`acos()`	Арккосинус	`> acos(1)` `[1] 0`
`asin()`	Арксинус	`> acos(-1)` `[1] 3.141593`
`atan()`	Арктангенс	`> atan(2)` `[1] 1.107149`
`factorial()`	Факториал	`> factorial(5)` `[1] 120`
`lfactorial()`	Натуральный логарифм факториала	`> lfactorial(20)` `[1] 42.33562`
`ceiling()`	Округление в сторону большего целого	`> ceiling(2.11)` `[1] 3`
`floor()`	Округление в сторону меньшего целого	`> floor(2.99)` `[1] 2`
`round()`	Округление в сторону ближайшего среднего	`> round(2.49)` `[1] 2`
`trunc()`	Округление в сторону нуля	`> trunc(c(-2.9, 2.9))` `[1] -2  2`

```
> sin(0.2)^2 + cos(0.2)^2
[1] 1
> 1 / (2 * sqrt(2 * 7))
[1] 0.1336306
> log(exp(floor(abs(-20.79))))
[1] 20
```

В последнем примере сначала вычисляется абсолютное значение числа –20.79, которое передается в качестве аргумента функции `floor()`, округленное

вниз значение передается функции `exp()`, а результат потенцирования передается функции `log()`, которая возвращает итоговый результат.

В приведенных примерах мы использовали только числа, арифметические операторы и некоторые простейшие функции. Как и большинство других языков программирования, R позволяет присваивать значения переменным и обращаться к ним по именам. Основным оператором *присваивания* служит правая стрелка “<-”: комбинация символа “меньше” и минуса. Например, можно присвоить переменной `x` значение 5, а переменной `y` значение 8:

```
> x <- 5
> y <- 8
```

Обратите внимание, что после выполнения этих выражений R не выдал никакого ответа. Тем не менее, в рабочем пространстве появились две новые переменные, которым присвоены необходимые нам значения. После такого присвоения можно использовать символы “x” и “y” в выражениях:

```
> x^2 + y^2
[1] 89
> z <- sqrt(log(x) * log(y))
> z
[1] 1.829408
```

В первом примере вычисляется сумма квадратов значений переменных `x` и `y`, во втором примере переменной `z` присваивается значение квадратного корня из произведения логарифмов значений переменных `x` и `y`, в третьем примере никаких действий над переменной `z` не производится, в консоль выводится (то есть “печатается”) ее исходное значение. Когда необходимо сразу распечатать результат присвоения, можно заключить все выражение в круглые скобки:

```
> (w <- x * y)
[1] 40
```

Операторами присвоения также могут служить символ равенства “=”, а также левая стрелка “->”. Операторы присваивания допускают последовательное выполнение. Например, в следующем выражении сначала переменной `r` присваивается значение 5, после чего переменной `k` присваивается значение переменной `r`, в итоге обе переменные принимают значение 5:

```
> k = r = 5
```

При использовании таких конструкций необходимо помнить, что стрелки работают в том направлении, куда они указывают, а символ равенства “=” работает справа налево.

В некоторых случаях бывает удобно разместить несколько выражений в одной строке, например несколько присвоений и вычисление с выводом в

консоль. В таких случаях для разделения отдельных выражений служит точка с запятой:

```
> a <- 3; b <- 4; sqrt(a + b)
[1] 2.645751
```

R хранит историю выполняемых выражений (при работе в RStudio историю можно увидеть на вкладке History). Часто бывает удобно воспользоваться историей для повторного ввода ранее выполненного выражения. Если в консоли нажать стрелку вверх, появится последнее выполненное выражение, которое можно выполнить снова, нажав Enter. Если стрелку вверх нажать несколько раз, будут появляться предшествующие выражения из истории. Для перемещения между такими «историческими» выражениями можно также воспользоваться стрелкой вниз. Типичным случаем использования стрелок является исправление ошибок или изменение аргументов функций. Допустим, мы ввели сложное выражение, но при этом допустили некую ошибку (например, использовали лишнюю скобку при вызове функции). При попытке выполнить такое выражение мы получим сообщение об ошибке, например:

```
> sin(cos(sqrt(a/b))))
Error: unexpected ')' in "sin(cos(sqrt(a/b))))"
```

Чтобы не набирать вручную всё выражение, можно воспользоваться стрелкой вверх и оно появится в окне консоли. Остается только удалить лишнюю скобку и выполнить выражение:

```
> sin(cos(sqrt(a/b)))
[1] 0.6034809
```

2.5. Рабочее пространство и рабочая директория

Создаваемые в ходе работы с R переменные хранятся в рабочем пространстве (в конечном итоге – в оперативной памяти компьютера). Перечень объектов, которые находятся в рабочем пространстве, можно получить в любой момент с помощью функции `ls()`:

```
> ls()
[1] "k" "r" "w" "x" "y" "z"
```

При работе в RStudio переменные рабочего пространства можно видеть на вкладке Environment. Для удаления ненужных переменных из рабочего пространства служит функция `rm()`, которой в качестве аргументов нужно передать имена удаляемых переменных:

```
> rm(k, r, w)
> ls()
[1] "x" "y" "z"
```

Полностью очистить рабочее пространство можно следующим образом:

```
> rm(list = ls())
```

Приведенная конструкция передает функции `rm()` перечень переменных рабочего пространства, который создается вызовом функции `ls()`. В результате будут удалены все имеющиеся переменные. При работе в RStudio для очистки рабочего пространства можно воспользоваться кнопкой Clear (иконка в виде метлы) на вкладке Environment.

При старте новой сессии работы с R (например, при запуске RStudio) создается новое пустое рабочее пространство. Если мы хотим сохранить результаты работы текущей сессии, чтобы созданные переменные и функции были доступны в будущем, можно воспользоваться функцией `save.image()`. Применение этой функции создаст файл `.RData`, который будет сохранен в рабочей директории (см. ниже). Функции `save.image()` можно передать в качестве аргумента имя файла, в который нужно сохранить рабочее пространство, это имя может включать полный путь к директории, в которую нужно сохранить файл. Если путь не указан, файл будет сохранен в рабочей директории. Для загрузки ранее сохраненного рабочего пространства нужно воспользоваться функцией `load()`, которой в качестве аргумента необходимо передать имя загружаемого файла.

Рабочая директория – это та директория, с которой R будет работать в случае, если не указан полный путь к файлу. Если какой-либо функции, предназначенной для загрузки данных передать только имя файла, R попытается найти этот файл в рабочей директории, если же передать относительный путь, то R будет искать файл в подпапках рабочей директории. Аналогично и с функциями для записи файлов.

В операционной системе Windows XP рабочей директорией по умолчанию будет “C:\Documents and settings\UserName\”, в Windows 7 и Windows 10 – “C:\Users\UserName\Documents”, где UserName – имя учетной записи пользователя. Получить путь к текущей рабочей директории можно с помощью функции `getwd()`, изменить рабочую директорию можно с помощью функции `setwd()`, которой в качестве аргумента передается путь:

```
> getwd()
[1] "C:/Users/yakimov"
> setwd("D:/YandexDisk/Teaching/data analysis/scripts/")
> getwd()
[1] " D:/YandexDisk/Teaching/data analysis/scripts/"
```

При работе в RStudio выбрать и установить рабочую директорию можно через последовательность меню Session – Set Working Directory – Choose Directory (Ctrl+Shift+H), либо с помощью браузера файловой системы, располагающегося на вкладке Files (после выбора необходимой директории необходимо кликнуть по иконке More и выбрать пункт Set as Working Directory). В обоих случаях в консоли появится вызов функции `setwd()`, то есть RStudio избавляет пользователя от необходимости вручную вводить путь к директории.

При завершении сессии (например, при выходе из RStudio) R обычно спрашивает, нужно ли сохранить рабочее пространство и по умолчанию предлагает сохранить его в файл `.RData` текущей рабочей директории. Если при старте следующей сессии в рабочей директории R обнаружит файл `.RData` (это файл без имени, но с расширением), то он его загрузит, в итоге будет восстановлено рабочее пространство последней сохраненной сессии. Такое поведение можно изменить в настройках RStudio (меню Tools – Global Options).

В качестве практической рекомендации мы не советуем использовать функции сохранения и восстановления рабочего пространства. Содержимое рабочего пространства сложно документировать, а исходя из названий, размера и типов переменных бывает сложно понять, что они содержат и как они были созданы. Для воспроизведения ранее полученных результатов лучше воспользоваться скриптами.

2.6. Работа со скриптами

Скрипт представляет собой программу (последовательность выражений) на языке R, сохраненную в виде текстового файла с расширением .r, например “myScript.r”. В ходе работы с R скрипт можно выполнить целиком, а можно выполнять блоками, построчно или даже отдельными фрагментами.

В RStudio новый скрипт можно создать с помощью последовательности меню File – New File – R Script, либо нажав на иконку с зеленым плюсиком, которая располагается под меню File. После создания нового скрипта откроется панель источников, в которой на вкладке Untitled1 будет открыт текстовый редактор, в котором можно набирать и редактировать только что созданный файл. Прежде чем приступать к работе со скриптом, мы рекомендуем его сохранить с каким-нибудь осмысленным названием (по умолчанию созданный скрипт называется Untitled1 – “без названия 1”), для чего можно воспользоваться меню File – Save as. После сохранения вкладка панели источников поменяет название на имя нового файла.

Выражения набираются в текстовом редакторе. Встроенный в RStudio текстовый редактор обладает значительной функциональностью. Во-первых, он окрашивает элементы разного типа разными цветами (имена функций и переменных – черные, числовые константы – синие, текстовые константы – зеленые и т.д.). Во-вторых, здесь есть возможность автоматической индентации кода (расстановка отступов при использовании сложных управляющих конструкций, Ctrl-I) и комментирования (Ctrl+Shift+C). В-третьих, он обладает продвинутыми возможностями исполнения кода.

Если необходимо выполнить все содержимое файла целиком, можно воспользоваться кнопкой Source на панели источников. При этом будет вызвана функция `source()`, которой передан путь к скрипту. Все выражения, содержащиеся в скрипте, будут выполнены (созданы переменные, произведены вычисления, построены графики и т.п.), но в консоли не появятся ни сами выражения, ни результаты их выполнения (если таковые предусмотрены). Если необходимо увидеть выражения и результаты, нужно кликнуть по стрелке справа от кнопки Source и в выпадающем меню выбрать Source with Echo, что приведет к исполнению выражения вида `source("myScript.r", echo = TRUE)`. Теперь все выражения появятся в консоли, равно как и результаты их выполнения.

Если нужно выполнить не весь скрипт, а какую-то его часть, необходимые выражения нужно выделить (мышью либо комбинацией Shift + стрелки клавиатуры) и нажать кнопку Run на панели источников, либо воспользоваться комбинацией клавиш Ctrl+Enter. Выделять можно несколько строк (блок кода), отдельную строку, либо ее часть, в последнем случае будут выполнены только выделенные символы. Например, в сложном арифметическом выражении, содержащем несколько операций и функций, можно выделить отдельную операцию и выполнить ее. Это очень удобно при отладке кода и поиске ошибок.

Если нажать кнопку Run (либо Ctrl+Enter) в момент, когда в редакторе скрипта ничего не выделено, будет выполнена та строка кода, на которой в данный момент находится курсор. После этого курсор будет перемещен в следующую непустую строку скрипта. Этот способ очень удобен для просмотра и последовательного выполнения скрипта.

Важным элементом культуры программирования является комментирование кода. Комментарий – это текст в скрипте, который не предназначен для исполнения. Комментарии служат для описания того, что тот или иной фрагмент кода выполняет (выражения могут быть настолько изощренными, что разобраться в их смысле можно далеко не с первого взгляда), а также для отделения смысловых блоков кода, что улучшает читаемость скрипта. В определенных ситуациях можно воспользоваться комментированием,

чтобы “выключить” фрагмент кода, что бывает полезно при отладке. Символом комментария в языке R служит “#”: все, что содержится в строке справа от этого символа, является комментарием и выполнено не будет. Комментарий может начинаться в начале строки, а может находиться в строке после исполняемого выражения, например:

```
> # len - длина стороны квадрата
> len <- 45
> square <- len^2 # вычисляем площадь квадрата
```

При работе в R мы рекомендуем всегда писать скрипт. С точки зрения количества вводимых символов нет практически никакой разницы, будет ли выражение набрано непосредственно в консоли и затем нажата клавиша Enter, или оно будет набрано в скрипте и затем нажата комбинация Ctrl+Enter. При использовании скрипта вся последовательность выражений легко может быть воспроизведена, а в случае необходимости – модифицирована тем или иным образом (например, исправлена закравшаяся ошибка или подставлено другое значение параметра). Непосредственно в консоль имеет смысл вводить простые выражения при отладке кода (например, распечатать переменную, выяснить длину вектора или посмотреть структуру таблицы данных).

2.7. Функциональность R и пакеты

Функциональность R организована в виде библиотек, которые называются *пакетами* (packages). Пакет представляет собой коллекцию функций и учебных наборов данных, а также соответствующей документации. Базовая поставка среды R включает несколько пакетов, составляющих основу функциональности R – это пакеты base, methods, datasets, utils, grDevices, graphics, stats. Содержимое этих пакетов доступно всегда, они загружаются при старте новой сессии R по умолчанию. Помимо этих базовых пакетов существует огромное количество пакетов практически на все случаи жизни. Существуют пакеты, реализующие те или иные методы анализа (например, смешанные линейные модели, вейвлет-анализ), пакеты, обеспечивающие возможность работы с нестандартными источниками данных (например, с удаленными базами данных или с файлами в формате xml), пакеты, реализующие нестандартные графические системы, пакеты для анализа данных в какой-либо предметной области (например, филогенетические деревья) и т.д. и т.п. По мере развития сообщества разработчиков и пользователей R число доступных пакетов неуклонно растет. Быстрое внедрение новых методов анализа и их доступность в виде готовых к

использованию пакетов является одним из главных преимуществ R перед альтернативными системами анализа и обработки данных. Основным хранилищем пакетов служит CRAN (всеобъемлющая сеть архивов R). На момент написания пособия было доступно 13673 пакета.

Чтобы воспользоваться функциями пакета, его сначала необходимо установить, а затем загрузить в память компьютера. Установка пакета на компьютере пользователя осуществляется один раз. Повторная установка может потребоваться только для установки новой версии пакета (также как и сама среда R, пакеты имеют версии и многие из них активно обновляются).

Проще всего установить пакет на компьютере, имеющем доступ к сети Internet. В этом случае пакет устанавливается функцией `install.packages(“name”)`, где `name` – название устанавливаемого пакета. Можно также воспользоваться меню Tools – Install Packages…, где в выпадающем списке Install from необходимо выбрать пункт Repository (CRAN, CRANextra), а в поле Packages ввести имя пакета. После нажатия кнопки Install будет вызвана функция `install.packages()`. Например, установка пакета бутстреп-анализа boot может выглядеть так:

```
> install.packages("boot")
Installing package into ‘C:/Users/User/Documents/R/win-library/3.3’
(as ‘lib’ is unspecified)
Trying  URL  'https://cran.rstudio.com/bin/win/contrib/3.3/boot_1.3-
20.zip'
Content type 'application/zip' length 592553 bytes (578 KB)
downloaded 578 KB
package ‘boot’ successfully unpacked and MD5 sums checked
The downloaded binary packages are in
C:\Users\User\AppData\Local\Temp\RtmpgLgS2u\downloaded_packages
```

В третьей строке ответа системы указан адрес, с которого был загружен пакет, затем указан тип загруженного файла (zip), его размер, затем следуют сообщения об успешной загрузке и установке пакета, а также адрес директории, куда были распакованы необходимые для работы пакета файлы.

Пакеты часто используют функциональность друг друга. Так, функции одного пакета могут использовать функции другого пакета. В этом случае для работы первого пакета необходимо, чтобы был установлен и загружен второй пакет. Такая ситуация называется зависимостью пакетов. При автоматической установке пакета функцией `install.packages()` зависимости пакетов будут проверены и автоматически установлены все пакеты, от которых зависит устанавливаемый пакет, т.е. пользователю самому не нужно об этом беспокоиться.

Для установки пакета на компьютер без доступа к сети Internet необходимо перенести на него (например, с помощью USB-накопителя) архив пакета, загруженный с одного из зеркал CRAN на другом компьютере. Для загрузки архива пакета в боковом меню сайта нужно выбрать раздел Packages и перейти к таблице доступных пакетов (Table of available packages, sorted by name) и выбрать необходимый пакет. На странице с описанием пакета в разделе Downloads нужно выбрать и скачать zip-файл для Windows (любую из версий) с архивом пакета. Именно этот файл необходимо переместить на компьютер, где предполагается установка. Затем в RStudio в меню Tools – Install Packages… в выпадающем списке Install from необходимо выбрать пункт Package Archive File, после чего нажать кнопку Browse и выбрать файл с архивом пакета. После нажатия кнопки Install будет выполнено выражение вида `install.packages("путь/к/файлу/архива")` и пакет будет установлен. «Подводным камнем» при такой установке являются зависимости пакетов. Для полноценной установки пакета при скачивании архива с CRAN нужно обратить внимание на строчку Depends в описании пакета (обычно располагается сразу после номера версии), в которой содержатся названия и ссылки на все пакеты, от которых данный пакет зависит. Все эти пакеты также необходимо скачать и установить, не забывая при этом об их собственных зависимостях.

При работе в операционной системе Windows необходимо помнить, что распаковка пакетов обычно осуществляется в директорию, для модификации которой необходимы права администратора. Из-за отсутствия необходимых прав у текущего пользователя могут возникать проблемы с установкой пакетов. В этом случае можно воспользоваться запуском RStudio с правами администратора, для чего после нажатия на ярлык приложения правой кнопкой мыши в выпадающем контекстном меню нужно выбрать вариант «Запуск от имени администратора».

Для того чтобы содержимое установленного на компьютере пакета стало доступно для использования в текущей сессии, пакет необходимо загрузить с помощью функции `library()`, которой передается имя пакета, например:

```
> library(boot)
```

Обратите внимание, что название пакета не обязательно помещать в кавычки. Часто система при загрузке пакета будет выдавать предупреждения о несоответствии версии R и версии пакета. Это происходит вследствие того, что разработчики пакетов не всегда успевают за разработчиками базовой версии R, либо, наоборот, на вашем компьютере установлена старая версия R, тогда как текущая версия пакета собиралась уже для более актуальной версии R. Чаще всего такие несоответствия не имеют критического характера.

После загрузки пакета становится доступна вся его функциональность, можно использовать соответствующие функции и загружать тестовые наборы данных. Для загрузки набора данных из пакета нужно воспользоваться функцией `data()`, например:

```
> data(catsM)
> catsM[5,3]
[1] 7.6
```

После выполнения первого выражения в рабочее пространство будет загружен набор данных `catsM` из пакета boot, содержащий вес тела и сердца домашних кошек. В рабочем пространстве появится фрейм данных `catsM`, к которому можно обращаться (см. раздел 4.5).

В отличие от процедуры установки, которая выполняется один раз, загружать пакет для использования нужно в каждой новой сессии работы с R. В RStudio загружать пакеты можно также вручную на вкладке Packages. Здесь в алфавитном порядке отображаются все установленные на компьютере пакеты (приводится имя пакета, его описание и номер версии). Для загрузки пакета необходимо отметить галочкой поле слева от имени пакета.

Время от времени рекомендуется обновлять версии установленных пакетов на актуальные, то есть по сути переустанавливать пакеты. В RStudio это проще всего осуществить, нажав кнопку Update на вкладке Packages. При наличии доступа к Internet, все установленные пакеты будут проверены на наличие обновлений, а соответствующие обновления будут автоматически загружены и установлены.

Глава 3. Типы данных

Все данные, с которыми работает R, принадлежат пяти основным типам:
- числовые (numeric);
- комплексные (complex);
- текстовые (character);
- логические (logical);
- необработанные (raw).

Эти типы являются элементарными (атомарными, atomic) кирпичиками, из которых строится многообразие структур данных, которые анализируются в среде R. Все они, так или иначе, сводятся к комбинациям переменных этих основных типов.

Основной структурой хранения данных в R является вектор. Вектор представляет собой совокупность элементов одного и того же типа. Элементы вектора упорядочены, то есть следуют друг за другом и имеют порядковый номер, по которому к ним можно обращаться (подробнее см. гл. 5). Для объединения однотипных объектов в вектор служит функция `c()`, которая может принимать неограниченное число аргументов, отделяемых друг от друга запятыми, например:

```
> a <- c(2, 7, 4.3, -8.75, 0, 62, 0.33)
> a
[1]  2.00  7.00  4.30 -8.75  0.00 62.00 0.33
```

Здесь создан числовой вектор из 6 элементов.

3.1. Числовые данные

Существует два типа числовых данных: целочисленные (integer) и действительные (real). Целочисленные данные хранятся в 4-байтном формате, действительные – в 8-байтном формате с плавающей запятой.

Принадлежность переменной к тому или иному типу можно выяснить с помощью функций `typeof()` и `mode()`, которым передается имя переменной. Первая различает целочисленные и действительные переменные (и возвращает соответственно `"integer"` либо `"double"`), вторая для обоих возвращает значение `"numeric"`.

Основным типом числовых данных является действительный, поэтому в случае присвоения переменной целочисленного значения по умолчанию она все равно получит действительный тип:

```
> x <- 1
> typeof(x)
[1] "double"
```

В случае если необходимо создать именно целочисленную переменную, следует либо добавить суффикс `L`, либо воспользоваться оператором “:” (двоеточие), который предназначен для создания целочисленных последовательностей:

```
> y <- 1L
> typeof(y)
[1] "integer"
> z <- 1:5
> typeof(z)
[1] "integer"
```

При работе с действительными числами разделителем целой и дробной части в R служит точка, запятая используется для отделения аргументов функции и при индексировании. Соответственно, при попытке использовать запятую в качестве разделителя выражение окажется некорректным и будет выдано сообщение об ошибке:

```
> w <- 4,5
Error: unexpected ',' in "w <- 4,"
> (w <- 4.5)
[1] 4.5
```

Для создания векторов действительных чисел можно воспользоваться функциями `numeric()` и `double()`, а для создания целочисленного вектора – функцией `integer()`. Всем трем функциям необходимо передать число элементов создаваемого вектора. Во всех случаях будет выделен необходимый объем памяти для хранения данных, который по умолчанию будет заполнен нулями:

```
> v1 <- double(10)
> typeof(v1)
[1] "double"
> v1
 [1] 0 0 0 0 0 0 0 0 0 0
> v2 <- integer(10)
> typeof(v2)
[1] "integer"
> v2
 [1] 0 0 0 0 0 0 0 0 0 0
```

3.2. Комплексные данные

Данные комплексного типа создаются как сумма действительной и мнимой компонент, но при этом хранятся в едином комплексном формате. Мнимая часть вводится в виде числа, после которого ставится суффикс `i`, означающий мнимую единицу, между числом и суффиксом не допускается наличие пробела.

```
> (s1 <- 4 + 5i)
[1] 4+5i
> typeof(s1)
[1] "complex"
```

Другим способом создания комплексных переменных является функция `complex()`, с помощью которой можно создавать комплексные переменные на основе векторов действительных и мнимых значений, либо векторов модулей и аргументов комплексных чисел.

3.3. Логические данные

Логические данные имеют всего два возможных значения: `TRUE` (истина) и `FALSE` (ложь). Значения должны набираться заглавными буквами, иначе произойдет ошибка. Единственное допустимое сокращение: `T` для `TRUE`, `F` для `FALSE`:

```
> (t1 <- TRUE)
[1] TRUE
> (t2 <- F)
[1] FALSE
> (t3 <- True)
Error: object 'True' not found
```

Вектор логических данных может быть создан с помощью функции `logical()`, которая создаст вектор необходимой длины и заполнит его значениями `FALSE`:

```
> (t3 <- logical(5))
[1] FALSE FALSE FALSE FALSE FALSE
```

Логические данные имеют большое значение в программировании, поскольку часто используются для индексирования векторов (см. гл. 4). Чаще всего на практике логические данные создаются как результат выполнения логического выражения, например, оператора сравнения. Операторами сравнения являются “>” (больше), “>=” (больше либо равно), “<” (меньше), “<=” (меньше либо равно), “==” (тождественно) и “`!=`” (не тождественно). Операторы,

состоящие из двух символов, не допускают наличия пробела между ними. Результатом выполнения логического выражения всегда является логическое значение (ложь FALSE либо истина TRUE):

```
> 5 < 3
[1] FALSE
> 4 <= 4
[1] TRUE
> 8 == 5
[1] FALSE
> 7 != 10
[1] TRUE
```

Данные логического типа используются в выражениях, содержащих логические операции: отрицание (!), логическое И (&) и логическое ИЛИ (|). Отрицание меняет логическое значение на противоположное:

```
> !FALSE
[1] TRUE
> !TRUE
[1] FALSE
```

Логическое И возвращает TRUE, если оба компонента имеют значение TRUE, в противном случае возвращается FALSE:

```
> TRUE & TRUE
[1] TRUE
> FALSE & TRUE
[1] FALSE
```

Логическое ИЛИ возвращает TRUE, если хотя бы один из компонентов имеет значение TRUE, FALSE возвращается, когда оба компонента имеют значение FALSE:

```
> FALSE | FALSE
[1] FALSE
> TRUE | FALSE
[1] TRUE
```

Операторы сравнения и логические операции могут комбинироваться в едином выражении аналогично арифметическим операциям. При этом операции выполняются в следующем порядке: сравнительные операции, отрицание, логическое И, логическое ИЛИ. Одноранговые операции выполняются слева направо, порядок выполнения операций можно менять с использованием скобок:

```
> ! 2 != 3 & 3 >= 4
[1] FALSE
> ! (2 != 3 & 3 >= 4)
[1] TRUE
```

В обоих приведенных примерах в первую очередь производятся сравнения, результатом которых является логическое значение. Затем в первом примере

результат первого сравнения заменяется на противоположный, после чего выполняется логическое И. Во втором примере оператор отрицания выполняется уже после выполнения логического И.

3.4. Текстовые данные

Текстовые данные (типа Character) в R представляют собой строки, состоящие из множества символов. При вводе текстовых данных можно использовать одинарные либо двойные кавычки. При выводе в консоль R использует двойные кавычки:

```
> (u1 <- "Это текстовое значение")
[1] "Это текстовое значение"
> (u2 <- 'Это еще одно текстовое значение')
[1] "Это еще одно текстовое значение"
```

В приведенном примере созданы два вектора типа Character, каждый из которых состоит из одного элемента. В отличие от многих других языков программирования в R один элемент данных типа Character может состоять из множества символов (нет разделения на символы и строки) и будет восприниматься как единое целое при операциях сравнения, объединения и т.п. Для работы с текстовыми данными применяются специальные функции. Например, выяснить число символов можно с помощью функции `nchar()`, а обратиться к конкретным символам – с помощью функции `substr()`, которой нужно передать помимо имени переменной номера начального и конечного символов:

```
> nchar(u1)
[1] 22
> substr(u1, 15, 22)
[1] "значение"
```

3.5. Необработанные данные

Тип данных raw предназначен для непосредственной работы с байтами информации. Создать вектор необработанных данных можно с помощью функции `raw()`. Данные этого типа при распечатывании отображаются в шестнадцатеричном формате:

```
> (r <- raw(5))
[1] 00 00 00 00 00
```

Необходимость в переменных этого типа иногда возникает при написании функций для работы с нестандартными форматами файлов. В практике анализа данных этот тип почти никогда не встречается.

3.6. Специальные типы

Помимо описанных пяти атомарных типов в R существует несколько специальных типов данных, которые могут содержаться в векторах наряду с обычными данными.

В векторах любых атомарных типов могут встречаться так называемые отсутствующие значения, которые обозначаются символами `NA` (от англ. Not Available – не доступно). В практике анализа данных часто возникает ситуация, когда по каким-то причинам часть данных отсутствует для одного или нескольких объектов (например, вследствие потери части полевого дневника или невнимательного заполнения анкет добровольцев, участвовавших в проведении эксперимента). Чтобы иметь возможность правильно обрабатывать такую ситуацию и существует тип данных `NA`. Специальные функции для статистического анализа данных предусматривают различные варианты обработки отсутствующих значений. Характер такой обработки зависит от вида анализа, например при расчете описательных статистик отсутствующие значения лучше проигнорировать. `NA` – это зарезервированное в языке R сочетание символов, которое используется при вводе и выводе отсутствующих значений, например:

```
> (height <- c(183, 174, 169, NA, 167))
[1] 183 174 169  NA 167
> NA <- 56
Error in NA <- 56 : invalid (do_set) left-hand side to assignment
```

Первым выражением создан вектор `height`, причем четвертое значение отсутствует (возможно, исходная анкета четвертого добровольца с его морфометрическими параметрами была утеряна). Второе выражение пыталось использовать `NA` в качестве имени переменной, что в языке R недопустимо, поэтому было выдано сообщение об ошибке.

В векторах числовых значений могут встречаться значения `Inf`, `-Inf` и `NaN`. Эти значения могут появляться в результате выполнения арифметических операций. `Inf` и `-Inf` означают бесконечность (соответственно положительную и отрицательную, от англ. Infinity), они появляются в некоторых операциях переполнения:

```
> 5/0
[1] Inf
> 50^1000
[1] Inf
> log(0:3)
[1] -Inf 0.0000000 0.6931472 1.0986123
```

В первом примере бесконечность появилась в результате деления на ноль, во втором итогом возведения в степень будет настолько большое число, которое невозможно сохранить в обычном 8-битном формате, поэтому результат также представлен в форме `Inf`. В третьем примере сначала создается вектор целых чисел от 0 до 3 и каждое из них логарифмируется, а результат выводится в виде вектора. Первое значение итогового вектора представляет собой логарифм нуля, которому соответствует минус бесконечность. Обратите внимание, что `-Inf` присутствует в обычном числовом векторе.

Значения `NaN` образуются, когда результат вычислений не является числом (от англ. Not a Number), то есть когда результат не определен:

```
> 0/0
[1] NaN
> Inf - Inf
[1] NaN
```

`Inf` и `NaN`, также как и `NA` – это зарезервированные сочетания символов, использовать их в качестве имен переменных и функций нельзя.

Последний специальный тип объекта – `NULL`. Объект такого типа означает отсутствие объекта. `NULL` может возвращаться некоторыми функциями в исключительных ситуациях, когда результирующий объект не может быть определен.

3.7. Приведение типов

Когда функции передается аргумент не того типа, для которого функция предназначена, R попытается преобразовать тип передаваемых данных таким образом, чтобы функция сработала корректно (хотя, возможно, и не совсем так, хотел пользователь). Такое преобразование называется *приведением* типа (coercion). Общее правило приведения сводится к тому, что R преобразует данные от менее информативных типов к более информативным, то есть в направлении logical > integer > double > complex > character.

Простейшей ситуацией, в которой происходит приведение, является попытка объединить элементы данных разного типа в одном векторе. Вектор не

может хранить данные разного типа, поэтому осуществляется приведение к минимально необходимому типу, например:

```
> (v1 <- c(3, TRUE, FALSE))
[1] 3 1 0
> typeof(v1)
[1] "double"
> (v2 <- c(4.5, TRUE, 5+2i))
[1] 4.5+0i 1.0+0i 5.0+2i
> typeof(v2)
[1] "complex"
> (v3 <- c(FALSE, 5.68, "string", 4+6i))
[1] "FALSE"  "5.68"   "string" "4+6i"
> typeof(v3)
[1] "character"
```

В первом примере в один вектор собираются один числовой элемент и два логических, в результате логические значения приводятся к числовому типу, причем значение `TRUE` преобразуется в `1`, а значение `FALSE` в `0`. Вектор `v1` содержит данные типа double. Во втором примере в один вектор собираются данные числового, логического и комплексного типов, в результате все данные приводятся к комплексному типу. В третьем примере объединяются данные четырех основных типов, в итоге все они приводятся к текстовому формату. При выводе в консоль содержимого вектора `v3` может показаться, что первый элемент имеет значение `FALSE` и является логическим, а второй имеет числовое значение 5.68, однако обратите внимание на кавычки, которые указывают на то, что исходные данные были преобразованы в последовательность символов и получили текстовый тип.

Второй типичный случай, при котором происходит неявное приведение типа – использование операций с данными неверного типа (на самом деле это та же самая ситуация, что и с функциями, поскольку арифметические и логические операторы тоже являются функциями). Например, при попытке сложения логических значений, мы получим числовой результат:

```
> TRUE + FALSE + TRUE
[1] 2
```

Операция сложения не определена для логических значений, однако после преобразования их в числовую форму, можно получить осмысленный результат. Это свойство часто используется на практике, поскольку сумма значений логического вектора показывает, сколько значений `TRUE` в нем содержится. Аналогично, при использовании логических операторов с числовыми данными, произойдет обратное преобразование: нули преобразуются в `FALSE`, а любые ненулевые значения (включая отрицательные), преобразуются в `TRUE`:

```
> !8
[1] FALSE
> -2 & -8.5
[1] TRUE
> FALSE | 4
[1] TRUE
```

В описанных выше примерах приведение типа осуществляется автоматически. Приведение типа можно осуществлять принудительно с использований семейства функций `as.[type]()`, где вместо `[type]` нужно подставить название того типа, к которому нужно осуществить приведение.

```
> as.logical(v1)
[1]  TRUE  TRUE FALSE
> as.character(v2)
[1] "4.5+0i" "1+0i"   "5+2i"
> as.logical(v3)
[1] FALSE    NA    NA    NA
> as.numeric(v3)
[1]   NA 5.68   NA   NA
Warning message:
NAs introduced by coercion
```

В первом примере числовой вектор преобразуется в логический, во втором – комплексный вектор преобразуется в текстовый. В этих направлениях преобразование возможно всегда. В третьем примере текстовый вектор преобразуется в логический, при этом осмысленное преобразование возможно только в случае первого элемента, поскольку он имеет текстовое значение “`TRUE`”, три остальных элемента не имеют соответствующих значений, поэтому элементы преобразованного вектора получают значение `NA`. Аналогично происходит в четвертом примере, когда в числовое значение преобразуется только второй элемент текстового вектора, а остальные получают значение `NA`.

Для контроля за типами данных, которые содержатся в объектах, можно пользоваться не только функциями `typeof()` и `mode()`, но и семейством функций `is.[type]()`, где вместо [type] нужно подставить значение проверяемого типа. Функции этого семейства возвращают не название типа, а логическое значение, что во многих случаях удобнее.

```
> is.double(v1)
[1] TRUE
> is.character(v2)
[1] FALSE
> is.numeric(v3)
[1] FALSE
```

Эти функции возвращают одно логическое значение независимо от длины вектора, поскольку в векторе содержатся данные одного типа. Существуют аналогичные функции `is.na()`, `is.nan()`, `is.finite()` и `is.infinite()`, которые проверяют содержимое объекта на принадлежность к отсутствующим значениям `NA`, нечисловым значениям `NaN`, конечным либо бесконечным значениям (`Inf` и `-Inf`). Особенностью этих функций является то, что они возвращают вектор логических значений, в котором каждому элементу проверяемого вектора соответствует свой результат проверки. Это происходит потому, что специальные типы могут содержаться в векторах обычных значений. При таких проверках значения типа `NaN` считаются отсутствующими, но обратное неверно:

```
> (v4 <- c(4, 0, 3/0, NA))
[1]   4   0 Inf  NA
> is.na(v4)
[1] FALSE FALSE FALSE  TRUE
> is.finite(v4)
[1]  TRUE  TRUE FALSE FALSE
> is.infinite(v4)
[1] FALSE FALSE  TRUE FALSE
> (v5 <- c(NA, 5, -6, Inf/Inf))
[1]  NA   5  -6 NaN
> is.na(v5)
[1]  TRUE FALSE FALSE  TRUE
> is.nan(v5)
[1] FALSE FALSE FALSE  TRUE
```

Обратите также внимание на проверку вектора `v4` на конечность и бесконечность значений. Поскольку значение последнего элемента этого вектора отсутствует, оно не является ни конечным, ни бесконечным, в результате обе функции `is.finite()` и `is.infinite()` возвращают для этого элемента значение `FALSE`.

Глава 4. Структуры данных

Структуры данных определяют способ хранения упорядоченных наборов значений в оперативной памяти компьютера. В данной главе будут рассмотрены ключевые особенности различных структур данных и наиболее распространенные способы их создания. Способы обращения к элементам структур будут рассмотрены в гл. 5.

4.1. Вектор

Вектор представляет собой упорядоченный набор элементов. Упорядоченность заключается не в том, что элементы расположены по возрастанию, по алфавиту или еще в каком-нибудь осмысленном порядке, а просто в том, что элементы следуют один за другим и при этом имеют свою позицию, которой соответствует номер элемента. Вектор можно рассматривать как одномерную цепочку значений. Главным свойством вектора является то, что он обязательно содержит элементы одного и того же типа. Это могут быть числовые значения, текстовые, логические и т.д. Но при этом в векторе не могут содержаться элементы разных типов, при попытке объединения автоматически произойдет приведение типа (см. разд. 3.7).

Для создания векторов можно использовать огромное количество функций. В гл. 3 мы уже познакомились с универсальной функцией конкатенации `c()`, а также с рядом тип-специфических функций (`integer()`, `logical()` и т.п.). Для создания пустых векторов можно использовать функцию `vector()`, которая в качестве первого аргумента принимает обозначение необходимого типа данных, а второго – длину создаваемого вектора:

```
> (n1 <- vector("numeric", 3))
[1] 0 0 0
> typeof(n1)
[1] "double"
> (ch1 <- vector("character", 5))
[1] "" "" "" "" ""
> typeof(ch1)
[1] "character"
```

Обратите внимание, что во втором примере текстовый вектор заполняется значениями, не содержащими ни одного символа.

Векторы можно составлять из существующих векторов, а также добавлять к существующим векторам новые значения. Для этого используется функция конкатенации `c()`:

```
> n2 <- c(3, 5, 9)
> (n3 <- c(n2, n1, 100))
[1]   3   5   9   0   0   0 100
```

Элементы векторов объединяются в том порядке, в котором они переданы функции `c()`. Другой распространенный способ создания векторов – функция `rep()`, которая используется для повторения необходимых элементов. Типичные примеры использования:

```
> rep(n2, times = 3)
[1] 3 5 9 3 5 9 3 5 9
> rep(n2, each = 2)
[1] 3 3 5 5 9 9
> rep(n2, times = 2, each = 3)
 [1] 3 3 3 5 5 5 9 9 9 3 3 3 5 5 5 9 9 9
> rep(n2, length.out = 12)
 [1] 3 5 9 3 5 9 3 5 9 3 5 9
```

Прежде всего обратите внимание, что здесь мы используем ранее не встречавшийся способ передачи аргументов функции по имени (подробнее см. гл. 7), для чего используется конструкция `имя_аргумента = значение`. Передаваемые аргументы отделяются запятыми. В приведенных примерах первым неименованным аргументом функции `rep()` передается подлежащий повторению вектор `n2`. Аргумент `times` определяет количество последовательных повторений, поэтому в первом примере вектор `n2` повторен три раза. Аргумент `each` определяет количество повторений каждого из элементов вектора, поэтому во втором примере друг за другом следуют не копии вектора `n2`, а каждый из элементов вектора повторяется по 2 раза. Аргументы `times` и `each` могут использоваться вместе, когда необходимо получить вектор, состоящий из блоков повторяющихся значений, как в третьем примере. В четвертом примере использован аргумент `length.out`, определяющий общую длину необходимо вектора повторяющихся значений. В качестве первого аргумента функции `rep()` можно передавать не вектор, а конкретное значение, которое нужно размножить, например:

```
> rep("male", times = 5)
[1] "male" "male" "male" "male" "male"
```

Еще один распространенный случай создания векторов – формирование числовых последовательностей. Если нужно создать последовательность чисел с единичными интервалами, то можно воспользоваться оператором двоеточие “`:`”.

Числа могут быть как целыми, так и нецелыми, а также располагаться в обратном порядке:

```
> 1:5
[1] 1 2 3 4 5
> 3.3:6.3
[1] 3.3 4.3 5.3 6.3
> 3:-3
[1]  3  2  1  0 -1 -2 -3
```

Этот способ часто используется для создания индексирующих векторов при обращении к элементам какой-либо структуры данных.

Для создания последовательности чисел с интервалами, отличающимися от единицы, можно воспользоваться функцией `seq()`, которой необходимо передать начальное и конечное значения последовательности, а также либо аргумент `by` с интервалом, либо аргумент `length.out` с необходимой длиной.

```
> seq(0, 20, by = 5)
[1]  0  5 10 15 20
> seq(3, 4, length.out = 11)
 [1] 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0
```

Этот способ часто используется при формировании переменных для создания графиков функций.

Важнейшим свойством вектора является его длина, то есть число элементов, составляющих вектор. Выяснить длину вектора можно с помощью функции `length()`:

```
> length(n3)
[1] 7
> length(13:78)
[1] 66
```

4.2. Матрица

Матрицы в R представляют собой двумерные таблицы данных. Аналогично векторам, матрицы могут содержать данные только одного типа, при этом необязательно, чтобы этот тип был числовым.

Простейшим способом создания матрицы является функция `matrix()`:

```
> (m1 <- matrix(1:15, nrow = 3, ncol = 5))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

Первым аргументом этой функции передается вектор, который нужно преобразовать в матрицу (в данном случае, последовательность целых чисел от 1 до 15), аргументы `nrow` и `ncol` определяют необходимое число строк и столбцов соответственно. По умолчанию матрица заполняется по столбцам. При необходимости можно использовать аргумент логического типа `byrow`, определяющий построчный порядок заполнения матрицы:

```
> (m2 <- matrix(1:8, nrow = 2, byrow = TRUE))
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

Обратите внимание, что в данном примере опущен параметр `ncol`, в такой ситуации число столбцов определяется, исходя из числа строк и длины преобразуемого вектора.

Другим распространенным способом создания матриц являются функции `rbind()` и `cbind()`, которые служат для объединения в матрицу нескольких векторов по строкам и по столбцам соответственно. С помощью этих функций можно также объединять вектора и матрицы, если соответствующие размерности совпадают.

```
> (m3 <- cbind(1:3, 7:9))
     [,1] [,2]
[1,]    1    7
[2,]    2    8
[3,]    3    9
> (m4 <- rbind(m2, rep(0, 4)))
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    0    0    0    0
```

В языке R есть специальный оператор, который осуществляет перемножение матриц в соответствии с правилами линейной алгебры: “%*%”, в случае использовании обычного умножения будет осуществлено поэлементное перемножение элементов (подробнее см. гл. 8). Еще одной часто применяемой к матрицам операцией является транспонирование (когда строки и столбцы меняются местами), которое в R осуществляется функцией `t()`.

```
> t(m1)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
[5,]   13   14   15
```

```
> m3 %*% m2
     [,1] [,2] [,3] [,4]
[1,]   36   44   52   60
[2,]   42   52   62   72
[3,]   48   60   72   84
```

Размерность матрицы можно выяснить с помощью функции `dim()`, которая возвращает вектор из двух значений, первое из которых – число строк (считается первым измерением), второе – число столбцов (считается вторым измерением). Если требуется выяснить только одну размерность, можно воспользоваться функциями `nrow()` и `ncol()`, возвращающими число строк и столбцов соответственно:

```
> dim(m3)
[1] 3 2
> nrow(m3)
[1] 3
> ncol(m3)
[1] 2
```

4.3. Массив

Массив (array) представляет собой обобщение матрицы на большее число измерений, при этом массив также должен состоять из элементов одинакового типа. В некоторых случаях массивы удобны для организации хранения и обработки отдельных типов данных.

Массив создается функцией `array()`, которой передается вектор, содержащий данные, которые следует преобразовать в массив, а также векторный аргумент `dim` с размерностями необходимого массива:

```
> (a1 <- array(1:18, dim = c(2, 3, 3)))
, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3
     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18
```

В данном примере создан трехмерный массив последовательных целых чисел. Обратите внимание на «послойную» распечатку значений при выводе в консоль. В случае необходимости, массив может иметь и большее число измерений.

Размерность массива можно выяснить с помощью функции `dim()`, которая вернет вектор с длиной, соответствующей числу измерений массива, и значениями по каждому измерению:

```
> dim(a1)
[1] 2 3 3
```

4.4. Список

Список (list) – это структура данных в R, допускающая хранение в одном наборе элементов разного типа. Элементами списка могут являться как векторы различных атомарных типов (в том числе – и векторы единичной длины), так и матрицы, массивы и даже другие списки. Списки имеют важное значение в R, потому что многие функции возвращают результаты именно в виде списка.

Список можно создать с помощью функции `list()`:

```
> (ls1 <- list(3:7, "Some string", c(TRUE, FALSE, FALSE)))
[[1]]
[1] 3 4 5 6 7

[[2]]
[1] "Some string"

[[3]]
[1]  TRUE FALSE FALSE

> (ls2 <- list(n3, ls1, m2))
[[1]]
[1]   3   5   9   0   0   0 100

[[2]]
[[2]][[1]]
[1] 3 4 5 6 7

[[2]][[2]]
[1] "Some string"

[[2]][[3]]
[1]  TRUE FALSE FALSE
```

```
[[3]]
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

В первом примере создается список `ls1`, состоящий из трех элементов: целочисленного вектора, текстового вектора единичной длины и логического вектора. При распечатке содержимого списка в консоли элементы обозначаются номерами в двойных квадратных скобках, содержимое распечатывается с новой строки, между элементами списка оставляется пустая строка (это делается для улучшения читаемости). Во втором примере список `ls2` создается из уже имеющихся в рабочем пространстве элементов: числового вектора, списка и матрицы. Обратите внимание на особенности распечатки в консоли вложенных списков: сначала в двойных квадратных скобках отображается номер элемента во внешнем списке, на следующей строке в двойных квадратных скобках отображаются номера во вложенных списках (сначала номер во внешнем списке, затем номер во внутреннем), и только затем следует содержимое элемента.

Длину списка (число элементов) можно выяснить с помощью функции `length()`:

```
> length(ls2)
[1] 3
```

4.5. Фрейм данных

В оригинале обсуждаемая структура данных называется data frame. В русскоязычной литературе терминология еще не устоялась, используется несколько вариантов, включая «таблица данных», «фрейм данных», «кадр данных». Первый вариант не представляется нам удачным, поскольку собственно таблицей (table) в R называется таблица сопряженности, содержащая частоты наблюдений по нескольким категориальным переменным, существует специальный тип объекта и функция `table()`. Последний вариант также не представляется удачным ввиду слишком сильной аналогии с фотографией. Из возможных вариантов прямого перевода слова frame мы бы выбрали «остов, каркас», но этот вариант также не слишком удачен. В сложившейся ситуации мы считаем оптимальным использование прямой транслитерации, тем более что термин «фрейм» уже довольно широко распространен в различных предметных областях.

Фрейм данных является наиболее удобной структурой для использования при статистическом анализе. Он представляет собой прямоугольную таблицу

данных, которая в отличие от матрицы может содержать различные типы данных в разных столбцах. Технически фрейм представляет собой особый вид списка, элементами которого являются вектора одинаковой длины.

Из множества структур данных, доступных в R, фрейм больше всего соответствует типичной форме хранения научных данных: строки таблицы соответствуют наблюдениям (например, отобранным пробам), а столбцы – переменным, причем переменные часто имеют разный тип данных (например, данные о географическом положении могут иметь формат координат, данные об объекте представляют собой текстовую переменную, данные о концентрации химических веществ имеют числовой формат и т.д.). Именно в таком виде обычно хранятся исходные данные для обработки в популярных программах для анализа (включая Microsoft Excel, Statistica, SPSS).

В практике анализа данных фреймы обычно создаются в результате импорта из внешних источников (текстовых файлов, баз данных), однако в учебных целях полезно научиться создавать фреймы из векторов. Для создания фрейма служит функция `data.frame()`, которой передаются пары `имя_переменной = вектор_значений`:

```
> # названия озер
> v1 <- c("Лунское", "Силикатное", "Сормовское", "Мещерское")
> # даты отбора проб
> v2 <- c("05.2017", "06.2017", "06.2017", "08.2017")
> # концентрация железа
> v3 <- c(15.5, 16.2, 18.9, 25.6)
> # концентрация марганца
> v4 <- c(3.1, 6.1, 8.9, 2.5)
> # кислотность воды
> v5 <- c(5.4, 5.5, 6.0, 5.8)
> df <- data.frame(lake = v1, date = v2, Fe2 = v3, Mn = v4, pH = v5)
> df
        lake    date  Fe2  Mn  pH
1    Лунское 05.2017 15.5 3.1 5.4
2 Силикатное 06.2017 16.2 6.1 5.5
3 Сормовское 06.2017 18.9 8.9 6.0
4  Мещерское 08.2017 25.6 2.5 5.8
```

Здесь создается искусственный пример фрейма данных с информацией о мониторинге качества воды в озерах Нижнего Новгорода. Сначала создаются вектора с названиями озер и датами отбора проб (текстовые переменные), а также с результатами химического анализа воды (числовые переменные). Затем вектора комбинируются во фрейм `df`. Обратите внимание, что при распечатке в консоли были отображены номера строк и имена переменных. Дело в том, что

фрейм всегда обладает *атрибутами* имен переменных (names) и имен строк (row.names). По умолчанию именами строк назначаются их номера, а именами переменных служат имена объектов (векторов), которые включены во фрейм.

Несколько слов об атрибутах. Атрибуты – это характеристики объектов, которые относятся к их описанию, а не к содержимому. Объекты разных классов могут обладать разными атрибутами. Получить перечень атрибутов можно с использованием функции `attributes()`, которая возвращает список всех имеющихся у объекта атрибутов. Получить значения конкретных атрибутов можно с использованием функций, соответствующих их названиям, в случае фреймов `names()` и `row.names()`:

```
> attributes(df)
$names
[1] "lake" "date" "Fe2"  "Mn"   "pH"
$row.names
[1] 1 2 3 4
$class
[1] "data.frame"
> names(df)
[1] "lake" "date" "Fe2"  "Mn"   "pH"
> row.names(df)
[1] "1" "2" "3" "4"
```

Некоторые атрибуты допускают не только их получение с помощью функции, соответствующей названию атрибута, но и изменение. Например, имена строк фрейма можно менять следующими способами:

```
> row.names(df) <- c("obj1", "obj2", "obj3", "obj4")
> row.names(df)[2] <- "second object"
```

В первой строке атрибут фрейма `row.names` заменяется содержимым текстового вектора. Во второй строке меняется только имя второго объекта. Аналогичным образом можно менять и имена переменных, например:

```
> names(df)[5] <- "acidity"
> df
                    lake    date  Fe2  Mn acidity
obj1             Лунское 05.2017 15.5 3.1     5.4
second object Силикатное 06.2017 16.2 6.1     5.5
obj3          Сормовское 06.2017 18.9 8.9     6.0
obj4           Мещерское 08.2017 25.6 2.5     5.8
```

При создании фрейма с использованием функции `data.frame()` ей можно передавать не только вектора, но и другие фреймы, а также матрицы, при этом длина векторов и число строк объединяемых фреймов и матриц должны совпадать.

4.6. Фактор

Фактор представляет собой особый тип хранения текстового вектора. Факторы используются для оптимального хранения категориальных переменных, то есть данных, которые принимают небольшое количество фиксированных значений. Типичной категориальной переменной является пол. В стандартной ситуации переменная «пол» может принимать два значения (мужской и женский) и для ее хранения можно было бы использовать текстовый вектор:

```
> sex <- c("male", "male", "female", "male", "female")
```

Этот способ удобен, потому что по самим текстовым значениям легко понять, что они обозначают – в этом смысле текстовые значения почти всегда осмысленны. Недостатком этого способа является неоптимальное расходование памяти, поскольку текстовые переменные требуют для хранения значительно больше места, чем числовые. Вместо хранения текстовых значений можно было бы закодировать пол в виде чисел (например, 1 – мужчина, 2 – женщина) и хранить целочисленный вектор:

```
> sexInt <- as.integer(c(1, 1, 2, 1, 2))
```

Такой способ оптимален в плане хранения данных в памяти компьютера, но неудобен при анализе данных: расшифровку закодированных значений сложно постоянно держать в голове.

Для решения обозначенной проблемы в R и существуют факторы. Фактор кодирует значения категориальных переменных и хранит их в виде целочисленного вектора, при этом создается дополнительный текстовый вектор, который хранится в качестве атрибута `levels` (уровни фактора). Фактор можно создать из существующего вектора с помощью функции `factor()`:

```
> (fsex <- factor(sex))
[1] male   male   female male   female
Levels: female male
> attributes(fsex)
$levels
[1] "female" "male"
$class
[1] "factor"
```

При выводе в консоль фактор выглядит как текстовый вектор, однако наличие строки с расшифровкой уровней указывает на то, что это именно фактор. Получить доступ к числовым значениям фактора можно путем его преобразования в числовой вектор с помощью функции `as.numeric()`, а уровни фактора можно получить с помощью функции `levels()`. Поскольку уровни

хранятся в виде атрибута, их можно менять с помощью той же функции `levels()`, совмещенной с оператором присваивания:

```
> as.numeric(fsex)
[1] 2 2 1 2 1
> levels(fsex)
[1] "female" "male"
> levels(fsex) <- c("F", "M")
> fsex
[1] M M F M F
Levels: F M
```

Обратите внимание, что при первом преобразовании вектора `sex` в фактор `fsex` текстовые значения были закодированы в алфавитном порядке. Если необходим какой-то определенный порядок следования категорий, можно использовать аргумент `levels` функции `factor()`, также можно использовать аргумент `labels` для изменения обозначения категорий:

```
> (fsex2 <- factor(sex, levels = c("male", "female"),
+                       labels = c("Male", "Female")))
[1] Male   Male   Female Male   Female
Levels: Male Female
> as.numeric(fsex2)
[1] 1 1 2 1 2
```

В этом примере фактор `fsex2` сразу создан таким образом, чтобы мужские особи получили целочисленный код 1, а женские – код 2, при этом сразу скорректированы текстовые обозначения.

Другой удобный способ создания факторов в ситуациях, когда наблюдения сгруппированы – функция `gl()`. Эта функция принимает число градаций фактора (первый аргумент), число повторений каждой градации (второй аргумент), при этом можно использовать аргумент `labels`, который задает обозначения категорий:

```
> (fsex3 <- gl(2, 10, labels = c("M", "F")))
 [1] M M M M M M M M M M F F F F F F F F F F
Levels: M F
```

Здесь создан фактор для описания пола добровольцев, участвовавших в эксперименте, сгруппированных именно по полу: сначала 10 мужчин, а затем 10 женщин.

Факторы имеют большое значение в R, поскольку часто используются при статистическом моделировании. Некоторые специальные функции по-разному обрабатывают факторы и обычные текстовые переменные и даже используют тип переменной для определения необходимого вида анализа.

Глава 5. Обращение к данным

Анализируемые в среде R данные хранятся в памяти компьютера в виде специализированных структур (фреймов, векторов, списков). Для проведения различных манипуляций с данными, осуществления конкретных видов анализа необходимо иметь доступ к элементам структур, а также осуществлять выбор необходимых для анализа значений. Стандартными операторами обращения к данным в R служат квадратные скобки `[ ]`, двойные квадратные скобки `[[ ]]`, а также символ доллара `$`. Различные структуры данных имеют свои особенности обращения.

Обращение к отдельным подмножествам структур данных в R называется индексированием, поскольку для этого чаще всего используются так называемые индексирующие вектора.

5.1. Индексирование векторов

Структуры данных в R хранят элементы в определенной последовательности, то есть каждый элемент имеет свой номер, по которому к нему можно обращаться. Этот номер обычно называется индексом. В отличие от многих языков программирования в R нумерация начинается с единицы, то есть первый элемент всегда имеет индекс 1.

Оператором обращения к элементам векторов, матриц, массивов и фреймов служат квадратные скобки. Чтобы получить доступ к элементу после имени объекта в квадратные скобки нужно поместить его индекс:

```
> (num <- seq(3, 5, by = 0.2))
 [1] 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0
> num[1]
[1] 3
> num[10]
[1] 4.8
> sqrt(num[6]) ^ 3 + 2
[1] 10
```

В этом примере создана последовательность чисел от 3 до 5 с интервалом 0.2, затем получены 1-й и 10-й элементы, а 6-й элемент использован в арифметическом выражении. Вектор всегда имеет конечную длину; если при обращении к элементу индекс превысит длину вектора (то есть мы будем обращаться к элементу, которого в векторе нет), будет возвращено отсутствующее значение `NA`:

```
> length(num)
[1] 11
> num[13]
[1] NA
```

С помощью оператора обращения можно не только получать значения элементов вектора, но и менять их, то есть присваивать элементам новые значения:

```
> num[1] <- 2
> num
 [1] 2.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0
```

Здесь первому элементу присвоено новое значение, при этом остальные значения не изменились. При попытке присвоения нового значения отсутствующему в векторе элементу, вектор будет расширен до соответствующего индекса, при этом элементы между концом исходного вектора и новым элементом получат значения NA:

```
> num[13] <- 6
> num
 [1] 2.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0  NA 6.0
```

Если поместить в квадратные скобки не конкретное значение, а вектор целочисленных значений, то будут возвращены все значения с указанными индексами в том порядке, в котором приведены значения индексов:

```
> num[6:10]
[1] 4.0 4.2 4.4 4.6 4.8
> num[c(7, 2, 3, 1)]
[1] 4.2 3.2 3.4 2.0
```

В качестве индекса можно использовать целые отрицательные значения, в этом случае будут возвращены все элементы за исключением тех, чьи индексы имеют отрицательное значение:

```
> num[-10:-5]
[1] 2.0 3.2 3.4 3.6 5.0  NA 6.0
```

В этом примере возвращены все элементы вектора, за исключением тех, которые имеют номера с 5 по 10.

С использованием векторного индекса можно изменять сразу несколько значений:

```
> num[10:14] <- 7
> num
 [1] 2.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 7.0 7.0 7.0 7.0 7.0
```

В приведенных выше примерах индексирующий вектор создавался непосредственно в выражениях с помощью констант, оператора : и функции c(). Индекс можно создать заранее и сохранить в отдельном векторе:

```
> index <- c(1:3, 7)
> num[index]
[1] 2.0 3.2 3.4 4.2
```

Помимо индексирования с помощью целочисленных векторов существуют еще два способа: индексирование с помощью логических векторов и индексирование по именам элементов.

При индексировании с помощью логического вектора этот вектор должен иметь такую же длину, что и тот вектор, к которому происходит обращение. В результате будут возвращены те элементы вектора, которым соответствует значение `TRUE` в индексирующем логическом векторе:

```
> (index <- rep(c(TRUE, FALSE), times = 7))
 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
[11]  TRUE FALSE  TRUE FALSE
> num[index]
[1] 2.0 3.4 3.8 4.2 4.6 7.0 7.0
```

Здесь создан новый логический вектор длины 14, состоящий из чередующихся значений `TRUE` и `FALSE`. При использовании этого вектора в качестве индекса возвращены значения элементов с нечетными номерами. Для создания логических индексов обычно используются логические выражения. С их помощью можно легко выбирать из вектора значения, соответствующие каким-либо критериям:

```
> num[num == 7]
[1] 7 7 7 7 7
> num[num <= 6 & num >= 4]
[1] 4.0 4.2 4.4 4.6
> num[num %% 2 == 0]
[1] 2 4
```

В первом примере осуществляет выбор значений, равных 7. Выражение `num == 7` производит сравнение каждого из элементов вектора `num` с числом 7, его результатом является логический вектор, в котором значения `TRUE` стоят только в тех позициях, в которых в векторе `num` находятся числа 7. Полученный логический вектор используется в качестве индексирующего вектора, в результате происходит выборка всех семерок из вектора `num`. Во втором примере используется более сложное логическое выражение, которое находит в векторе `num` числа от 4 до 6 включительно, но логика применения та же самая: создается логический индексирующий вектор, с помощью которого происходит выбор необходимых нам данных. В третьем примере выбираются целые четные числа (такие, остаток от деления которых на 2 равен нулю). В практике анализа данных выражения такого рода постоянно используются для того, чтобы осуществлять выбор из наборов данных.

Последним способом обращения к элементам вектора является индексирование по имени. Элементам вектора можно присвоить имена и затем использовать их в качестве индекса. Имена элементов хранятся в атрибуте `names`, их можно задать при создании вектора с помощью функции `c()`, либо назначить потом с применением функции `names()` и оператора присваивания:

```
> (fruitColors <- c(apple = "green",cherry = "red",banana = "yellow"))
   apple   cherry   banana
 "green"    "red" "yellow"
> integers <- 1:3
> names(integers) <- c("one", "two", "three")
> integers
  one   two three
    1     2     3
```

Обратите внимание, что при выводе в консоль векторов с поименованными элементами не выводится первый номер элемента в начале новой строки. Обращаться к элементам таких векторов можно с помощью текстовых векторов, содержащих имена, возвращены будут значения вместе с именами:

```
> fruitColors["cherry"]
cherry
 "red"
> integers[c("one", "three")]
  one three
    1     3
```

5.2. Индексирование матриц и массивов

Матрицы и массивы в отличие от векторов имеют более одного измерения, поэтому для обращения к их элементам необходим не один, а несколько индексов, указывающих положение элементов по каждому из измерений. В случае матриц таких измерений два. Оператором обращения к элементам матриц и массивов также служат квадратные скобки, но в них помещается несколько индексов, которые отделяются запятой. Первым измерением считаются строки, вторым – столбцы.

Способов задания индексов также три: целочисленный вектор, логический вектор, текстовый вектор с именами элементов по измерениям (если имена заданы). В приведенных ниже примерах мы проиллюстрируем индексирование матриц. Индексирование массивов осуществляется аналогично с тем отличием, что указывается столько индексов, сколько измерений у массива.

```
> (m <- matrix(1:20, nrow = 4, ncol = 5))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> m[2,4]
[1] 14
> m[3:4,4:5]
     [,1] [,2]
[1,]   15   19
[2,]   16   20
> m[c(TRUE, TRUE, FALSE, FALSE), c(FALSE, FALSE, TRUE, FALSE, TRUE)]
     [,1] [,2]
[1,]    9   17
[2,]   10   18
```

Как видно, можно обращаться как к отдельному элементу, так и осуществлять выбор целых блоков. В последнем примере для выбора использованы логические вектора. Если необходимо извлечь строку или столбец целиком, то индекс соответствующего измерения можно опустить, но наличие запятой в этом случае все равно обязательно:

```
> m[2, 1:5]
[1]  2  6 10 14 18
> m[2, ]
[1]  2  6 10 14 18
> m[, 2:3]
     [,1] [,2]
[1,]    5    9
[2,]    6   10
[3,]    7   11
[4,]    8   12
```

В первом примере выбор второй строки осуществляется обычным способом с указанием индекса по второму измерению, во втором примере то же самое делается проще с применением пропуска индекса. В третьем примере выбираются второй и третий столбцы.

Существует возможность обращения к элементам матриц и массивов с использованием единственного индекса. В этом случае нумерация элементов производится последовательно вдоль измерений согласно их порядку. В матрицах сначала нумеруется первый столбец, затем второй, третий и так далее. В нашем примере с матрицей `m` значения элементов соответствуют их номерам при таком индексировании:

```
> m[3]
[1] 3
> m[5]
[1] 5
> m[17]
[1] 17
```

5.3. Индексирование списков

Ввиду того, что элементами списка могут являться объекты разного типа, способ обращения к элементам списка имеет свои особенности. Оператор обращения к элементам `[ ]` и описанные выше способы индексирования (вектор целочисленных значений, вектор логических значений, вектор имен элементов) работают со списками точно так же как с векторами в том смысле, что возвращается всегда объект того же класса, что и исходный. В случае векторов мы получаем набор элементов в виде вектора и это именно то, что необходимо нам для проведения вычислений. В случае списков мы будем получать набор элементов в виде списка. Даже если мы обращаемся к единственному элементу, оператор `[ ]` сформирует список из этого элемента.

```
> ls <- list(1:5, c("male", "female"), rep(TRUE, 5))
> ls[2:3]
[[1]]
[1] "male"   "female"

[[2]]
[1] TRUE TRUE TRUE TRUE TRUE

> ls[c(TRUE, TRUE, FALSE)]
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] "male"   "female"

> ls[1]
[[1]]
[1] 1 2 3 4 5
> class(ls[1])
[1] "list"
```

Для получения доступа к содержимому некоторого элемента списка используются двойные квадратные скобки `[[ ]]`. В результате использования

этого оператора возвращается уже не список, а объект того класса, который был помещен в список (вполне возможно, что этот объект также является списком). К элементам возвращаемого объекта можно обращаться обычным образом (к вектору – через квадратные скобки, к списку – через двойные квадратные скобки).

```
> ls[[3]]
[1] TRUE TRUE TRUE TRUE TRUE
> class(ls[[2]])
[1] "character"
> ls[[1]][3]
[1] 3
```

Элементы списка, также как и элементы вектора, могут иметь имена. Присвоить имена можно сразу при создании вектора (тогда функции `list()` нужно передавать пары `имя = элемент`), а можно воспользоваться атрибутом `names`:

```
> names(ls) <- c("integers", "strings", "logicals")
```

Для обращения к элементам поименованного списка существует специальный оператор `$` (знак доллара), который используется в форме `имя_списка$имя_элемента`. Можно также использовать имена в комбинации с рассмотренными выше операторами `[ ]` и `[[ ]]`.

```
> ls["logicals"]
$logicals
[1] TRUE TRUE TRUE TRUE TRUE
> ls[["strings"]]
[1] "male"   "female"
> ls$integers[5]
[1] 5
```

Обратите внимание, что в первом примере возвращен список, состоящий из одного элемента (на это указывает его название `$logicals`), во втором примере возвращен полный вектор текстовых значений, а в третьем – конкретный элемент вектора.

Последняя форма обращения к элементам списка используется в практике анализа данных чаще всего, поскольку лучше читается (нет нагромождения следующих друг за другом квадратных скобок) и позволяет не следить за порядком следования элементов (нужно знать только имя).

5.4. Индексирование фреймов

Фреймы данных являются частным случаем списков, поэтому при их использовании можно применять способы обращения к элементам, характерные для списков (операторы `[[ ]]` и `$`). С другой стороны, фреймы представляют собой прямоугольные таблицы, поэтому для обращения к их элементам можно использовать квадратные скобки, в которые помещаются два индекса (так же, как при обращении с матрицами).

```
> df <- data.frame(lake = c("Лунское", "Силикатное",
+                            "Сормовское", "Мещерское"),
+                 date = c("05.2017", "06.2017", "06.2017", "08.2017"),
+                   Fe2 = c(15.5, 16.2, 18.9, 25.6),
+                   Mn = c(3.1, 6.1, 8.9, 2.5),
+                   pH = c(5.4, 5.5, 6.0, 5.8))
> df[[4]]
[1] 3.1 6.1 8.9 2.5
> df$pH[3]
[1] 6
> df[2, 3]
[1] 16.2
> df[, 3]
[1] 15.5 16.2 18.9 25.6
> df[3:4, ]
        lake    date  Fe2  Mn  pH
3 Сормовское 06.2013 18.9 8.9 6.0
4  Мещерское 08.2013 25.6 2.5 5.8
```

В приведенном примере воссоздан фрейм с информацией о мониторинге качества воды в озерах Нижнего Новгорода из раздела 3.5. При обращении к элементам, содержащимся в одном столбце, возвращается вектор значений; если же происходит обращение к данным нескольких столбцов (которые могут быть разного типа), то возвращается фрейм данных.

При работе с фреймами, имеющими длинное название, постоянное использование оператора `$` для обращения к переменным (столбцам) может быть довольно утомительным. Для облегчения работы пользователя можно воспользоваться функциями для прикрепления и открепления фрейма `attach()` и `detach()`. Первая добавляет фрейм в перечень путей, в которых R ищет соответствие между вводимыми именами и объектами, существующими в рабочем пространстве. В результате появляется возможность обращаться к переменным фрейма по их именам без необходимости вводить название фрейма

и оператор `$`. Функция `detach()` открепляет фрейм и удаляет его из путей поиска, что возвращает ситуацию к исходному состоянию.

```
> attach(df)
> pH[1]
[1] 5.4
> df$Fe2
[1] 15.5 16.2 18.9 25.6
> detach(df)
> pH[1]
Error: object 'pH' not found
> df$pH[1]
[1] 5.4
```

После прикрепления фрейма появляется возможность использовать имена переменных фрейма как имена обычных векторов, существующих в рабочем пространстве. При этом сохраняется возможность обращения к ним как к элементам фрейма (например, `df$Fe2`). После открепления фрейма его переменные перестают быть доступны без указания имени фрейма (выдаваемое сообщение об ошибке после предпоследнего выражения гласит, что объект `pH` не найден).

Прикрепление фрейма может быть удобным, однако пользоваться этим методом необходимо с осторожностью. Дело в том, что многие наборы данных часто содержат стандартные переменные, названия которых совпадают (например, пол – sex, дата – date, вид – species). При попытке одновременного прикрепления таких фреймов произойдет конфликт названий и по имени будут доступны только одноименные переменные того фрейма, который был прикреплен последним. Такая ситуация чревата множеством трудно обнаружимых ошибок. Мы не рекомендуем использовать прикрепление при работе с несколькими фреймами. Если вы все же решили прикрепить фрейм, не забудьте обязательно открепить его вызовом функции `detach()`.

Еще один способ обращения к переменным фрейма без использования его имени и оператора `$` – специальные функции `with()` и `within()`. Первым аргументом этим функциям передается фрейм или список, к элементам которого необходимо обращаться напрямую. Вторым аргументом передается выражение, которое будет выполнено. В случае необходимости выполнения нескольких выражений их организуют в блок, заключенный в фигурные скобки. В приведенном ниже примере строится регрессионная модель зависимости кислотности от концентраций железа и марганца и выводятся в консоль коэфициенты этой модели (см. разд. 8.2 части 2).

```
> with(df,
+        {
+            model <- lm(pH ~ Fe2 + Mn)
+            model$coefficients
+        })
(Intercept)         Fe2          Mn
 4.31653626  0.05101611  0.07506930
```

При использовании функции `with()` исходный объект остается неизменным. Функция `within()` предназначена для модификации исходного объекта, она возвращает результат манипуляций с этим объектом. Например, добавить новую переменную с суммой концентраций металлов можно следующим образом:

```
> within(df,
+ {
+     Me <- Fe2 + Mn
+ })
        lake    date  Fe2  Mn  pH   Me
1    Лунское 05.2017 15.5 3.1 5.4 18.6
2 Силикатное 06.2017 16.2 6.1 5.5 22.3
3 Сормовское 06.2017 18.9 8.9 6.0 27.8
4  Мещерское 08.2017 25.6 2.5 5.8 28.1
```

Обратите внимание, что функция вернула обновленный фрейм, который был распечатан в консоли. Если мы хотим сохранить произведенные изменения, результат выполнения этой функции нужно присвоить исходному объекту:

```
> df <- within(df, Me <- Fe2 + Mn)
```

Глава 6. Управляющие структуры

Как любой полноценный язык программирования, R предоставляет пользователю набор конструкций для ветвления процесса выполнения программ. В R такими управляющими структурами являются условный оператор `if else`, циклы `for`, `while` и `repeat`. Управляющие структуры редко применяются в интерактивных сессиях, связанных с простым анализом данных, но они незаменимы при написании собственных функций.

6.1. Условный оператор if else

Условный оператор позволяет выполнять те или иные выражения в зависимости от выполнения некоторого условия. Общая конструкция оператора следующая:

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

Условие представляет собой выражение, которое возвращает логическое значение (например, сравнение). Если результатом выполнения этого выражения является `TRUE`, то выполняется выражение 1, если `FALSE` – выполняется выражение 2. Если в зависимости от условия необходимо выполнить несколько выражений, то соответствующий блок кода помещается в фигурные скобки, при этом для читаемости кода желательно придерживаться определенного стиля отступов. Условный оператор может быть в сокращенной форме, если опущено ключевое слово `else` и соответствующий блок выражений. При последовательной проверке нескольких условий, второй блок выражений может также являться условным оператором:

```
if (<условие 1>)
{
      блок выражений 1
} else if (<условие 2>)
{
      блок выражений 2
} else if …
```

Число последовательно проверяемых условий не ограничено.

В качестве примера рассмотрим простейший выбор значения переменной `y` в зависимости от заранее созданной переменной `x`.

```
> x <- 4
> if (x <= 0)
+ {
+     y <- x ^ 2
+ } else if (x > 0)
+ {
+     y <- sqrt(x)
+ } else
+ {
+     y <- NA
+ }
> y
[1] 2
```

Приведенный фрагмент кода присваивает переменной `y` квадрат значения `x`, если оно не положительное, либо квадратный корень значения `x`, если оно положительное. Последнее выражение после оператора `else` гарантирует, что переменной `y` будет присвоено хотя бы отсутствующее значение (например, если `x` также имеет значение `NA`). В нашем примере `x` равнялось 4, поэтому `y` было присвоено значение 2.

6.2. Цикл for

Для выполнения одного и того же блока кода несколько раз применяются операторы цикла, наиболее употребимым из которых является цикл `for`, который применяется в следующей форме:

```
for (итератор in вектор)
{
      блок выражений
}
```

`Итератор` – это переменная, которая меняет свое значение на каждой из итераций (повторений) цикла. Она последовательно принимает значения, содержащиеся в `векторе`. Если необходимо просто повторить блок кода несколько раз, то `вектор` создается прямо внутри оператора `for` и представляет собой целочисленную последовательность:

```
> w <- numeric(10)
> for (ii in 1:10)
+ {
+     w[ii] <- 5 ^ ii
+ }
```

```
> w
 [1]  5  25  125  625  3125  15625  78125  390625  1953125  9765625
```

В приведенном примере создается нулевой вектор `w` длины 10 и последовательно заполняется степенями числа 5, при этом итератор `ii` используется в качестве индекса для вектора `w`.

Вектор значений итератора может быть создан заранее. В следующем примере подсчитывается число целых значений в заранее созданной последовательности чисел:

```
> w <- seq(5, 500, length = 51)
> n <- 0
> for (ii in w)
+ {
+     if (ii == round(ii)) n <- n + 1
+ }
> n
[1] 6
```

Перед началом цикла создана переменная `n`, в которой будет храниться число целых значений, ей присвоено значение 0. Затем цикл `for` последовательно присваивает итератору `ii` все значения, содержащиеся в векторе `w`, проверяет каждое из них на целочисленность (с помощью условного оператора) и в случае необходимости увеличивает счетчик `n` на единицу. Обратите внимание, что управляющие структуры могут быть вложены одна в другую, это обычная практика.

Многие задачи, которые можно решать с помощью циклов (включая примеры настоящего раздела), могут быть преобразованы в форму векторизованных вычислений (см. гл. 8), которые выполняются в R с использованием низкоуровневых функций, что гораздо более эффективно. Поэтому рекомендуется воздерживаться от использования циклов при наличии альтернативы.

6.3. Цикл while

Цикл `while` имеет следующую структуру:

```
while (<условие>)
{
      блок выражений
}
```

Блок выражений выполняется до тех пор, пока условное выражение не вернет значение `FALSE`. В качестве примера рассмотрим реализацию метода Ньютона для вычисления корней положительных чисел:

```
> x <- 35
> y <- x/2
> while ( abs(y ^ 2 - x) > 0.00001)
+ {
+     y <- (y + x/y) / 2
+ }
> y
[1] 5.91608
> y ^ 2
[1] 35
```

На каждой итерации цикла происходит приближение `y` к корню из `x`. Поскольку приближаться к точному значению можно бесконечно, то необходимо задать необходимую точность. Как только заданная точность будет достигнута, цикл прекращает работу.

Цикл `while` используется в случаях, когда нельзя заранее определить необходимое число итераций. При использовании этого цикла необходимо соблюдать осторожность, формулируя условия выполнения, поскольку в случае ошибки легко можно получить бесконечные вычисления.

6.4. Бесконечный цикл repeat

Цикл `repeat` представляет собой бесконечный цикл, который выполняется до тех пор, пока не будет выполнен оператор `break`, который обычно содержится внутри условных операторов.

Метод Ньютона из предыдущего раздела может быть реализован следующим образом:

```
> x <- 47
> repeat
+ {
+     y <- (y + x/y) / 2
+     if (abs(y ^ 2 - x) < 0.00001) break
+ }
> y
[1] 6.855655
> y^2
[1] 47
```

Оператор выхода из цикла `break` является универсальным и может быть использован и в циклах `for` и `while`. В циклах также может быть использован оператор `next`, который позволяет пропустить текущее выражение и перейти к следующему:

```
> x <- 0
> repeat
+ {
+     x <- x + 1
+     if (x < 3) next
+     print(x)
+     if (x > 5) break
+ }
[1] 3
[1] 4
[1] 5
[1] 6
```

Данный пример увеличивает значение `x` на единицу на каждой итерации и выводит в консоль текущее значение. Если текущее значение меньше 3, происходит переход к следующей итерации, благодаря чему пропускается несколько начальных значений. Цикл прекращает работу, когда текущее значение превышает 5.

Цикл `repeat` нечасто применяется на практике, поскольку легко может порождать бесконечные вычисления, но может быть полезен при организации интерактивных интерфейсов для взаимодействия с пользователем.

Глава 7. Функции

Функции являются основным рабочим инструментом в среде R. В обычных ситуациях пользователь использует функции, которые содержатся в пакетах (включая базовую поставку R). Тем не менее, R позволяет пользователю создавать свои собственные функции, что удобно для автоматизации часто встречающихся задач.

7.1. Определение функции

Общая конструкция для определения функции:

```
имя_функции <- function(<аргументы>)
{
      тело функции
}
```

Телом функции называется набор выражений, который выполняется при вызове функции. Например, если мы хотим написать функцию, суммирующую два аргумента, можно выполнить следующий фрагмент кода:

```
simpleSum <- function(x, y)
{
    x + y
}
```

Функции в R являются объектами, поэтому после выполнения приведенного фрагмента в рабочем пространстве появится объект с именем `simpleSum`. После этого созданной функцией можно воспользоваться для суммирования:

```
> simpleSum(2, 3)
[1] 5
> simpleSum(1:4, seq(10, 40, by = 10))
[1] 11 22 33 44
```

7.2. Аргументы функции

При определении функции можно задать столько аргументов, сколько необходимо, при этом все они должны получить имена. Можно также сразу определить значение по умолчанию, которое будет использовано, если при

вызове функции не передается соответствующий аргумент. Переопределим нашу суммирующую функцию с использованием значений по умолчанию:

```
simpleSum <- function(x = 5, y = 10)
{
    x + y
}
```

Теперь можно использовать эту функцию вообще без аргументов, либо только с одним из них:

```
> simpleSum()
[1] 15
> simpleSum(x = 10)
[1] 20
> simpleSum(y = 20)
[1] 25
```

При вызове функции соответствие аргументов устанавливается либо по имени аргумента, либо по его позиции согласно следующим правилам:

1) сначала осуществляется поиск аргументов по имени и находятся полные соответствия;
2) затем осуществляется поиск частичных совпадений по именам: если имя передаваемого аргумента совпадает с началом имени аргумента функции, то между ними устанавливается соответствие; в случае нескольких частичных совпадений будет выдано сообщение об ошибке;
3) соответствие оставшихся аргументов устанавливается по положению, то есть первый аргумент в определении функции получает значение первого аргумента при вызове функции (при этом из обоих перечней исключаются аргументы, между которыми установлено полное или частичное соответствие по имени);
4) если после установления позиционного соответствия остались неиспользованные переданные аргументы, будет выдано сообщение об ошибке.

Перечень аргументов функции и их значения по умолчанию можно выяснить с помощью функции `args()`.

Для иллюстрации принципов установления соответствия аргументов напишем функцию для расчета объема усеченного конуса, который рассчитывается на основе радиусов верхнего и нижнего оснований (baseRadius и topRadius) и высоты конуса (height):

```
coneVolume <- function(baseRadius, topRadius = 0, height)
{
    pi*height*(baseRadius^2 + baseRadius*topRadius + topRadius^2) / 3
}
> args(coneVolume)
function (baseRadius, topRadius = 0, height)
NULL
```

Следующие три вызова функции эквивалентны:

```
> coneVolume(5, 1, 2)
[1] 64.92625
> coneVolume(height = 2, baseRadius = 5, topRadius = 1)
[1] 64.92625
> coneVolume(top = 1, 2, base = 5)
[1] 64.92625
```

В первом примере имена аргументов не использованы вообще, соответствие аргументов установлено исключительно по положению. Во втором примере использованы полные имена аргументов. Несмотря на то, что аргументы перечислены не в том порядке, в котором они идут в определении функции, соответствие аргументов установлено правильно. В третьем примере два аргумента сопровождаются сокращенными именами, а один передается без имени. Сначала устанавливается соответствие между аргументами `top` и `base` из передаваемого перечня аргументам `topRadius` и `baseRadius` из определения функции, после чего устанавливается соответствие единственного неименованного аргумента по положению (он соответствует первому незадействованному аргументу из определения).

Для радиуса верхнего основания установлено нулевое значение по умолчанию, что позволяет использовать нашу функцию для расчета объема обычного конуса:

```
> coneVolume(base = 3, h = 2)
[1] 18.84956
```

В R можно создавать функции, которые принимают неопределенное количество неименованных аргументов, в этом случае используется специальный объект `...` (многоточие). Этот аргумент в теле функции может быть передан другой функции (тогда эта функция получит все переданные аргументы, для которых не установлено соответствие), либо он может быть преобразован в список, к которому можно обращаться. В качестве примера напишем функцию, суммирующую квадраты всех полученных аргументов (приведенная реализация не является оптимальной и приведена исключительно в иллюстративных целях):

```
squareSum <- function(...)
{
    args <- list(...)
    res <- 0
    for (ii in 1:length(args))
    {
        res <- res + args[[ii]] ^ 2
    }
    res
}
```

В определении функции использован аргумент `...`, который преобразуется в список `args` с помощью функции `list()`. Затем создается вспомогательная переменная `res` для хранения промежуточного результата, после чего в цикле `for` происходит обращение к каждому из полученных аргументов, а переменная `res` увеличивается на квадрат соответствующего аргумента. Определенная таким образом функция может принимать неограниченное число аргументов:

```
> squareSum(1, 2, 3, 4, 5)
[1] 55
> squareSum(2, 7, 10)
[1] 153
```

7.3. Возвращаемое значение

Функции в R всегда возвращают один объект. Если необходимо вернуть несколько объектов, они должны быть объединены в виде списка (именно поэтому огромное количество функций статистического анализа возвращают именно списки).

По умолчанию возвращается результат последнего выражения в теле функции, если он не содержался в теле цикла (именно поэтому в функцию `squareSum` добавлена последняя строчка, распечатывающая итоговый результат). Для явного определения объекта, который необходимо вернуть, можно воспользоваться функцией `return()`, которая в теле функции должна получить возвращаемый объект. Эта функция используется также в тех случаях, когда необходимо прекратить выполнение программы. Когда внутри тела функции происходит вызов функции `return()` (например, при наступлении какого-либо условия), выполнение кода прекращается и функция возвращает значение.

```
howMany <- function(x)
{
    if (x == 0) return("null")
    if (x == 1) return("one")
    if (x > 1 & x <= 10) return("many")
    if (x > 10) return("lots")
    else return(NA)
}
```

Приведенная функция возвращает текстовую оценку полученного числа, последовательно сравнивая его с заданными величинами. Как только находится соответствие, функция возвращает текстовое значение и дальнейшие условные операторы не выполняются, что экономит вычислительные ресурсы.

```
> howMany(5)
[1] "many"
> howMany(0)
[1] "null"
> howMany(-1)
[1] NA
```

Глава 8. Векторизованные вычисления

Одним из главных преимуществ R является изначальная поддержка так называемых векторизованных вычислений, позволяющих использовать вектора в арифметических выражениях. Смысл векторизованных вычислений заключается в том, что арифметические операторы применяются к каждому из элементов вектора, а результатом является вектор соответствующей длины.

```
> (x <- 1:5)
[1] 1 2 3 4 5
> (y <- seq(0, 40, by = 10))
[1]  0 10 20 30 40
> x + 1
[1] 2 3 4 5 6
> x + y
[1]  1 12 23 34 45
```

В первом примере константа прибавляется к вектору, то есть эта константа прибавляется к каждому из элементов вектора и возвращается вектор полученных значений. Во втором примере суммируются два вектора, в результате осуществляется поэлементное сложение, возвращается также вектор. Аналогичным образом векторизуются другие арифметические операции:

```
> y / x
[1] 0.000000 5.000000 6.666667 7.500000 8.000000
> y ^ x
[1]   0   100   8000   810000   102400000
> y %/% x
[1] 0 5 6 7 8
```

Стандартные арифметические функции также возвращают вектор, состоящий из результатов применения необходимой операции к каждому из элементов переданного функции вектора:

```
> sqrt(y)
[1] 0.000000 3.162278 4.472136 5.477226 6.324555
> sin(1/x)
[1] 0.8414710 0.4794255 0.3271947 0.2474040 0.1986693
```

Во втором примере сначала осуществляется векторизованное деление, результатом является вектор отношений, который передается функции `sin()`, которая также возвращает вектор значений.

Векторизованные вычисления существенно упрощают выполнение операций при анализе данных. Рассмотрим простой пример рутинной операции: вычисление стандартного отклонения для выборки значений, содержащихся в

векторе. Пусть мы имеем вектор значений роста группы добровольцев, принимающих участие в исследовании:

```
> h <- c(191, 171, 175, 168, 164)
```

Для расчета стандартного отклонения необходимо выполнить следующие последовательные операции (см. гл. 2 части 2): просуммировать все значения, разделить сумму на число значений (получим среднее), рассчитать отклонения от среднего (вычесть среднее из каждого значения), возвести отклонения в квадрат, просуммировать квадраты отклонений, разделить сумму квадратов на число значений (с поправкой), взять квадратный корень. Описанная последовательность без использования векторизованных вычислений может выглядеть так:

```
> hMean <- (h[1] + h[2] + h[3] + h[4] + h[5])/length(h)
> devSum <- (h[1] - hMean)^2 + (h[2] - hMean)^2 + (h[3] - hMean)^2 +
+     (h[4] - hMean)^2 + (h[5] - hMean)^2
> ( hSD <- sqrt(devSum/(length(h)-1)) )
[1] 10.42593
```

Первое выражение вычисляет среднее и сохраняет его в переменную hMean, второе выражение вычисляет сумму квадратов отклонений от среднего и сохраняет ее в переменную devSum, последнее выражение вычисляет квадратный корень из отношения суммы квадратов отклонений к числу степеней свободы и сохраняет результат в переменную hSD. Легко видеть, что приведенный код очень громоздкий, поскольку требует обращения к каждому из элементов исходного вектора.

Для облегчения расчета сумм можно воспользоваться функцией sum(), которая осуществляет суммирование переданных ей аргументов (в типичном случае передается один вектор, но можно передать и несколько), что упростит первое выражение:

```
> hMean <- sum(h)/length(h)
```

Упростить второе выражение без использования векторизованных вычислений можно только с помощью цикла. К счастью, R поддерживает векторизованные вычисления, благодаря чему второе выражение проще организовать следующим образом:

```
> devSum <- sum( (h - hMean)^2 )
```

Разберем это выражение на составляющие. Сначала осуществляется вычитание среднего hMean из вектора исходных значений h, результатом является вектор разностей:

```
> h - hMean
[1] 17.2 -2.8  1.2 -5.8 -9.8
```

Затем к полученному вектору разностей применяется оператор возведения в квадрат, результатом также является вектор:

```
> (h - hMean)^2
[1] 295.84   7.84   1.44  33.64  96.04
```

Полученный вектор квадратов отклонений суммируется:

```
> sum( (h - hMean)^2 )
[1] 434.8
```

В результате вся последовательность операций по расчету стандартного отклонения вектора может быть выполнена в одном выражении:

```
> sqrt( sum( (h - sum(h)/length(h))^2 ) / (length(h) - 1) )
[1] 10.42593
```

Разумеется, для выполнения рутинных операций существуют специальные функции. В частности, вычислить стандартное отклонение можно с использованием функции `sd()`:

```
> sd(h)
[1] 10.42593
```

Тем не менее, этот пример демонстрирует удобство использования векторизованных вычислений.

Векторизованные вычисления обычно не вызывают проблем, когда применяются к векторам одинаковой длины. В этом случае операция выполняется поэлементно. Однако синтаксис R допускает выполнение векторизованных операций с векторами разной длины. В этом случае происходит достраивание меньшего вектора до тех пор, пока не будет достигнута длина большего вектора, при этом происходит последовательное копирование меньшего вектора. Рассмотрим следующий пример, в котором суммируются вектора разной длины:

```
> x <- 1:6
> y <- c(10, 20)
> x + y
[1] 11 22 13 24 15 26
```

Вектор `x` в три раза длиннее вектора `y`. При их суммировании вектор `y` повторяется три раза, после чего осуществляется поэлементное суммирование. В результате к нечетным элементам вектора `x` прибавляется 10, а к четным – 20. Такая же операция «размножения» происходит при использовании констант в векторизованных вычислениях. Поскольку любая константа в R трактуется как вектор единичной длины, при использовании в комбинации с векторами константа будет размножена необходимое число раз.

Если длины векторов не соотносятся как целые числа, вектор меньшей длины используется нецелое число раз при достраивании вектора, при этом выдается соответствующее предупреждение:

```
> x <- 1:5
> y <- c(10, 100, 1000)
> x + y
[1]   11 102 1003 14 105
Warning message:
In x + y : longer object length is not a multiple of shorter object
length
```

Использование вычислений с векторами неравной длины является своего рода трюком, которым в некоторых ситуациях удобно пользоваться. Например, чтобы выбрать из некого вектора каждое третье значение, можно применить следующую форму индексации:

```
> x <- 30:60
> x[c(FALSE, FALSE, TRUE)]
 [1] 32 35 38 41 44 47 50 53 56 59
```

Для индексации с помощью логических значений необходим вектор, длина которого соответствует длине индексируемого вектора. В данном случае вектор x имеет длину 31, а в качестве индекса используется вектор из трех логических значений. В ходе выполнения операции обращения индексирующий вектор (длиной 3) «размножается» до длины индексируемого вектора (31), в результате получается вектор логических значений, в котором на каждой третьей позиции стоит значение TRUE. В итоге происходит выбор каждого третьего элемента вектора x.

Использование такого рода конструкций нетипично. Чаще всего операции с векторами неравной длины возникают по ошибке. Поскольку R не воспринимает такие операции как ошибочные и просто использует «размножение» вектора меньшей длины (при этом даже не во всех случаях выдавая предупреждение), отследить и исправить такие ошибки довольно проблематично.

Глава 9. Манипулирование данными

При работе с данными львиная доля времени затрачивается на их подготовку, преобразование и форматирование для конкретного вида анализа. При подготовке данных для анализа зачастую возникает необходимость преобразования, создания новых или удаления существующих переменных, сортировки либо ранжирования значений, а также формирования подвыборок из массивных наборов данных.

В настоящем разделе мы рассмотрим манипулирование данными на примере набора данных `trees`, который содержится в пакете `datasets` базовой поставки R.

```
> data(trees)
> names(trees)
[1] "Girth"  "Height" "Volume"
```

После загрузки этого набора данных в рабочем пространстве становится доступным фрейм `trees`, содержащий данные об окружности ствола в дюймах (переменная `Girth`), высоте в футах (`Height`) и объеме древесины в кубических футах (`Volume`) 31 дерева американской вишни.

9.1. Создание и удаление переменных

В первую очередь преобразуем имеющиеся данные в метрическую систему мер (1 дюйм = 2.54 см, 1 фут = 0.3048 м):

```
> trees$Girth <- trees$Girth * 2.54
> trees$Height <- trees$Height * 0.3048
> trees$Volume <- trees$Volume * 0.3048 ^ 3
```

При выполнении каждого из приведенных выражений происходит следующее: содержимое столбца фрейма умножается на коэффициент пересчета (результат этой операции представляет собой вектор), затем результат присваивается исходной переменной, в итоге исходные данные оказывается перезаписанными. Так следует поступать, когда не предполагается работа с исходными данными. Однако во многих случаях разумнее сохранить исходные данные, а результат преобразования сохранить в новую переменную.

Предположим, что мы хотим исследовать зависимость высоты деревьев от радиуса ствола, тогда как имеются исходные данные только о длине окружности. Исходя из предположения о круглом сечении ствола, длина окружности легко может быть преобразована в оценку радиуса (путем деления на 2π). Чтобы

создать в имеющемся фрейме новую переменную нужно просто выбрать ее имя и присвоить необходимые значения:

```
> trees$Radius <- trees$Girth / 2 / pi
```

Во фрейме `trees` будет создана новая переменная `Radius` (четвертый столбец), в которую будет записан результат деления длины окружности на 2π. Аналогичного результата можно добиться с использованием индексации через номера столбцов, но тогда необходимо будет присвоить имя новой переменной вручную с использованием атрибута `names`:

```
> trees[, 4] <- trees[, 1] / 2 / pi
> names(trees)[4] <- "Radius"
```

При анализе данных часто возникает необходимость выбора отдельных переменных из большого набора данных. Для решения этой задачи можно либо удалить ненужные переменные из исходного фрейма, либо создать новый набор данных, воспользовавшись инструментами индексации и присвоения. Предположим, что мы не собираемся использовать в анализе данные о длине окружности ствола и объеме древесины. Тогда можно создать новый фрейм данных и далее работать с ним:

```
> newTrees <- trees[, c("Height", "Radius")]
```

В результате выполнения этого выражения будут выбраны значения высоты и радиуса ствола всех деревьев (поскольку первый индекс пропущен) и помещены в новый фрейм данных. Если же мы уверены, что данные об окружности ствола и объеме древесины нам не понадобятся (по крайней мере, в текущей сессии) результат выбора можно переписать под имеющимся именем `trees`, что будет эквивалентно удалению двух переменных:

```
> trees <- trees[, c("Height","Radius")]
```

Еще один способ удаления переменной заключается в присвоении ей значения `NULL`, означающего отсутствие объекта:

```
> trees$Girth <- trees$Volume <- NULL
```

В результате ненужные нам более переменные будут удалены.

9.2. Сортировка и ранжирование

Для простой сортировки данных служит функция `sort()`, которой, помимо подлежащего сортировке вектора, можно передать аргумент логического типа `decreasing`, который указывает, нужно ли сортировать в порядке от большего к меньшему. По умолчанию, аргумент `decreasing` имеет значение `FALSE`, поэтому

при обычной сортировке в порядке увеличения его можно не указывать. Отсортируем значения высоты деревьев:

```
> (sortedHeight <- sort(trees$Height))
 [1] 19.2024 19.5072 19.8120 20.1168 21.0312 21.3360 21.6408 21.9456
 [9] 21.9456 22.5552 22.5552 22.8600 22.8600 22.8600 23.1648 23.1648
[17] 23.4696 23.7744 24.0792 24.3840 24.3840 24.3840 24.3840 24.3840
[25] 24.6888 24.6888 24.9936 25.2984 25.9080 26.2128 26.5176
```

Если поместить значения вектора `sortedHeight` в соответствующую переменную фрейма `trees` мы потеряем соответствие между содержимым столбцов, поскольку значения одной переменной будут переставлены, а второй – останутся на прежних позициях. Часто бывает необходимо отсортировать наблюдения (строки фрейма данных) в соответствии со значениями какой-либо переменной. Для решения этой задачи подходит функция `order()`. Эта функция возвращает индексирующий вектор, который можно использовать для сортировки:

```
> order(trees$Height)
 [1]  3 20  2  7 14  1 19  4 24 16 23  8 10 15 12 13 25 21 11  9 22
[22] 28 29 30  5 26 27  6 17 18 31
```

Полученный вектор содержит номера элементов, которые необходимо поместить в соответствующие позиции для получения отсортированного вектора. Так, на первую позицию нужно поставить 3-й элемент исходного вектора, на вторую – 20-й элемент и так далее. Результат будет идентичен применению функции `sort()`:

```
> trees$Height[order(trees$Height)]
 [1] 19.2024 19.5072 19.8120 20.1168 21.0312 21.3360 21.6408 21.9456
 [9] 21.9456 22.5552 22.5552 22.8600 22.8600 22.8600 23.1648 23.1648
[17] 23.4696 23.7744 24.0792 24.3840 24.3840 24.3840 24.3840 24.3840
[25] 24.6888 24.6888 24.9936 25.2984 25.9080 26.2128 26.5176
```

Таким образом, для сортировки наблюдений во фрейме нужно использовать результат выполнения функции `order()` в качестве индекса:

```
> trees <- trees[order(trees$Height), ]
```

Строки фрейма `trees` будут переставлены в порядке увеличения высоты деревьев, а результат сохранен под тем же именем. Обратите внимание, что в данном случае исходные позиции деревьев сохранятся в текстовом формате в атрибуте `row.names`.

Рангом называется положение наблюдения в отсортированном векторе. Наименьший элемент имеет ранг 1, следующий ранг 2 и так далее. Для получения рангов служит функция `rank()`. Рассчитаем ранги деревьев по их радиусу и поместим в новую переменную:

```
> (trees$RadiusRank <- rank(trees$Radius))
 [1]  3.0 20.0  2.0  7.5 14.0  1.0 19.0  4.0 24.0 16.5 23.0  7.5
[13] 10.0 15.0 12.5 12.5 25.0 21.0 11.0  9.0 22.0 28.0 29.5 29.5
[25]  5.0 26.0 27.0  6.0 16.5 18.0 31.0
```

Обратите внимание, что ранги могут принимать нецелое значение. Так происходит в том случае, когда имеются совпадающие значения, тогда каждое из таких значений получает одинаковый средний ранг. Например, если совпадают седьмое и восьмое значения в упорядоченном векторе, каждое из этих значений получит ранг 7.5.

9.3. Формирование подвыборок

Типичной задачей при подготовке данных является формирование подвыборок из обширного набора данных. Предположим, что мы хотим отдельно проанализировать высокие и низкие деревья. Для этого нам необходимо разделить имеющийся фрейм на два: в один выбрать деревья ниже 23 м, а в другой – выше 23 м. Для решения этой задачи можно воспользоваться индексированием с помощью логического вектора, который можно получить с использованием логического оператора.

```
> trees.low <- trees[trees$Height < 23, ]
> trees.high <- trees[trees$Height >= 23, ]
```

При выполнении первого выражения происходит следующее: сначала выполняется логическое выражение (`trees$Height < 23`), сравнивающее высоту деревьев с константой, результатом является логический вектор, в котором значения `TRUE` находятся в позициях, соответствующих низким деревьям, затем логический вектор используется в качестве индекса при обращении к фрейму, за счет чего происходит выбор строк, соответствующих низким деревьям, результат помещается в новый фрейм. Аналогично формируется отдельный фрейм с данными по высоким деревьям.

При формировании подвыборок можно использовать логические выражения любой сложности. Приведенный ниже пример помещает во фрейм `trees.specific` деревья высотой от 20 до 22 метров, имеющие радиус ствола более 5 см:

```
> (trees.specific <- trees[trees$Height > 20 & trees$Height < 22 &
+                              trees$Radius > 5, ])
   Height   Radius RadiusRank
19 21.6408 5.538274         19
24 21.9456 6.468057         24
```

Одновременно с формированием подвыборок наблюдений можно осуществлять и выбор необходимых для анализа переменных, если не опускать второй индекс при обращении к исходному фрейму.

Для формирования подвыборок из фреймов существует специальная функция `subset()`, которая работает эффективнее прямого индексирования и позволяет упростить синтаксис выражений. Первым аргументом (`x`) этой функции передается объект, из которого осуществляется выбор, вторым (`subset`) аргументом обычно является логическое выражение, определяющее, что нужно выбрать, также можно использовать аргумент `select`, указывающий подлежащие выбору столбцы (по умолчанию выбираются все). Приведенный ниже пример осуществляет выбор высоких деревьев с относительно небольшим радиусом ствола:

```
> subset(trees, Height > 25 & Radius < 6)
    Height   Radius RadiusRank
6  25.2984 4.365938        6.0
17 25.9080 5.214871       16.5
18 26.2128 5.376572       18.0
```

Обратите внимание, что при вызове функции `subset()` при формировании логического выражения можно не указывать имя фрейма (поскольку функция знает, откуда предстоит сделать выбор), что упрощает синтаксис сложных логических выражений.

Глава 10. Основы графической системы

10.1. Основные графические подсистемы R

Одним из самых существенных достоинств среды R является ее графическая система, позволяющая создавать графики практически любой сложности в любом необходимом качестве и формате (в частности, для разведочного анализа обычно используются графики невысокого разрешения, которые отображаются на экране, а при подготовке отчетов или публикаций создаются иллюстрации высокого разрешения в графических форматах).

Основными графическими подсистемами R являются `base`, `lattice` и `ggplot2`. Как следует из названия, первая является базовой, она реализована в стандартных пакетах `graphics` и `grDevices`, которые присутствуют в среде R изначально. Именно эта подсистема будет использоваться в настоящем пособии. Ключевой особенностью системы `base` является то, что графики могут создаваться и аннотироваться последовательно путем вызова соответствующих функций. Обычно новый график создается одной из базовых функций (`plot()`, `hist()`, `bar()` и т.п. в зависимости от необходимого типа графика), а затем необходимые элементы (дополнительные линии, подписи, оси, легенды) добавляются или модифицируются вызовом соответствующих функций.

Подсистемы `lattice` и `ggplot2` реализованы в одноименных пакетах расширения, которые требуют дополнительной установки. Подсистема `lattice` предназначена для быстрого создания категоризованных графиков, ее особенностью является использование однократного вызова функций для создания графика (все особенности и параметры графика передаются в функцию через ее аргументы), дальнейшее аннотирование и модификации не применяются. Подсистема `ggplot2` является реализацией идей, изложенных в книге Леланда Уилкинсона «Грамматика графики» (Leland Wilkinson “Grammar of Graphics”). В этой подсистеме графики разбираются на семантические составляющие, такие как слои, цветовые кодировки, доверительные интервалы и т.п. Создатели подсистемы `ggplot2` постарались использовать лучшие элементы из подсистем `base` и `lattice`, оставив за бортом неудобства. Тем не менее, эта подсистема наиболее сложна в освоении.

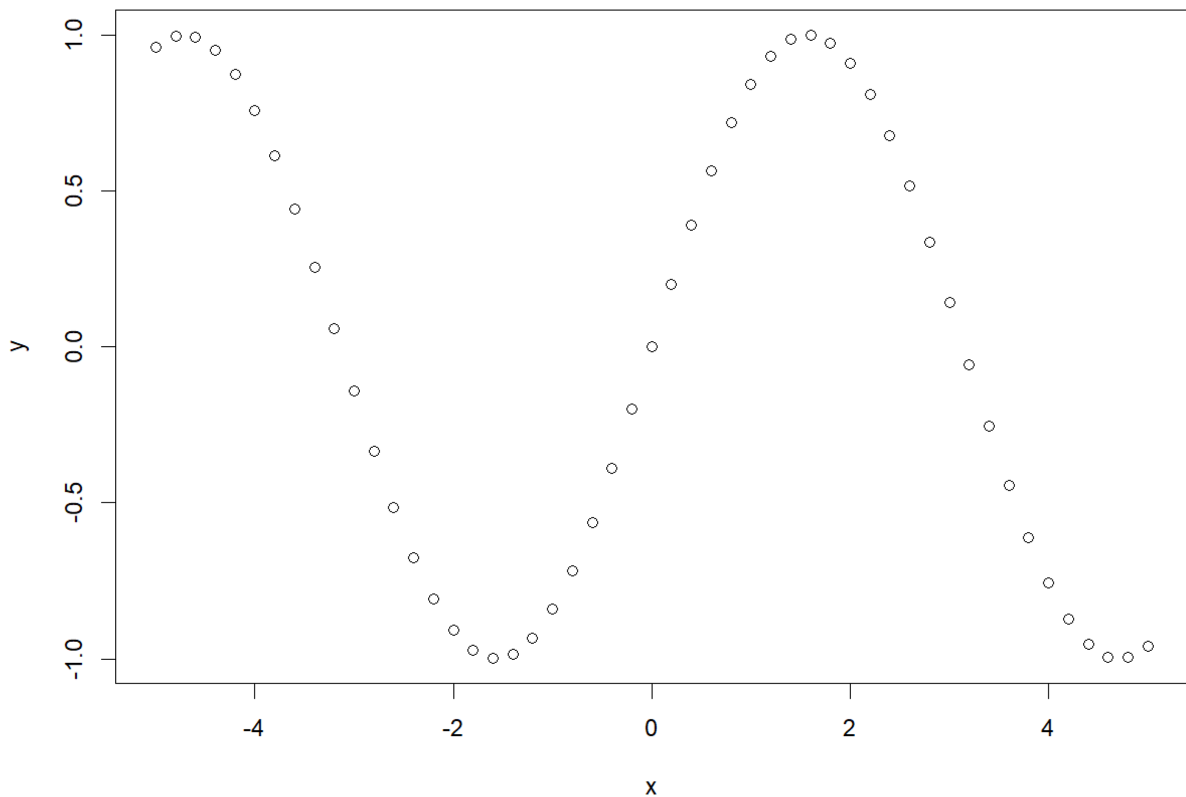

Рис. 2. Простейший график синусоиды

10.2. Функция `plot()` и ее основные аргументы

Начнем знакомство с графической подсистемой `base` с построения простейшего графика – синусоиды. Для начала необходимо сгенерировать координаты точек, которые мы поместим на график:

```
> x <- seq(from = -5, to = 5, by = 0.2)
> y <- sin(x)
```

Вектор `x` представляет собой абсциссы точек (от -5 до 5 с шагом 0.2), `y` – соответствующие им ординаты, вычисленные с помощью стандартной функции `sin()`. Нанесем эти точки на график с помощью функции `plot()`:

```
> plot(x, y)
```

При работе в RStudio после выполнения этого выражения будет создано новое окно графика, которое появится во вкладке Plots нижнего правого блока. В окне будет создан график, представленный на Рис. 2. На этом графике отображены точки, абсцисса которых содержится в первом аргументе (векторе `x`), а ордината во втором (векторе `y`), причем размеры поля автоматически выбираются таким образом, чтобы вместить все точки. Также присутствуют проградуированные оси координат и подписи к ним (по умолчанию они соответствуют именам аргументов, принимаемых при вызове функции).

Помимо координат точек, подлежащих отображению, функции plot() можно передать огромное количество аргументов, определяющих вид графика. Аргументы main, sub, xlab и ylab должны быть текстовыми константами, они определяют основной заголовок, подзаголовок, подписи осей абсцисс и ординат соответственно. Аргументы xlim и ylim должны быть двухэлементными векторами, они определяют пределы отображения по осям.

Аргумент type определяет тип графика и является текстовой константой со следующими возможными значениями: «p» (points) – точки, «l» (lines) – линии, «b» (both) –точки и линии без пересечения, «o» (overlap) – точки и линии с пересечением, «h» (histogram) – гистограммо-подобные вертикальные линии, «s» (steps) – ступенчатая линия, «n» (nothing) – ничего не отображается, но при этом создается график и определяются оси.

Аргумент pch (point character) определяет тип символа и может принимать значения от 0 до 25, по умолчанию этот аргумент имеет значение 1, соответствующее незалитым кругам. При использовании символов с 21 по 25 можно отдельно задавать цвет заливки символа с использованием аргумента bg.

Аргумент lty (line type) определяет тип линии и может принимать текстовые либо соответствующие им цифровые значения: «blank» (0) – линия отсутствует, «solid» (1) – сплошная линия, «dashed» (2) – штриховая линия, «dotted» (3) – пунктирная линия, «dotdash» (4) – штрих-пунктирная линия, «longdash» (5) – линия с длинными штрихами, «twodash» (6) – линия с двойными штрихами.

Аргумент col (color) определяет цвет линий и символов, он может принимать текстовые либо целочисленные значения от 0 до 8. Текстовые значения аргумента col могут быть двух видов: англоязычные наименования цветов («red», «brown», «orchid» и т.п., всего доступно 657 вариантов, полный список можно получить, воспользовавшись функцией colors()), а также строки вида «#RRGGBB», где RR, GG и BB – шестнадцатеричные значения компонентов системы RGB. Для облегчения использования системы RGB можно воспользоваться функцией rgb(r, g, b, alfa), которая принимает три числовых аргумента, соответствующих интенсивностям цветов, а также один аргумент, определяющий степень прозрачности (все они должны быть в диапазоне от 0 до 1).

Аргумент cex (character expansion) определяет размер символов, он принимает числовые значения и по умолчанию имеет значение 1. Размер символов указывается в относительных величинах, то есть cex = 1.5 означает, что размер символов будет увеличен в 1.5 раза относительно стандартного

размера, соответствующего `cex = 1`. Аргумент `lwd` (line width) определяет толщину линий также в относительных величинах.

Выше были описаны только самые основные аргументы графических функций, позволяющие настраивать вид графиков. Полный перечень графических параметров представлен в Таблице 2 (см. разд. 10.4).

Возвращаясь к нашему примеру с графиком синусоиды, перестроим его с использованием залитых ромбиков, соединенных сплошной линией двойной толщины темно-синего цвета, зададим диапазон отображения таким образом, чтобы уместился один период, а также добавим заголовок и подписи осей:

```
> plot(x, y, main = "График функции", xlab = "x", ylab = "f(x)",
+      type = "o", lty = 1, lwd = 2, pch = 1, col = "darkgrey",
+      xlim = c(-pi, pi), ylim = c(-1.5, 1.5))
```

После выполнения этого выражения будет построен новый график, который также отобразится на вкладке Plots интерфейса RStudio. При этом предыдущий график также сохранится, а перемещаться между последовательно созданными графиками можно будет с помощью стрелок в верхнем левом углу вкладки.

10.3. Аннотирование графиков и добавление элементов

Предположим теперь, что нам необходимо разместить на графике не только синусоиду, но и косинусоиду, и при этом неплохо будет создать легенду для облегчения чтения графика. Базовая графическая подсистема R позволяет легко добавлять новые элементы на уже существующий график.

Для добавления косинусоиды можно воспользоваться функцией `points()` либо функцией `lines()`. Первая изначально предназначена для добавления точек на существующий график, вторая – для добавления линий. Тем не менее, обе функции поддерживают весь набор стандартных аргументов (`type`, `pch`, `lty` и т.д.). Сначала необходимо создать новый вектор, в котором будут храниться ординаты косинусоиды, а затем добавить саму линию на наш график:

```
> y2 <- cos(x)
> points(x, y2, type = "o", lty = 1, lwd = 2, pch = 23, cex = 0.75,
+        col = "darkgrey", bg = "white")
```

Добавим теперь легенду на наш график, для чего воспользуемся функцией `legend()`. Аргументы `x` и `y` этой функции определяют положение легенды, их можно задать в виде координат, но чаще используются текстовые ключевые слова для аргумента `x`: «bottomright» (справа вверху), «bottom» (внизу по центру), «bottomleft» (внизу слева), «left» (слева по центру), «topleft» (сверху слева), «top» (сверху по центру), «topright» (справа вверху), «right» (справа по центру) и

«center» (в центре графика). При использовании ключевых слов аргумент y не используется.

Аргумент legend представляет собой вектор подписей для элементов легенды. Для описания типов линий, символов и цветов необходимо воспользоваться стандартными графическими аргументами, при этом, если для различных элементов легенды какие-то характеристики не отличаются, то их можно передать в виде константы. Если же графические компоненты легенды отличаются, то для каждого такого аргумента необходимо передать вектор, содержащий столько элементов, сколько будет элементов в легенде, чаще всего такие вектора создаются с использованием функции c(). В нашем примере с тригонометрическими функциями легенду можно построить так:

```
> legend("topleft", legend = c("sin(x)", "cos(x)"), lty = 1, lwd = 2,
+         pch = c(18, 5), col = "darkgrey")
```

В левом верхнем углу появится легенда из двух элементов.

Для добавления различных элементов на график существует множество дополнительных функций. Функция grid() добавляет на график сетку опорных линий. Функция abline() предназначена для рисования прямых линий с указанием пересечения с осью ординат (аргумент a) и наклона (аргумент b), при рисовании вертикальных либо горизонтальных линий достаточно указать только соответствующую координату (аргумент v для вертикальных линий и аргумент h для горизонтальных). Воспользуемся этими функциями для добавления сетки и координатных осей на наш график.

```
> grid()
> abline(h = 0, v = 0)
```

Предположим теперь, что нам необходимо указать точку пересечения графиков, для этого воспользуемся функциями arrows() и text():

```
> arrows(x0 = -2.35, y0 = -1.2, x1 = -2.35, y1 = -0.8, angle = 15)
> text(x = -2.35, y = -1.35, labels = "пересечение")
```

Функция arrows() предназначена для нанесения стрелок на график. Аргументы x0, y0, x1 и y1 определяют координаты начальной и конечной точек стрелки, аргумент angle определяет угол стрелки, можно также воспользоваться аргументом code (1 – стрелка будет нарисована только в конце, 2 – стрелка будет нарисована только в начале, 3 – будет нарисована двусторонняя стрелка). Функция text() добавляет текст в произвольную точку графика, определяемую аргументами x и y, собственно текст передается аргументу labels, для определения размера, стиля, цвета текста можно воспользоваться стандартными аргументами.

Итоговый график представлен на Рис. 3.

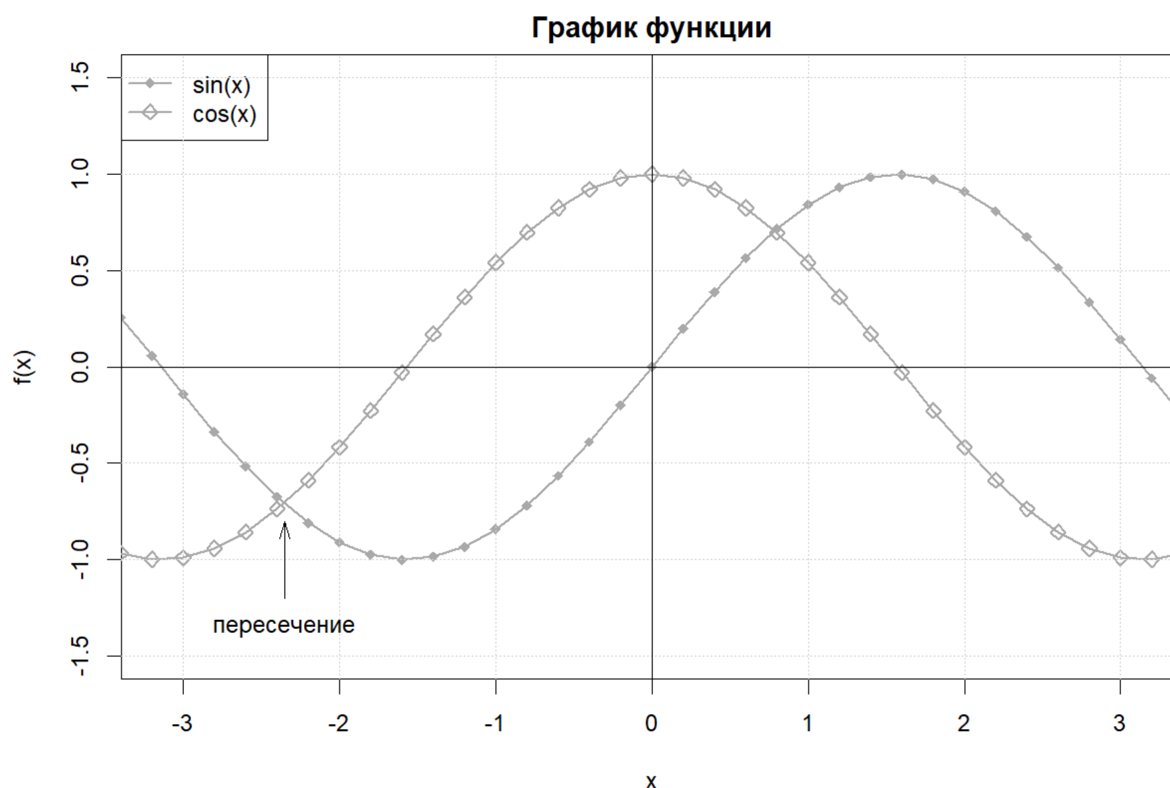

Рис. 3. Аннотированный график двух функций

Отметим, что многие функции, такие как `abline()`, `arrows()`, `text()`, позволяют добавлять сразу множество объектов на график, для чего необходимо передавать ключевым аргументам соответствующие вектора. Упомянем также часто используемые функции `segments()` – добавляет линейные сегменты, `polygon()` – рисует многоугольники, `rect()` – рисует прямоугольники, `mtext()` – добавляет надписи на полях графика.

В настоящем разделе перечислены далеко не все возможные аргументы графических функций, при настройке вида ваших графиков мы рекомендуем обращаться к справочной системе и не бояться использовать метод проб и ошибок.

10.4. Использование функции `par()`

Для настройки графических параметров, которые предполагается использовать неоднократно, служит функция `par()`. Эта функция позволяет изменять настройки графических параметров, установленные в системе по умолчанию, чтобы использовать их в течение текущей сессии R. При вызове этой функции без аргументов, в консоль будет выведен полный список графических параметров с их текущими значениями. При необходимость изменения

параметров функция вызывается в форме `par(имя_параметра1 = значение_параметра1, имя_параметра2 = значение_параметра2, …)`.

Предположим, что в течение ближайшей сессии работы с R мы собираемся строить графики, используя в качестве символа маленькие залитые квадраты. Вместо того, чтобы каждый раз при вызове функций `plot()`, `points()`, `lines()` и т.п. использовать аргументы `pch = 15` и `cex = 0.5`, можно поменять параметры по умолчанию:

```
> par(pch = 15, cex = 0.5)
```

После выполнения этого выражения на всех графиках будут использоваться залитые квадраты в два раза меньше стандартных, если при вызове функций не используются в явном виде другие значения.

Полный список графических параметров, настраиваемых с помощью функции `par()`, приведен в Таблице 2. Часть этих параметров может быть настроена только через функцию `par()` и не может быть изменена при вызове других графических функций. В первую очередь это касается параметров, определяющих разбиение графического устройства на несколько панелей (подграфиков) и размер полей вокруг графика.

Разбиение графического устройства на панели определяется параметрами `mfrow` и `mfcol`. Параметр `mfrow` представляет собой вектор из двух целых чисел, определяющих на сколько строк и столбцов нужно разбить графическое устройство. По умолчанию этот параметр имеет значение `c(1,1)`, то есть графическое устройство на панели не делится. Если задать `mfrow = c(2,1)` графическое устройство будет разделено на две панели, расположенные одна под другой (2 строки, один столбец). Панели будут заполняться графиками последовательно в порядке вызова функций, создающих новый график. При использовании параметра `mfrow` панели будут заполняться построчно. Параметр `mfcol` действует абсолютно аналогично, но при его использовании панели будут заполняться по столбцам. После заполнения всех незанятых панелей на текущем графике, будет создан следующий.

Размер полей вокруг графика контролируется параметром `mar`, который представляет собой вектор из четырех элементов, определяющих число строк текста соответственно снизу, слева, сверху и справа от графика (по часовой стрелке снизу). По умолчанию параметр `mar` имеет значение `c(5.1, 4.1, 4.1, 2.1)`. Необходимость менять этот параметр возникает, когда нужно использовать нестандартные подписи или надписи на полях, а также при комбинировании графиков в панелях одного графического устройства.

Рассмотрим конкретный пример, в котором используем классический набор данных по характеристикам цветков ириса трех видов: *Iris setosa*, *I. versicolor* и

I. virginica. Для каждого из видов было собрано 50 цветков и измерены длина и ширина лепестков (переменные `Sepal.Height` и `Sepal.Width`) и чашелистиков (переменные `Petal.Height` и `Petal.Width`).

```
> data(iris)
```

Фрейм содержит 150 строк, при этом данные по видам следуют один за другим и принадлежность к разным таксонам задается текстовой переменной `Species`, в которой указаны видовые названия. Построим графики зависимости ширины лепестков от их длины отдельно для трех видов ирисов. Для удобства сопоставления расположим их один под другим. Для этого необходимо воспользоваться параметром `mfrow`, с помощью которого укажем, что графическое устройство необходимо разбить на три строки, сохранив один столбец. Подкорректируем также значение параметра `mar` так, чтобы меньше места занимали поля сверху и справа от графика. При изменении графических параметров рекомендуется сохранить текущие значения в отдельном объекте, чтобы потом можно было быстро их восстановить (функция `par()` возвращает исходные значения тех параметров, которые изменяются при ее вызове, в виде списка).

```
> oldPar <- par(mfrow = c(3, 1), mar = c(4, 4, 2, 1))
> plot(iris$Sepal.Length[1:50], iris$Sepal.Width[1:50],
+   main = "Iris setosa", xlab = "", ylab = "")
> plot(iris$Sepal.Length[51:100], iris$Sepal.Width[51:100],
+   main = "Iris versicolor", xlab = "", ylab = "Ширина лепестка, см")
> plot(iris$Sepal.Length[101:150], iris$Sepal.Width[101:150],
+   main = "Iris virginica", xlab = "Длина лепестка, см", ylab = "")
```

В данном случае мы воспользовались тем фактом, что во фрейме `iris` данные по разным видам сгруппированы, так что первые 50 наблюдений относятся к виду *I. setosa*, наблюдения с 51 по 100 – к виду *I. versicolor*, а последние 50 – к виду *I. virginica*. Мы также опустили подписи оси абсцисс у двух верхних графиков, а подпись оси ординат сделали только для второго графика, что улучшило читаемость. Итоговый график представлен на Рис. 4.

Возврат к старым значениям параметров осуществляется следующим образом:

```
> par(oldPar)
```

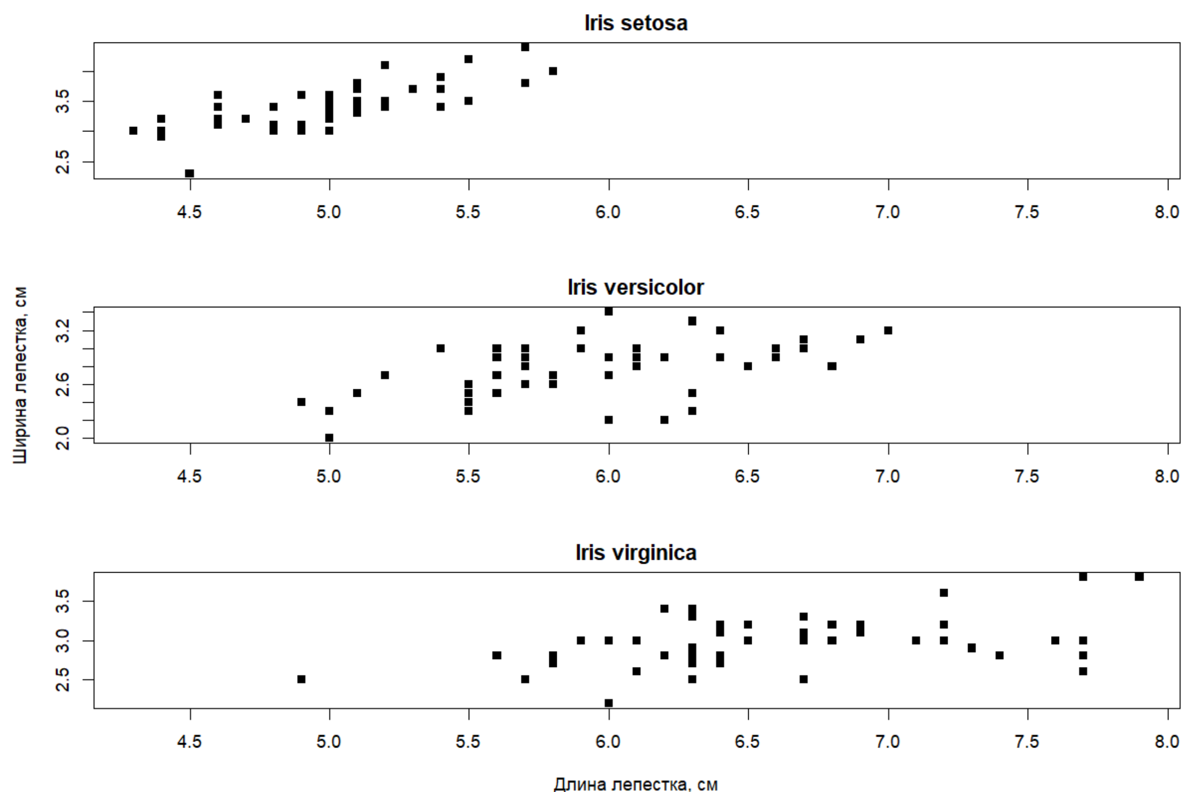

Рис. 4. Многопанельный график

Таблица 2

Основные графические параметры

Параметр	Описание	Запоминание	Значение по умолчанию
`adj`	Контролирует выравнивание текста в строках `text`, `mtext` и `title`. Установите `adj = 0` для левого выравнивания, `adj = 1` для правого выравнивание и `adj = 0.5` для центрирования.	ADJust – регулировать	`0.5`
`ann`	Если `ann = TRUE`, заголовки осей и всего графика отображаются, если `ann = FALSE`, эти аннотации опускаются.	ANNotation – аннотация	`TRUE`
`ask`	При работе в интерактивной сессии, если `ask = TRUE`, то перед отрисовкой нового графика информация запрашивается у пользователя.	ASK – запрашивать	`FALSE`

Параметр	Описание	Запоминание	Значение по умолчанию
bg	Цвет фона графического устройства.	BackGround – фон	transparent
bty	Тип рамки вокруг графика. Используйте bty = "o" для рамок со всех сторон, bty = "l" для рамок только слева и снизу, bty = "7" для рамок только справа и сверху, bty = "]" для рамок сверху, справа и снизу, bty = "c" для рамок сверху, слева и снизу, bty = "u" для рамок слева, снизу и справа, bty = "n" для отсутствия рамки. Рамку можно нарисовать с помощью функции box().	Box TYpe – тип рамки	o
cex	Контролирует размер текста и отображаемых символов на графике. cex = 1 означает «нормальный размер», cex = 0.75 означает «уменьшить текст и символы до 75 % нормального размера».	Character EXpansion – увеличение символов	1
cex.axis	Увеличение надписей на осях относительно cex.		1
cex.lab	Увеличение подписей осей относительно cex.		1
cex.main	Увеличение текста основного заголовка относительно cex.		1
cex.sub	Увеличение текста дополнительного заголовка относительно cex.		1
cin	Размер символов в дюймах (эквивалентно cra, но в другой размерности).	Character size in INches – размер символов в дюймах	c(0.15,0.2)
col	Цвет графика по умолчанию.	COLor – цвет	black
col.axis	Цвет аннотации оси.		black
col.lab	Цвет подписей осей.		black
col.main	Цвет основного заголовка.		black
col.sub	Цвет дополнительного заголовка.		black

Параметр	Описание	Запоминание	Значение по умолчанию
cra	Размер символов в пикселях.	Character size in RAsters – размер символов в растрах	c(10.8,14.4)
crt	Числовое значение в градусах, определяющее вращение отдельных символов (см. srt для вращения целых строк).	Character RoTation – вращение символов	0
csi	Высота символов по умолчанию в дюймах.		0.2
cxy	Размер символов по умолчанию в координатных единицах пользователя.		c(0.02181950, 0.07431491)
din	Размер графического устройства в дюймах.	Dimension in INches – размерность в дюймах	c(7,7)
family	Название семейства шрифтов при отображении текста. Часто используются шрифты "serif", "mono", "sans", и "symbol".	FAMILY – семейство	
fg	Цвет переднего плана.	ForeGround – передний план	Black
fig	Числовой вектор, определяющий координаты области отображения фигуры.	FIGure – фигура	c(0,1,0,1)
fin	Размер области отображения фигуры в дюймах.		c(7,7)
font	Целое число, определяющее тип шрифта. font = 1 соответствует обычному шрифту, font = 2 соответствует полужирному, font = 3 соответствует курсиву, font = 4 соответствует полужирному курсиву, а font = 5 соответствует замене на шрифт Adobe Symbol.	FONT – шрифт	1
font.axis	Тип шрифта для аннотации осей.		1
font.lab	Тип шрифта для подписей осей.		1

Параметр	Описание	Запоминание	Значение по умолчанию
font.main	Тип шрифта для основного заголовка.		2
font.sub	Тип шрифта для дополнительного заголовка.		1
lab	Числовой вектор из трех элементов (x,y,len), которые определяют, как аннотируются оси. x и y определяют примерное число отсечек на осях абсцисс и ординат соответственно, а len определяет дину подписи.	LABel – ярлык, этикетка	c(5,5,7)
las	Определяет направление подписей осей. las = 0 параллельно оси, las = 1 горизонтально, las = 2 перпендикулярно оси, las = 4 вертикально.		0
lend	Определяет стиль окончания линий. lend = 0 или lend = "round" соответствует закругленным окончаниям, lend = 1 или lend = "butt" соответствует отсутствию окончаний, lend = 2 или lend = "square" соответствует квадратным окончаниям.	Line END – конец линии	round
lheight	Множитель, определяющий высоту строки при форматировании нескольких строк текста.	Line HEIGHT – высота линии	1
ljoin	Определяет стиль стыков линий. ljoin = 0 или ljoin = "round" соответствует закругленным стыкам, ljoin = 1 или ljoin = "mitre" соответствует заостренным стыкам, ljoin = 2 или ljoin = "bevel" соответствует плоским стыкам.	Line JOIN – соединение линий	Round

Параметр	Описание	Запоминание	Значение по умолчанию
`lmitre`	Определяет максимальный размер заостренного стыка, в случае превышения которого заостренный стык линий преобразуется в плоский. Не все графические устройства поддерживают этот параметр.	Line MITRE – скос линий	`10`
`lty`	Тип линии. Можно использовать числовые обозначения либо текстовые (0 либо "blank" – пустой, 1 либо "solid" – сплошной, 2 либо "dashed" – штриховой, 3 либо "dotted" – пунктирный, 4 либо "dotdash" – штрихпуктирный, 5 либо "longdash" – длинные штрихи, 6 либо "twodash" – двойные штрихи).	Line TYpe – тип линий	`solid`
`lwd`	Положительное число, определяющее толщину линий.	Line WiDth – ширина линий	`1`
`mai`	Числовой вектор из четырех элементов, определяющий размер полей вокруг графика в дюймах (снизу по часовой стрелке).	MArgin size in Inches – размер полей в дюймах	`c(1.02,0.82, 0.82,0.42)`
`mar`	Числовой вектор из четырех элементов, определяющий размер полей вокруг графика в строках текста (снизу по часовой стрелке).	MARgin size in lines – размер полей в строках	`c(5.1,4.1, 4.1,2.1)`
`mex`	Множитель, используемый при определении размера строк на полях графика (`mai = mar * mex * csi`).		`1`

Параметр	Описание	Запоминание	Значение по умолчанию
mfcol	Этот параметр позволяет разделить графическое устройство на «матрицу» подграфиков. Это числовой вектор из двух элементов c(nrows, ncols), где первый элемент – это число строк, а второй – это число столбцов. Ячейки «матрицы» заполняются по столбцам.		c(1,1)
mfg	В ходе заполнения «матрицы» подграфиков этот параметр контролирует положение следующей позиции. Это числовой вектор из четырех элементов c(row, col, nrows, ncols), первые два определяют номера строки и столбца, последние два – общее число строк и столбцов соответственно.		c(1,1,1,1)
mfrow	Эквивалент параметра mfcol, но ячейки заполняются построчно.		c(1,1)
oma	Числовой вектор из четырех элементов, определяющий размер внешних полей вокруг графика в строках текста (снизу по часовой стрелке). Внешние поля располагаются вокруг матрицы подграфиков.	Outer MArgin in lines – внешнее поле в строках	c(0,0,0,0)
omd	Числовой вектор из четырех элементов, определяющий размер внешних полей как доля размера графического устройства (снизу по часовой стрелке).	Outer Margin in Device coordinates – внешнее поле в координатах устройства	c(0,1,0,1)
omi	Числовой вектор из четырех элементов, определяющий размер внешних полей вокруг графика в дюймах (снизу по часовой стрелке).	Outer Margin in inches – внешнее поле в дюймах	c(0,0,0,0)

Параметр	Описание	Запоминание	Значение по умолчанию
pch	Тип символа. Может быть задан либо в виде числа (0-14 – незалитые фигуры, 15-20 – залитые фигуры, 21-25 – залитые фигуры с границей, 32-127 – символы ASCII), либо в виде символа.	Point CHaracter – точечный символ	1
pin	Размерность текущего графика в дюймах.	Plot in Inches – размер графика в дюймах	c(5.76, 5.16)
pty	Определяет тип заполнения графика. Используйте `pty = "s"` для квадратного графика, либо `pty = "m"` для максимального заполнения пространства графического устройства.		"m"
srt	Определяет угол вращения текста в градусах. Поддерживается только функцией `text()`.	String RoTation – вращение строки	0
tck	Длина отсечек на осях, выраженная в долях от наименьшей размерности (длины либо ширины) графика. Если `tck > 0.5`, длина интерпретируется как доля соответствующей стороны (соответственно в случае `tck = 1` на график накладывается сетка).	TiCK – отсечка	NA
tcl	Длина отсечек на осях, выраженная в долях от высоты строки текста.		-0.5

Параметр	Описание	Запоминание	Значение по умолчанию
xaxp	Определяет формат отображения отсечек на оси абсцисс. Задается в форме вектора вида c(x1, x2, n). При использовании линейного масштаба определяет минимальное (x1) и максимальное положения (x2) отсечек, а также их общее количество (n). При использовании логарифмического масштаба x1 – это наименьшая степень 10, x2 – наибольшая степень 10, а n – это число отсечек для каждой степени 10.	X-AXis tick Position – положение отсечек на оси абсцисс	c(0, 1, 5)
xaxs	Определяет метод задания интервала отображения по оси абсцисс. При xaxs = "r" диапазон варьирования данных расширяется на 4 % с каждой стороны и подбираются хорошо выглядящие отсечки. При xaxs = "i" используется диапазон варьирования данных без расширения.	X-AXis Style – стиль оси абсцисс	r
xaxt	Определяет тип оси абсцисс. При xaxt = "n" соответствует отсутствию оси, любое другое значение отображает ось.	X-AXis Type – тип оси абсцисс	s
xlog	Логическое значение, определяющее переход к логарифмическому масштабу по оси абсцисс.		FALSE
xpd	Логическое значение либо NA, определяет границу, по которой будут обрезаны отображаемые элементы (FALSE – по границе области отображения, TRUE – по границе всего графика, NA – по границе графического устройства).		

Параметр	Описание	Запоминание	Значение по умолчанию
yaxp	Определяет формат отображения отсечек на оси ординат. См. xaxp.	Y-AXis tick Position – положение отсечек на оси ординат	c(0, 1, 5)
yaxs	Определяет метод задания интервала отображения по оси ординат. См. xaxs.	Y-AXis Style – стиль оси ординат	r
yaxt	Определяет тип оси абсцисс. См. xaxt.	Y-AXis Type – тип оси ординат	s
ylog	Логическое значение, определяющее переход к логарифмическому масштабу по оси ординат.		FALSE

10.5. Графические устройства

При создании нового графика (например, функцией `plot()`) он помещается в графическое устройство, по умолчанию – просто в окно (или вкладку при работе в RStudio) для отображения на экране компьютера. При подготовке отчетов и публикаций часто бывает необходимо получить иллюстрации в виде файлов в конкретном графическом формате (например, jpeg определенного размера и разрешения). Интерфейс RStudio позволяет сохранять созданные в окне графики в виде файлов, нужно просто воспользоваться кнопкой Export на панели Plots, выбрать необходимый тип файла, директорию и имя файла, а также подобрать необходимый размер.

В целях автоматизации процесса создания иллюстраций часто бывает удобнее воспользоваться средствами языка R для создания и сохранения графических файлов. При создании графика нужно открыть новое графическое устройство необходимого типа, после чего результаты выполнения всех графических функций будут перенаправлены в данное устройство. После завершения создания графика устройство необходимо закрыть.

Для открытия нового графического устройства служат функции `windows()`, `pdf()`, `postscript()`, `xfig()`, `bitmap()`, `png()`, `tiff()`, `bmp()`, `jpeg()`. Эти функции создают графические устройства соответствующего названию типа. Основными аргументами являются `filename` – имя файла (за исключением функции

`windows()`, которая создает новое окно), `width` и `height` – соответственно ширина и высота изображения в пикселях. Помимо указанных, каждая из перечисленных функций обладает набором специфических аргументов, информацию о которых можно найти в справочной системе. Для закрытия графического устройства используется функция `dev.off()`.

В случае необходимости сохранения уже созданного в окне графика можно воспользоваться функцией `dev.copy(device, ...)`, аргумент `device` указывает имя функции (без кавычек), создающей графическое устройство, а `...` – аргументы, которые будут переданы этой функции. В процессе копирования будет создано новое графическое устройство, которое также необходимо закрыть функцией `dev.off()`.

В качестве примера создадим простой график зависимости ширины лепестков от их длины для трех видов ирисов и сохраним его в файле формата jpeg.

```
> jpeg(file = "iris.jpg", width = 640, height = 480, quality = 80)
```

После выполнения этого выражения создается графическое устройство, куда будут перенаправлены результаты выполнения графических функций, – файл iris.jpg, который появляется в рабочей директории (в данный момент он пустой). После открытия устройства создаем сам график:

```
> plot(iris$Sepal.Length,iris$Sepal.Width)
```

После выполнения этого выражения на экране ничего не появляется, поскольку вывод перенаправлен в файловое графическое устройство. Для того чтобы можно было работать с новым файлом, устройство необходимо закрыть:

```
> dev.off()
RStudioGD
        2
```

Функция `dev.off()` возвращает имя и номер текущего активного графического устройства, куда будут направлены результаты следующего использования графических функций. В нашем случае `RStudioGD` указывает на то, что открыто графическое окно RStudio, причем не одно (на это указывает номер 2).

Если мы теперь обратимся к созданному файлу iris.jpg, то обнаружим построенный график заданного размера и качества (аргумент `quality` функции `jpeg()` определяет качество при сжатии изображения). При этом сам график построен с использованием набора графических параметров по умолчанию, в данном случае – с использованием незаполненных кругов стандартного размера (несмотря на то, что мы изменили тип символа с использованием функции `par()`). Дело в том, что каждое графическое устройство в R имеет свой

собственный набор графических параметров, поэтому при необходимости их изменения нужно вызывать функцию `par()` тогда, когда устройство уже создано, но еще не закрыто (в приведенном выше примере – между вызовами функций `jpeg()` и `dev.off()`).

Василий Николаевич **Якимов**

Основы анализа биомедицинских и экологических данных в среде R

Часть 1

Учебное пособие

Федеральное государственное автономное образовательное учреждение
высшего образования
«Национальный исследовательский Нижегородский государственный университет
им. Н.И. Лобачевского»
603950, Нижний Новгород, пр. Гагарина, 23.
Подписано в печать . Формат 60×84 1/16.
Бумага офсетная. Печать офсетная. Гарнитура Таймс.
Усл. печ. л. 12,5. Уч.-изд. л.
Заказ № . Тираж 100 экз.

Отпечатано в типографии Нижегородского государственного
университета им. Н.И. Лобачевского
603600, г. Нижний Новгород, ул. Большая Покровская, 37