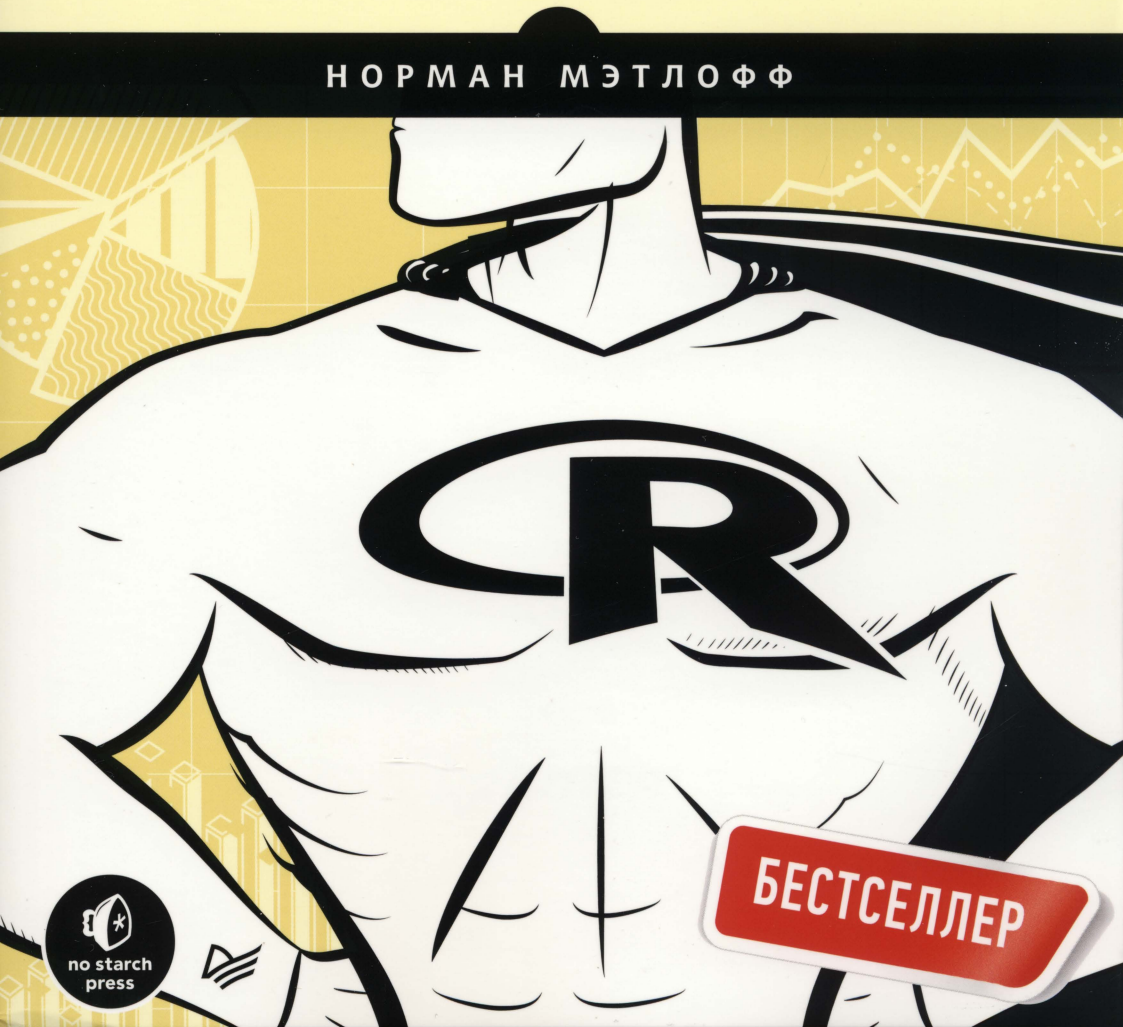
ИСКУССТВО ПРОГРАММИРОВАНИЯ НА R

ПОГРУЖЕНИЕ В БОЛЬШИЕ ДАННЫЕ

НОРМАН МЭТЛОФФ



no starch press

THE ART OF R PROGRAMMING

A Tour of Statistical Software Design

by Norman Matloff

no starch
press
San Francisco

НОРМАН МЭТЛОФФ

ИСКУССТВО ПРОГРАММИРОВАНИЯ НА R

ПОГРУЖЕНИЕ В БОЛЬШИЕ ДАННЫЕ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2019

ББК 32.973.2-018.1
УДК 004.43
М97

Мэтлофф Норман

М97 Искусство программирования на R. Погружение в большие данные. — СПб.: Питер, 2019. — 416 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1101-5

R является самым популярным в мире языком статистических вычислений: археологи используют его, изучая древние цивилизации, фармацевтические компании выясняют, какие лекарства наиболее безопасны и эффективны, а финансисты задействуют его для оценки рисков и удержания позиций на рынке.

«Искусство программирования на R» — это путешествие, в которое вы отправляетесь с опытным гидом, готовым поделиться всей информацией о разработке ПО: от типов и структур данных до таких продвинутых тем, как замыкания, рекурсия и анонимные функции. Вам не понадобятся специальные знания в области статистики, а программистский опыт может варьироваться от начинающего до профессионала. Вы познакомитесь с функциональным и объектно-ориентированным программированием, математическим моделированием и преобразованием сложных данных в простые и удобные форматы.

Проектируете ли вы самолет, прогнозируете ли вы... погоду, или просто хотите «приручить» свои данные, «Искусство программирования на R» станет руководством по использованию всей мощи статистических вычислений.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.1
УДК 004.43

ISBN 978-1593273842 англ.

ISBN 978-5-4461-1101-5

Краткое содержание

Оглавление

Отзывы о книге

«Если вы захотите освоить язык R и стать компетентным программистом на R, то <...> вам не найти лучшего пособия, чем "Искусство программирования на R. Погружение в большие данные" Нормана Мэтлоффа».

— Джозеф Рикерт (Joseph Rickert, Revolution Analytics)

«Я порекомендую эту книгу каждому, кто хочет изучить R, особенно людям, которые разбираются в программировании лучше, чем в статистике».

— Джон Д. Кук (John D. Cook), «The Endeavor»

«Превосходно от первой до последней страницы. Достаточно глубоко, чтобы даже опытные пользователи R узнали для себя что-то полезное ближе к концу книги».

— Джон Грэм-Камминг (John Graham-Cumming)

«Если вы серьезно относитесь к изучению R <...> книга "Искусство программирования на R" будет вам безусловно полезна».

— Паоло Сонего (Paolo Sonego), «One R Tip A Day»

«Упрощает картину для тех, кто хочет строить численные модели на основании статистического анализа. Серьезный материал как для уже программирующих на R, так и для начинающих».

— Хэнк Кэмпбелл (Hank Campbell), «Science 2.0»

«Если вы хотите заниматься программированием в области статистики, я рекомендую купить эту книгу».

— Брайан Белл (Bryan Bell), «Math And More»

«Книга по программированию на R, которая начинается с основ. Если вы хотя бы отдаленно представляете, что такое программирование, книга "Искусство программирования на R" вам пригодится. Я оставлю ее на своей полке».

— Нейтан Яо (Nathan Yau), Flowingdata.Com, автор книги «Visualize This»

Благодарности

В основу этой книги легло множество полезных источников.

В первую очередь я должен поблагодарить научного редактора Хэдли Уикхэма (Hadley Wickham), известного благодаря `ggplot2` и `plyr`. Я порекомендовал Хэдли издательству «No Starch Press» из-за его опыта в разработке этих и других популярных пакетов R в CRAN — репозитории кода R, опубликованного пользователями. Как и ожидалось, многие комментарии Хэдли привели к улучшению текста, особенно комментарии в отношении конкретных примеров кода, часто начинавшиеся словами: «Интересно, а что, если написать это вот так...» В некоторых случаях эти комментарии привели к тому, что пример с одной-двумя версиями кода в итоге демонстрировал два, три, а иногда даже четыре разных способа достижения цели написания кода. Это позволило сравнить преимущества и недостатки разных решений, что, как я полагаю, будет поучительно для читателя.

Я очень благодарен Джиму Порзаку (Jim Porzak), соучредителю группы «Bay Area useR Group» (BARUG, http://www.bay-r.org/), за его постоянную поддержку во время моей работы над книгой. И раз уж речь зашла о BARUG, я должен поблагодарить Джима и другого соучредителя, Майка Дрисколла (Mike Driscoll), за создание этого живого форума, стимулирующего творческую деятельность. После знакомства с людьми, рассказывавшими в BARUG о возможностях R, у меня всегда было чувство, что эта книга была достойным проектом.

Группа BARUG также получала финансовую поддержку от Revolution Analytics; Дэвид Смит (David Smith) и Джо Рикерт (Joe Rickert) из этой компании посвятили ей бесчисленные часы, свою творческую энергию и идеи.

Джей Эмерсон (Jay Emerson) и Майк Кейн (Mike Kane), авторы признанного пакета `bigmemory` в CRAN, прочитали раннюю версию главы 16, посвященной параллельному программированию на R, и сделали ряд полезных замечаний.

Джон Чемберс (John Chambers) (создатель языка S, «предка» R) и Мартин Морган (Martin Morgan) поделились советами, касающимися внутреннего устройства R; эти советы очень пригодились мне при обсуждении проблем быстродействия R в главе 14.

В разделе 7.8.4 рассматривается тема, вызывающая ожесточенные споры в сообществе программирования, — использование глобальных переменных. Чтобы представить широкий спектр точек зрения, я воспользовался мнением нескольких людей, среди которых я хочу отметить участника базовой группы R Томаса Ламли (Thomas Lumley) и своего коллегу по Калифорнийскому университету в Дейвисе Шона Дейвиса (Sean Davis). Разумеется, это вовсе не означает, что мы разделяем их мнение в этом разделе книги, но их комментарии были весьма полезными.

На ранней стадии работы проекта я опубликовал очень приблизительный (и далеко не полный) черновик книги для открытого обсуждения. Со мной поделились своим полезным мнением Рамон Диас-Уриарте (Ramon Diaz-Uriarte), Барбара Ф. Ла Скала (Barbara F. La Scala), Джейсон Ляо (Jason Liao) и мой старый друг Майк Хэннон (Mike Hannon). Моя дочь Лаура, студентка инженерной специальности, прочитала отдельные части ранних набросков глав и поделилась полезными советами, которые позволили мне улучшить книгу. Моим собственным проектам CRAN и другим исследованиям в области R (которые послужили основой для примеров книги) принесли пользу советы, обратная связь и/или поддержка многих людей; прежде всего это были Марк Бравингтон (Mark Bravington), Стивен Эглен (Stephen Eglen), Дирк Эдделбуэт (Dirk Eddelbuett), Джей Эмерсон (Jay Emerson), Майк Кейн (Mike Kane), Гэри Кинг (Gary King), Дункан Мердок (Duncan Murdoch) и Джо Рикерт (Joe Rickert).

Участник базовой группы R Дункан Темпл Лэнг (Duncan Temple Lang) работает в той же организации, что и я, — Калифорнийском университете в Дейвисе. Хотя мы работаем на разных факультетах и общались не так уж много, эта книга кое-чем обязана его присутствию в университетском городке. Он помог сформировать в Калифорнийском университете культуру R, что помогло мне оправдать большие затраты времени на работу над книгой на моем факультете.

Это мой второй проект в издательстве «No Starch Press». Как только я решил написать эту книгу, я обратился в «No Starch Press», потому что мне нравится неформальный стиль, высокая практичность и доступность их продуктов. Спасибо Биллу Поллоку (Bill Pollock) за утверждение проекта, сотрудникам издательства Кейт Фенчер (Keith Fancher) и Элисон Лоу (Alison Law), а также внештатному редактору Мэрилин Смит (Marilyn Smith).

Наконец, я хочу поблагодарить двух прекрасных, умных и интересных женщин — мою жену Гэмис и упоминавшуюся выше Лауру. Они обе спокойно принимали мой ответ «Я пишу книгу по R» каждый раз, когда спрашивали, почему я так погружен в работу.

Введение

R — язык сценариев для обработки и анализа статистических данных. Он создавался по образцу статистического языка S, разработанного в компании AT&T (и в основном совместим с ним). Название S (от «Statistics») было аллюзией на другой язык программирования с однобуквенным именем, разработанный в AT&T, — знаменитый язык C. Позднее технология S была продана меньшей компании, которая добавила графический интерфейс (GUI) и назвала полученный продукт S-Plus.

Язык R стал более популярным, чем S или S-Plus, потому что он распространялся бесплатно, а в его разработке участвовало больше людей. R также иногда называют GNU S, чтобы намекнуть на специфику проекта. (GNU Project — огромная коллекция продуктов с открытым исходным кодом.)

Зачем использовать R в статистических вычислениях?

Как говорят жители Кантона: *yauh peng, yauh leng*, что означает «недорого и красиво». Лучше спросить: зачем использовать что-то другое?

R обладает целым рядом достоинств:

- R представляет собой открытую реализацию признанного статистического языка S, а платформа R/S стала фактическим стандартом среди профессиональных статистиков.
- По своей мощи он сравним с коммерческими продуктами (а часто и превосходит их в большинстве практических аспектов — разнообразии поддерживаемых операций, программируемости, средствах графического вывода информации и т. д.).
- Доступны версии для операционных систем Windows, Mac и Linux.
- R не ограничивается выполнением статистических операций — это язык программирования общего назначения, который может использоваться

для автоматизации анализа данных и создания новых функций, расширяющих возможности языка.

- R обладает возможностями, присущими объектно-ориентированным и функциональным языкам программирования.
- Система сохраняет данные между сеансами, поэтому вам не придется перезагружать их снова и снова. Также сохраняется история команд.
- Так как R распространяется с открытым исходным кодом, вы сможете легко получить помощь от сообщества пользователей. Кроме того, новые функции создаются пользователями, многие из которых являются известными специалистами в области статистики.

Хочу сразу предупредить, что обычно для ввода команд R пользователь вводит текст в окне терминала, а не работает с мышью в графическом интерфейсе; большинство пользователей R графический интерфейс не используют. Это не означает, что R не обладает графическими возможностями. Напротив, в R имеются средства для построения чрезвычайно полезных и эффектных графических изображений, но они используются для вывода результатов работы системы (например, диаграмм), а не для пользовательского ввода.

Если вы решительно не можете обойтись без графического интерфейса, выберите одну из бесплатных графических оболочек, созданных для R. Несколько примеров таких продуктов — с открытым кодом или бесплатных:

- RStudio, http://www.rstudio.org/;
- StatET, http://www.walware.de/goto/statet/;
- ESS (Emacs Speaks Statistics), http://ess.r-project.org/;
- R Commander: Джон Фокс (John Fox), «The R Commander: A Basic-Statistics Graphical Interface to R», Journal of Statistical Software 14, №9 (2005):1–42;
- JGR (Java GUI for R), http://cran.r-project.org/web/packages/JGR/index.html.

Первые три продукта — RStudio, StatET и ESS — представляют собой интегрированные среды разработки (IDE), предназначенные скорее для программирования. StatET и ESS предоставляют программисту R средства разработки в известных средах Eclipse и ESS соответственно.

В области коммерческих предложений еще одну IDE предлагает Revolution Analytics, обслуживающая компания R (http://www.revolutionanalytics.com/). Так как R является языком программирования, а не набором разрозненных команд, вы можете объединять команды в цепочку; при этом выходные данные одной команды используются в качестве входных данных другой (эта возможность

хорошо знакома пользователям Linux, привыкшим объединять команды оболочки при помощи *каналов*, или *конвейеров* (pipes)). Механизм объединения функций R обладает огромной гибкостью, и при правильном использовании он весьма мощен. Простой пример: возьмем следующую (составную) команду:

```
nrow(subset(x03,z == 1))
```

Сначала функция `subset()` получает кадр данных `x03` и извлекает все записи, у которых переменная `z` равна 1. При этом создается новый кадр данных, который затем передается функции `nrow()`. Эта функция подсчитывает количество строк в кадре. В итоге команда выдает количество строк, для которых `z = 1`, в исходном кадре данных.

Ранее упоминались термины «объектно-ориентированное программирование» и «функциональное программирование». Эти темы сейчас широко обсуждаются специалистами в области теории вычислений. Несмотря на то что другим читателям эти термины могут быть незнакомы, они актуальны для всех, кто применяет R для программирования статистических вычислений. В следующем разделе предоставляются краткие обзоры этих тем.

Объектно-ориентированное программирование

Преимущества объектно-ориентированного программирования проще пояснить на конкретном примере. Возьмем статистическую регрессию. При проведении регрессионного анализа с использованием других статистических пакетов (например, SAS или SPSS) на экран выводится огромный объем информации. Напротив, при вызове регрессионной функции `lm()` в R функция возвращает объект, содержащий все результаты — оценки коэффициентов, их стандартные погрешности, остатки и т. д. Вам остается выбрать (на программном уровне), какие части извлечь из объекта.

Вы увидите, что подход R существенно упрощает программирование — отчасти потому, что он обеспечивает определенное единообразие работы с данными. Это однообразие происходит от того факта, что R является *полиморфным* языком; другими словами, одна функция может использоваться для разных типов входных данных, для которых выбирается подходящий способ обработки. Такие функции называются *обобщенными* (программистам C++ знакома похожая концепция виртуальных функций).

Например, возьмем функцию `plot()`. Если вызвать ее для списка чисел, вы получите простой график. Но если вызвать ее для выходных данных регрессионного анализа, вы получите набор графиков, представляющих различные аспекты

анализа. Собственно, функция `plot()` может использоваться практически с любым объектом, создаваемым в R. И это удобно — ведь вам как пользователю придется запоминать меньше команд!

Функциональное программирование

Как характерно для языков функционального программирования, в программировании на R часто встречается тема *предотвращения явного программирования итераций*. Вместо того чтобы программировать циклы, вы при помощи функциональных средств R выражаете итеративное поведение неявно. В результате можно получить код, который выполняется намного эффективнее, ведь при обработке больших наборов данных в R затраты времени могут быть весьма значительными.

Как вы вскоре увидите, природа функционального программирования языка R обладает рядом преимуществ:

- более четкий и компактный код;
- возможность многократного ускорения выполнения кода;
- снижение затрат времени на отладку из-за упрощения кода;
- упрощение перехода на параллельное программирование.

Для кого написана эта книга?

Многие пользователи используют R для конкретных задач — тут построить гистограмму, там провести регрессионный анализ или выполнить другие отдельные операции, связанные со статистической обработкой данных. Но эта книга написана для тех, кто хочет разрабатывать программное обеспечение на R. Навыки программирования предполагаемых читателей этой книги могут лежать в широком спектре — от профессиональной квалификации до «Я проходил курс программирования в колледже», но ключевой целью является написание кода R для конкретных целей. (Глубокое знание статистики в общем случае не обязательно.)

Несколько примеров читателей, которые могли бы извлечь пользу из этой книги:

- Аналитик (допустим, работающий в больнице или в правительственном учреждении), которому приходится регулярно выдавать статистические отчеты и разрабатывать программы для этой цели.

- Научный работник, занимающийся разработкой статистической методологии — новой или объединяющей существующие методы в интегрированные процедуры. Методологию нужно закодировать, чтобы она могла использоваться в сообществе исследователей.
- Специалисты по маркетингу, судебному сопровождению, журналистике, издательскому делу и т. д., занимающиеся разработкой кода для построения сложных графических представлений данных.
- Профессиональные программисты с опытом разработки программного обеспечения, назначенные в проекты, связанные со статистическим анализом.
- Студенты, изучающие статистику и обработку данных.

Таким образом, эта книга не является справочником по бесчисленным статистическим методам замечательного пакета R. На самом деле она посвящена программированию и в ней рассматриваются вопросы программирования, редко встречающиеся в других книгах о R. Даже основополагающие темы рассматриваются под углом программирования. Несколько примеров такого подхода:

- В этой книге встречаются разделы «Расширенные примеры». Обычно в них представлены полные функции общего назначения вместо изолированных фрагментов кода, основанных на конкретных данных. Более того, некоторые из этих функций могут пригодиться в вашей повседневной работе с R. Изучая эти примеры, вы не только узнаете, как работают конкретные конструкции R, но и научитесь объединять их в полезные программы. Во многих случаях я привожу описания альтернативных решений и отвечаю на вопрос: «Почему это было сделано именно так?»
- Материал излагается с учетом восприятия программиста. Например, при описании кадров данных я не только утверждаю, что кадр данных в R представляет собой список, но и указываю на последствия этого факта с точки зрения программирования. Также в тексте R сравнивается с другими языками там, где это может быть полезно (для читателей, владеющих этими языками).
- Отладка играет важнейшую роль в программировании на любом языке, однако в большинстве книг о R эта тема практически не упоминается. В этой книге я посвятил средствам отладки целую главу, воспользовался принципом «расширенных примеров» и представил полностью проработанные демонстрации того, как происходит отладка программ в реальности.
- В наши дни многоядерные компьютеры появились во всех домах, а программирование графических процессоров (GPU) производит незаметную революцию в области научных вычислений. Все больше приложений R требует очень больших объемов вычислений, и параллельная обработка

стала актуальной для программистов на R. В книге этой теме посвящена целая глава, в которой помимо описания механики также приводятся расширенные примеры.

- Отдельная глава рассказывает о том, как использовать информацию о внутренней реализации и других аспектах R для ускорения работы кода R.
- Одна из глав посвящена интерфейсу R с другими языками программирования, такими как C и Python. И снова особое внимание уделяется расширенным примерам и рекомендациям по выполнению отладки.

Немного о себе

Я пришел в мир R несколько необычно.

После написания диссертации по абстрактной теории вероятностей я провел ранние годы своей карьеры на должности профессора статистики — за преподаванием, научными исследованиями и консультациями по статистической методологии. Я был одним из дюжины профессоров Калифорнийского университета в Дейвисе, основавших там факультет статистики.

Позднее я перешел на факультет Computer Science в том же университете, где прошла большая часть моей карьеры. Я занимался исследованиями в области параллельного программирования, анализа веб-трафика, глубокого анализа данных, производительности дисковой системы и во многих других областях. Большая часть моей преподавательской и исследовательской работы была связана со статистикой.

Таким образом, я могу рассматривать ситуацию как с точки зрения опытного специалиста по компьютерным технологиям, так и с точки зрения статистика и ученого-исследователя в области статистики. Надеюсь, такое сочетание позволит этой книге заполнить пробел в литературе и сделает ее более ценной для вас, уважаемый читатель.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция). Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Первые шаги

R — чрезвычайно универсальный язык программирования с открытым исходным кодом, предназначенный для статистических расчетов и анализа данных. Он широко применяется во всех областях, в которых нам приходится иметь дело с данными, — в бизнесе, промышленности, управлении, медицине, мире науки и т. д.

В этой главе содержится краткий вводный курс по языку R: как его запустить, что он может делать и какие файлы использует. Приведенной информации достаточно ровно для того, чтобы вы могли понять примеры в следующих главах, где будут приведены подробности.

Возможно, язык R уже установлен в вашей системе, если ваш работодатель или университет предоставляет его своим пользователям. Если нет — вы найдете инструкции по установке в приложении А.

1.1. Как запустить R

R работает в двух режимах: интерактивном и пакетном (batch). На практике чаще используется интерактивный режим. В этом режиме вы вводите команды, R выводит результаты, вы вводите новые команды, и т. д. С другой стороны, пакетный режим обходится без взаимодействия с пользователем. Он может пригодиться для повседневной работы (например, если программа должна запускаться периодически — скажем, раз в сутки), потому что процесс можно автоматизировать.

1.1.1. Интерактивный режим

В системах Linux или Mac для запуска сеанса R введите команду `R` в приглашении командной строки в окне терминала. На машинах с Windows R запускается при помощи соответствующего значка.

На экран выводится приветствие и приглашение R — знак >. Экран выглядит примерно так:

```
R version 2.10.0 (2009-10-26)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
...
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Теперь все готово к выполнению команд R. Окно, в котором выводится вся эта информация, называется *консолью* R.

Короткий пример: возьмем стандартное нормальное распределение (с математическим ожиданием 0 и дисперсией 1). Если случайная переменная `X` имеет такое распределение, то ее значения сосредоточены вокруг центра 0 — одни из них положительны, другие отрицательны, а в среднем сумма равна 0. Теперь создадим новую случайную переменную `Y = |X|`. Так как мы вычисляем абсолютное значение, значения `Y` *не будут* сосредоточены вокруг 0 и математическое ожидание `Y` будет положительным.

Определим математическое ожидание `Y`. Наш подход будет основан на смоделированных случайных данных с распределением N(0,1).

```
> mean(abs(rnorm(100)))
[1] 0.7194236
```

Этот код генерирует 100 случайных переменных, находит их абсолютные значения, а затем вычисляет для них среднюю величину.

Здесь `[1]` означает, что первый элемент в этой строке вывода равен 1. В данном случае вывод состоит только из одной строки (и одного элемента), поэтому эта информация избыточна. Она становится полезной при чтении длинного вывода с множеством элементов, распределенных по многим строкам. Например, если бы в выводе было две строки по шесть элементов в каждой, то вторая строка была бы снабжена пометкой `[7]`.

```
> rnorm(10)
[1] -0.6427784 -1.0416696 -1.4020476 -0.6718250 -0.9590894 -0.8684650
[7] -0.5974668  0.6877001  1.3577618 -2.2794378
```

Здесь выходные данные состоят из 10 значений, и метка `[7]` во второй строке позволяет быстро увидеть, что 0,6877001, например, является восьмым элементом вывода.

Команды R также можно сохранить в файле. По соглашению файлы с кодом R имеют суффикс .R или .r. Если создать файл с именем z.R, то его содержимое можно будет выполнить следующей командой:

```
> source("z.R")
```

1.1.2. Пакетный режим

В некоторых ситуациях бывает удобно автоматизировать сеансы R. Например, можно запускать сценарий R для построения графика, вместо того чтобы вручную запускать R и выполнять сценарий самостоятельно. В таком случае R работает в пакетном режиме.

Например, включим код построения графика в файл с именем z.R и следующим содержимым:

```
pdf("xh.pdf") # Выбрать выходной файл
hist(rnorm(100)) # Сгенерировать 100 переменных N(0,1)
                 # и построить гистограмму
dev.off() # Закрыть выходной файл
```

За символом # следуют *комментарии*. Интерпретатор R их игнорирует. Комментарии напоминают нам и другим разработчикам, что именно делает код, в удобочитаемом формате.

Разберем шаг за шагом, что же происходит в этом коде:

- Мы вызываем функцию `pdf()`, чтобы сообщить R, что создаваемый график должен быть сохранен в PDF-файле xh.pdf.
- Вызов функции `rnorm()` (от «random normal», то есть «случайное нормальное») генерирует 100 случайных переменных с распределением N(0,1).
- Для сгенерированных переменных вызывается функция `hist()`, которая строит гистограмму полученных данных.
- Вызов `dev.off()` закрывает используемое графическое «устройство» — в данном случае файл xh.pdf. Эта операция приводит к фактической записи файла на диск.

Этот код может быть выполнен автоматически, без входа в интерактивный режим R. Запустите R командой оболочки операционной системы (например, из приглашения $, обычно используемого в системах Linux):

```
$ R CMD BATCH z.R
```

Чтобы убедиться в том, что команда была успешно выполнена, откройте сохраненную гистограмму в программе просмотра PDF-файлов (это будет очень простая гистограмма, но R также позволяет строить достаточно сложные разновидности).

1.2. Первый сеанс R

Создайте простой набор данных (в терминологии R — *вектор*), состоящий из чисел 1, 2 и 4, и присвойте ему имя `x`:

```
> x <- c(1,2,4)
```

Стандартным оператором присваивания в R является оператор `<-`. Также можно использовать оператор `=`, но лучше этого не делать, так как он не будет работать в некоторых специфических ситуациях. Обратите внимание: с переменными не связываются никакие фиксированные типы. Здесь вектор был присвоен переменной `x`, но позднее ему может быть присвоено значение другого типа. Векторы и другие типы будут рассматриваться в разделе 1.4.

Имя `c` означает *конкатенацию* (concatenate). В данном случае конкатенация выполняется с числами 1, 2 и 4. Точнее говоря, мы выполняем конкатенацию трех одноэлементных векторов, каждый из которых содержит одно из этих чисел. Дело в том, что любое число также рассматривается как вектор с одним элементом.

Теперь можно выполнить следующую команду:

```
> q <- c(x,x,8)
```

которая присваивает `q` значение `(1,2,4,1,2,4,8)` (да, включая дубликаты).

А теперь убедимся в том, что данные действительно хранятся в `x`. Чтобы вывести вектор на экран, просто введите его имя. При вводе любого имени переменной (или в более широком смысле — любого выражения) в интерактивном режиме R выведет значение этой переменной (или выражения). Эта особенность известна программистам, знакомым с другими языками (например, Python). В нашем примере это выглядит так:

```
> x
[1]124
```

Да, все верно — `x` состоит из чисел 1, 2 и 4.

Для обращения к отдельным элементам вектора используется синтаксис `[]`. Например, третий элемент `x` выводится следующей командой:

```
> x[3]
[1] 4
```

Как и в других языках, селектор (в данном случае 3) называется *индексом*. Программистам, знакомым с языками из семейства ALGOL (включая C и C++), стоит учитывать, что индексирование элементов векторов R начинается с 1, а не с 0.

Сегментация — одна из важнейших операций с векторами. Пример:

```
> x <- c(1,2,4)
> x[2:3]
[1]24
```

Выражение `x[2:3]` обозначает подвектор `x`, состоящий из элементов с 2 по 3 (в данном случае элементы со значениями 2 и 4.)

Математическое ожидание и среднеквадратическое отклонение для нашего набора данных легко вычисляются следующим образом:

```
> mean(x)
[1] 2.333333
> sd(x)
[1] 1.527525
```

Этот фрагмент снова демонстрирует, что при вводе выражения на экран выводится его значение. В первой строке выражением является вызов функции `mean(x)`. Возвращаемое значение этой функции выводится автоматически, вызывать функцию R `print()` для этого не обязательно.

Если вычисленное математическое ожидание нужно сохранить в переменной (вместо того, чтобы просто вывести его на экран), выполните следующую команду:

```
> y <- mean(x)
```

Как и прежде, убедимся в том, что `y` действительно содержит математическое ожидание `x`:

```
> y
[1] 2.333333
```

И снова в команды можно включать комментарии после символа `#`:

```
> y # Вывести значение y
[1] 2.333333
```

Комментарии особенно полезны для документирования программного кода, но в интерактивных сеансах они тоже пригодятся, поскольку R сохраняет историю команд (см. раздел 1.6). Если вы сохраните сеанс и загрузите его в будущем, комментарии помогут вспомнить, чем вы занимались.

Наконец, давайте сделаем что-нибудь с одним из внутренних наборов данных R (эти наборы предназначены для демонстраций). Чтобы получить список таких наборов, введите следующую команду:

```
> data()
```

Один из наборов данных с именем `Nile` содержит данные о течении Нила. Вычислим математическое ожидание и среднеквадратическое отклонение для этого набора данных:

```
> mean(Nile)
[1] 919.35
> sd(Nile)
[1] 169.2275
```

Также можно вывести гистограмму этих данных:

```
> hist(Nile)
```

На экране появляется окно с гистограммой (рис. 1.1). Диаграмма содержит минимум декоративных элементов, но R предоставляет массу возможностей для оформления вывода. Например, вы можете изменить количество групп при помощи переменной `breaks`. Вызов `hist(z,breaks=12)` выведет гистограмму набора данных `z` с разбиением на 12 групп. Также можно создавать более удобные метки, использовать цвета и вносить много других изменений для создания более информативных и привлекательных диаграмм. Когда у вас появится опыт работы с R, вы сможете строить сложные, яркие графики и диаграммы невероятной красоты.

Итак, первое пятиминутное знакомство с R подошло к концу. Закройте R вызовом функции `q()` (также это можно сделать нажатием клавиш Ctrl+D в Linux или Cmd+D на Mac):

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

Последний запрос предлагает сохранить переменные, чтобы вы смогли продолжить работу позднее. Если подтвердить сохранение (`y`), то все объекты

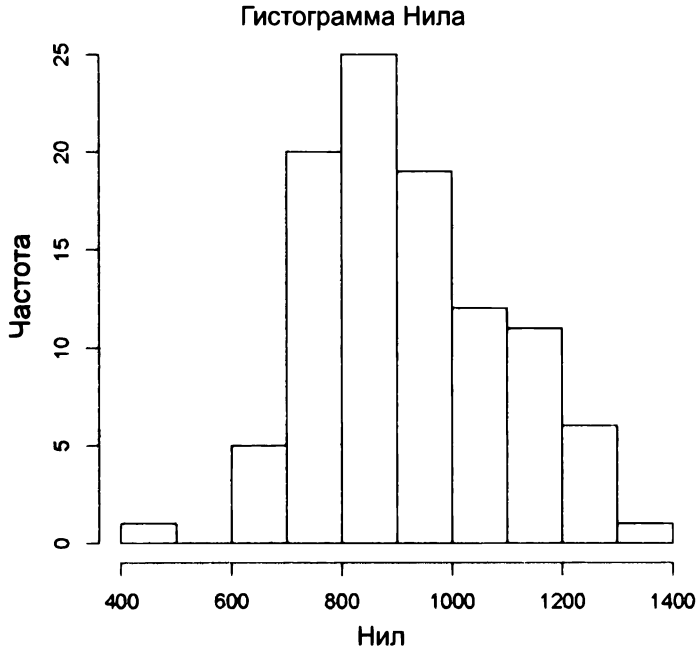


Рис. 1.1. Данные течения Нила (минимальное оформление)

будут автоматически загружены при следующем запуске R. Эта возможность очень важна, особенно при работе с большими или многочисленными наборами данных. Ответ у также сохраняет историю команд сеанса. Сохранение рабочего пространства и истории команд более подробно рассматривается в разделе 1.6.

1.3. Знакомство с функциями

Как и в большинстве языков программирования, суть программирования на R сводится к написанию функций. Функция представляет собой набор команд, который получает входные данные, использует их для вычисления других значений и возвращает результат.

В качестве простого примера определим функцию с именем `oddcount()`, предназначенную для подсчета нечетных чисел в целочисленном векторе. Обычно разработчик пишет код функции в текстовом редакторе и сохраняет его в файле, но в примере на скорую руку мы введем его строку за строкой в интерактивном режиме R. Затем функция будет вызвана в паре тестовых примеров.

```
# Подсчитывает количество нечетных целых чисел в x
> oddcount <- function(x) {
+    k <- 0 # Присвоить k значение 0
+    for (n in x) {
+       if (n %% 2 == 1) k <- k+1 # %% — оператор вычисления остатка
+    }
+    return(k)
+ }
> oddcount(c(1,3,5))
[1] 3
> oddcount(c(1,2,3,7,9))
[1] 4
```

Сначала мы сообщаем R, что хотим определить функцию с именем `oddcount` с одним аргументом `x`. Левая фигурная скобка обозначает начало тела функции. Мы записываем по одной команде R на строку.

Пока тело функции не будет завершено, R напоминает вам о том, что функция еще определяется; для этого обычное приглашение `>` заменяется приглашением `+`. (Хотя на самом деле `+` — признак продолжения строки, а не приглашение для нового ввода.) После того как будет введена правая фигурная скобка, завершающая тело функции, R возвращается к стандартному приглашению `>`.

После определения функции мы получаем результаты двух вызовов `oddcount()`. Так как вектор `(1,3,5)` содержит три нечетных числа, вызов `oddcount(c(1,3,5))` возвращает значение 3. В векторе `(1,2,3,7,9)` четыре нечетных числа, поэтому второй вызов возвращает 4.

Обратите внимание: оператор вычисления остатка в R имеет вид `%%`, как указано в комментарии. Например, при целочисленном делении 38 на 7 остаток равен 3:

```
>38 %% 7
[1] 3
```

А теперь посмотрим, что произойдет при выполнении следующего кода:

```
for (n in x) {
   if(n %% 2 == 1) k <- k+1
}
```

Сначала `n` присваивается значение `x[1]`, после чего переменная проверяется на четность. Если значение нечетно (как в данном случае), то переменная-счетчик `k` увеличивается на 1. Затем `n` присваивается `x[2]`, значение проверяется на четность и т. д.

Кстати говоря, у программистов C/C++ может появиться желание записать этот цикл в следующем виде:

```
for (i in 1:length(x)) {
   if (x[i] %% 2 == 1) k <- k+1
}
```

Здесь `length(x)` — количество элементов в `x`. Предположим, вектор содержит 25 элементов. Запись `1:length(x)` означает `1:25`, что, в свою очередь, означает 1, 2, 3, ..., 25. Такой код будет работать (если только длина `x` не равна 0), но один из главных принципов программирования на R гласит, что следует обходиться без циклов, если это возможно, а если нет, то циклы должны быть простыми. Взглянем еще раз на исходную формулу:

```
for (n in x) {
   if(n %% 2 == 1) k <- k+1
}
```

Она проще и элегантнее, потому что не требует использования функции `length()` и индексирования массива.

В конце кода функции располагается команда `return()`:

```
return(k)
```

Функция возвращает вычисленное значение `k` тому коду, из которого она была вызвана. Впрочем, следующая простая запись тоже работает:

```
k
```

При отсутствии явного вызова `return()` функции R возвращают последнее вычисленное значение. Тем не менее такой подход следует использовать с осторожностью, как будет показано в разделе 7.4.1.

В терминологии языков программирования `x` называется *формальным аргументом* (или *формальным параметром*) функции `oddcount()`. В первом вызове функции из предыдущего примера `c(1,3,5)` называется *фактическим аргументом*. Эти термины указывают на тот факт, что `x` в определении функции представляет собой условное имя, представляющее значение, а `c(1, 3, 5)` определяет само значение, использованное в вычислениях. Аналогичным образом во втором вызове функции `c(1,2,3,7,9)` является фактическим аргументом.

1.3.1. Область видимости переменной

Переменная, видимая только в теле функции, называется *локальной* по отношению к этой функции. В `oddcount()` `k` и `n` являются локальными переменными. Они пропадают после того, как функция вернет управление:

```
> oddcount(c(1,2,3,7,9))
[1] 4
> n
Error: object 'n' not found
```

Важно понимать, что формальные параметры в функции R являются локальными переменным. Предположим, программа содержит следующий вызов функции:

```
> z <- c(2,6,7)
> oddcount(z)
```

Теперь допустим, что код `oddcount()` изменяет `x`. Переменная `z` при этом *не* изменится. После вызова `oddcount()` `z` будет содержать то же значение, что и прежде. Чтобы вычислить результат вызова функции, R копирует каждый фактический аргумент в соответствующую переменную локального параметра, и изменения в этой переменной не будут видны за пределами функции. Правила области видимости будут более подробно рассмотрены в главе 7.

Переменные, созданные за пределами функций, являются *глобальными*; они также доступны внутри функций. Пример:

```
> f <- function(x) return(x+y)
> y<-3
> f(5)
[1] 8
```

Здесь `y` является глобальной переменной.

Запись нового значения в глобальную переменную осуществляется *оператором суперприсваивания* R `<<-`. Эта тема также обсуждается в главе 7.

1.3.2. Аргументы по умолчанию

В R также часто используются *аргументы по умолчанию*. Возьмем определение функции следующего вида:

```
> g <- function(x,y=2,z=T) { ... }
```

Здесь формальный аргумент `y` будет инициализирован 2, если программист не укажет значение `y` при вызове. Аналогичным образом `z` будет присвоено значение `TRUE`.

А теперь возьмем следующий вызов:

```
> g(12,z=FALSE)
```

Здесь значение 12 — фактический аргумент для x, для y принимается значение по умолчанию, но значение по умолчанию для z переопределяется — переменной присваивается FALSE. Предыдущий пример также показывает, что, как и во многих языках программирования, в R существует логический тип (то есть тип с логическими значениями TRUE и FALSE).

ПРИМЕЧАНИЕ

R позволяет сокращать TRUE и FALSE до T и F. Тем не менее вы можете отказаться от сокращения этих значений во избежание проблем, если в программе используются переменные с именами T или F.

1.4. Важнейшие структуры данных R

В R поддерживаются разнообразные структуры данных. Здесь мы кратко опишем несколько наиболее часто используемых структур, чтобы дать вам представление о возможностях R перед тем, как углубляться в подробности. Вы хотя бы увидите несколько содержательных примеров даже при том, что с полным описанием придется подождать.

1.4.1. Векторы

Векторный тип в действительности занимает важнейшее место в R. Трудно представить себе код R (и даже интерактивный сеанс), в котором бы не использовались векторы.

Все элементы вектора должны иметь один тип данных (или *режим* — mode). Например, вектор может содержать три символьные строки (режим character) или три целых числа, но не одно целое число и две символьные строки.

Векторы более подробно рассматриваются в главе 2.

1.4.1.1. Скаляры

Скаляры (то есть отдельные числа) не существуют в R. Как упоминалось ранее, то, что на первый взгляд кажется отдельным числом, в действительности является вектором из одного элемента.

Пример:

```
> x < -8
> x
[1] 8
```

Напомню: [1] означает, что следующая строка чисел начинается с элемента 1 вектора — в данном случае x[1]. Как видите, R действительно интерпретирует x как вектор, хотя и состоящий всего из одного элемента.

1.4.2. Символьные строки

Символьные строки в действительности являются одноэлементными векторами с элементами символьного (не числового) типа:

```
> x <- c(5,12,13)
> x
[1] 51213
> length(x)
[1] 3
> mode(x)
[1] "numeric"
> y <- "abc"
> y
[1] "abc"
> length(y)
[1] 1
> mode(y)
[1] "character"
> z <- c("abc","29 88")
> length(z)
[1] 2
> mode(z)
[1] "character"
```

В первом примере создается вектор x с числовыми элементами (режим numeric). Затем создаются два вектора с символьными элементами (режим character): вектор y содержит одну строку, а z — две строки.

R содержит различные функции для работы со строками. Многие из них предназначены для объединения или разбора строк, как две следующие функции:

```
> u <- paste("abc","de","f") # Конкатенация строк
> u
[1] "abc de f"
> v <- strsplit(u," ") # Разбиение строки по пробелам
> v
[[1]]
[1] "abc" "de" "f"
```

Строки более подробно описаны в главе 11.

1.4.3. Матрицы

Концепция матрицы в R не отличается от традиционного математического понятия: этим термином обозначается прямоугольный массив чисел. Технически матрица представляет собой вектор, но с двумя дополнительными атрибутами: количеством строк и количеством столбцов. Пример работы с матрицей:

```
> m <- rbind(c(1,4),c(2,2))
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    2
> m %*% c(1,1)
     [,1]
[1,]    5
[2,]    4
```

Сначала для построения матрицы из двух векторов, представляющих строки, используется функция `rbind()` (от Row Bind), а затем результат сохраняется в `m`. (Соответствующая функция `cbind()` объединяет несколько столбцов в матрицу.) Затем в командной строке вводится только имя переменной; как вы уже знаете, при этом выводится текущее значение переменной. По нему можно проверить, что была построена именно та матрица, которая вам нужна. Наконец, мы вычисляем матричную производительность вектора `(1,1)` и `m`. Оператор умножения матриц, знакомый вам из курса линейной алгебры, в R имеет вид `%*%`.

Матрицы индексируются по двум индексам; это делается почти так же, как в C/C++, хотя индексы начинаются с 1, а не с 0.

```
> m[1,2]
[1] 4
> m[2,2]
[1] 2
```

В R предусмотрена исключительно полезная возможность извлечения подматриц (по аналогии с извлечением подвекторов из векторов). Пример:

```
> m[1,]  # строка 1
[1] 1 4
> m[,2]  # столбец 2
[1] 4 2
```

О матрицах более подробно поговорим в главе 3.

1.4.4. Списки

По аналогии с векторами R списки R являются контейнерами для значений, однако они могут содержать элементы с разными типами данных. (Программисты C/C++ заметят аналогию со структурами языка C.) Для обращения к элементам списков используются составные имена, которые в R записываются со знаком `$`. Простой пример:

```
> x <- list(u=2, v="abc")
> x
$u
[1] 2

$v
[1] "abc"

> x$u
[1] 2
```

Выражение `x$u` обозначает компонент `u` списка `x`. Последний содержит еще один компонент с именем `v`.

Одно из стандартных применений списков — объединение нескольких значений в один пакет, который возвращается функцией. Это особенно удобно для статических функций, которые могут возвращать сложные результаты. Возьмем базовую функцию построения гистограмм R `hist()`, упомянутую в разделе 1.2. Функция вызывалась для встроенного в R набора данных течения Нила:

```
> hist(Nile)
```

Вызов строил диаграмму, но `hist()` также возвращает значение, которое можно сохранить:

```
> hn <- hist(Nile)
```

Что хранится в `hn`? Давайте посмотрим:

```
> print(hn)
$breaks
 [1] 400 500 600 700 800 900 1000 1100 1200 1300 1400

$counts
 [1]105202519121161

$intensities
 [1] 9.999998e-05 0.000000e+00 5.000000e-04 2.000000e-03 2.500000e-03
```

```
 [6] 1.900000e-03 1.200000e-03 1.100000e-03 6.000000e-04 1.000000e-04

$density
 [1] 9.999998e-05 0.000000e+00 5.000000e-04 2.000000e-03 2.500000e-03
 [6] 1.900000e-03 1.200000e-03 1.100000e-03 6.000000e-04 1.000000e-04

$mids
 [1] 450 550 650 750 850 950 1050 1150 1250 1350

$xname
[1] "Nile"

$equidist
[1] TRUE

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

Не старайтесь понять сразу всё. Сейчас важно то, что, кроме вывода диаграммы, `hist()` возвращает список с несколькими компонентами. В данном случае компоненты описывают характеристики гистограммы. Например, компонент `breaks` сообщает, где начинаются и заканчиваются группы гистограммы, а компонент `counts` содержит количество наблюдений в каждой группе.

Создатели R решили упаковать всю информацию, возвращаемую функцией `hist()`, в список R. Вы можете читать эту информацию и работать с ней при помощи других команд R со знаком `$`.

Напомню, что для вывода содержимого списка `hn` достаточно ввести его имя:

```
>hn
```

Впрочем, есть и более компактный способ вывода списков с функцией `str()`:

```
> str(hn)
List of 7
$ breaks      : num [1:11] 400 500 600 700 800 900 1000 1100 1200 1300 ...
$ counts      : int [1:10]105202519121161
$ intensities : num [1:10] 0.0001 0 0.0005 0.002 0.0025 ...
$ density     : num [1:10] 0.0001 0 0.0005 0.002 0.0025 ...
$ mids        : num [1:10] 450 550 650 750 850 950 1050 1150 1250 1350
$ xname       : chr "Nile"
$ equidist    : logi TRUE
- attr(*, "class")= chr "histogram"
```

Имя `str` является сокращением от *structure*. Эта функция выводит внутреннюю структуру любого объекта R, не только списков.

1.4.5. Кадры данных

Типичный набор данных содержит данные разных типов. Например, в наборе данных работников могут присутствовать символьные строки (имена работников и т. д.) и числовые значения (заработная плата и т. д.). Таким образом, хотя набор данных 50 работников с 4 переменными на работника напоминает матрицу 50 × 4, он не считается матрицей в R, потому что в нем содержатся смешанные типы.

Вместо матрицы мы будем использовать *кадр данных* (data frame). Кадр данных в R представляет собой список, в котором каждый компонент является вектором, соответствующим столбцу нашей «матрицы» данных. Вместо этого кадры данных создаются следующим образом:

```
> d <- data.frame(list(kids=c("Jack","Jill"),ages=c(12,10)))
>d
  kids ages
1 Jack 12
2 Jill 10
> d$ages
[1] 12 10
```

Впрочем, чаще кадры данных создаются в результате чтения из файла или базы данных.

Кадры данных подробно рассматриваются в главе 5.

1.4.6. Классы

R — объектно-ориентированный язык. *Объекты* являются экземплярами *классов*. Классы несколько более абстрактны, чем типы данных, с которыми вы имели дело до сих пор. Здесь мы кратко рассмотрим концепцию на примере классов S3 языка R (имя происходит из старого языка S версии 3, которым руководствовались создатели R). Значительная часть синтаксиса R основана на этих классах, которые чрезвычайно просты. Их экземпляры представляют собой обычные списки R, но с дополнительным *атрибутом* — именем класса.

Например, ранее мы упоминали о том, что (неграфический) вывод функции `hist()` представляет собой список с различными компонентами — `break`, `count` и т. д. Также присутствует атрибут, задающий имя класса списка, а именно `histogram`.

```
> print(hn)
$breaks
 [1] 400 500 600 700 800 900 1000 1100 1200 1300 1400
```

```
$counts
 [1]105202519121161
...
...
attr(,"class")
[1] "histogram"
```

Возникает резонный вопрос: если объекты классов S3 представляют собой обычные списки, то зачем они нужны? Дело в том, что классы используются *обобщенными функциями*. Обобщенная (generic) функция представляет семейство функций, которые решают похожие задачи, но каждая функция предназначена для конкретного класса.

На практике часто используется обобщенная функция `summary()`. Если пользователь R хочет использовать статистическую функцию (например, `hist()`), но не уверен в том, как обрабатывать ее вывод (который может быть довольно объемистым), он может просто вызвать `summary()` для вывода — не обычного списка, а экземпляра класса S3.

Функция `summary()` в действительности является целым семейством функций для вывода сводной информации, каждая из которых работает с объектами конкретного класса. Когда вы вызываете `summary()` для некоторого вывода, R ищет версию, соответствующую имеющемуся классу, и использует ее для вывода более удобного представления списка. Таким образом, вызов `summary()` для выходных данных `hist()` выдает сводку, относящуюся к указанной функции, а при вызове `summary()` для выходных данных функции регрессии `lm()` будет выдана сводка для этой функции.

Функция `plot()` — другая обобщенная функция. Ее тоже можно использовать практически с любым объектом R. Язык R найдет подходящую функцию построения графика в зависимости от класса объекта.

Классы используются для упорядочения объектов. В сочетании с обобщенными функциями они позволяют писать гибкий код для разнообразных, но логически связанных задач. Классы более подробно рассматриваются в главе 9.

1.5. Расширенный пример: регрессионный анализ экзаменационных оценок

А теперь разберем короткий пример статистического регрессионного анализа. В этом примере полноценного программирования не так уж много, но он показывает, как используются некоторые из представленных типов данных, включая

объекты S3 языка R. Кроме того, он будет служить основой для нескольких примеров в последующих главах.

У меня есть файл ExamsQuiz.txt с экзаменационными оценками класса, в котором я преподавал. Несколько начальных строк этого файла:

```
2    3.3   4
3.3  2     3.7
4    4.3   4
2.3  0     3.3
...
```

Каждая строка содержит данные одного студента: оценка промежуточного экзамена в середине семестра, оценка итогового экзамена и средняя оценка контрольной работы. Интересно посмотреть, насколько хорошо оценка промежуточного экзамена и оценка контрольной работы прогнозируют оценку студента на итоговом экзамене.

Для начала файл данных нужно загрузить:

```
> examsquiz <- read.table("ExamsQuiz.txt",header=FALSE)
```

Наш файл не содержит строки заголовка с именами переменнных каждой записи, поэтому при вызове функции передается аргумент `header=FALSE`. В действительности аргумент `header` уже содержит `FALSE` (значение по умолчанию — вы можете убедиться в этом в описании `read.table()` из электронной справки R), поэтому указывать этот параметр необязательно. Тем не менее с явно заданным аргументом вызов получается более четким.

Наши данные теперь содержатся в `examsquiz` — объекте R класса `data.frame`.

```
> class(examsquiz)
[1] "data.frame"
```

Просто чтобы убедиться в том, что файл был прочитан правильно, выведем несколько начальных строк:

```
> head(examsquiz)
   V1  V2  V3
1 2.0 3.3 4.0
2 3.3 2.0 3.7
3 4.0 4.3 4.0
4 2.3 0.0 3.3
5 2.3 1.0 3.3
6 3.3 3.7 4.0
```

При отсутствии заголовка данных R присваивает столбцам имена `V1`, `V2` и `V3`. Слева выводятся номера строк. Конечно, было бы лучше, чтобы в данных при-

сутствовал заголовок с осмысленными именами вида `Exam1`. В последующих примерах будут использоваться имена.

Попробуем спрогнозировать оценку экзамена 2 (второй столбец `examsquiz`) по данным экзамена 1 (первый столбец):

```
lma <- lm(examsquiz[,2] ~ examsquiz[,1])
```

Вызов функции `lm()` (от *linear model*, то есть «линейная модель») приказывает R выполнить подгонку прогностического уравнения:

```
прогноз экзамен 2 = β₀ + β₁ экзамен 1
```

Здесь β_0 и β_1 — константы, оцениваемые по данным. Другими словами, мы подгоняем прямую линию к имеющимся парам (экзамен 1, экзамен 2) в наших данных. Для подгонки используется классический метод наименьших квадратов. (Не беспокойтесь, если этот термин вам пока неизвестен.)

Оценки экзамена 1, хранящиеся в первом столбце кадра данных, совместно обозначаются `examsquiz[,1]`. Отсутствие первого индекса (номера строки) означает, что мы обращаемся ко всему столбцу кадра. Для оценок экзамена 2 используются аналогичные обозначения. Таким образом, вызов `lm()` прогнозирует второй столбец `examsquiz` по первому столбцу.

Также можно было бы использовать запись

```
lma <- lm(examsquiz$V2 ~ examsquiz$V1)
```

поскольку кадр данных — всего лишь список, элементы которого являются векторами. Здесь столбцами являются компоненты `V1`, `V2` и `V3` списка.

Результаты, возвращенные `lm()`, теперь находятся в объекте, который хранится в переменной `lma`. Объект является экземпляром класса `lm`. Список его компонентов можно получить вызовом `attributes()`:

```
> attributes(lma)
$names
 [1] "coefficients"  "residuals" "effects" "rank"
 [5] "fitted.values" "assign"    "qr"      "df.residual"
 [9] "xlevels"       "call"      "terms"   "model"

$class
[1] "lm"
```

Как обычно, для получения более подробной информации можно воспользоваться вызовом `str(lma)`. Полученные оценки β_i хранятся в `lma$coefficients`. Чтобы вывести их, введите имя в приглашении командной строки.

Чтобы уменьшить количество вводимых символов, можно сокращать имена компонентов. Главное — не сократить имя компонента до такой степени, чтобы оно стало неоднозначным. Например, если список содержит компоненты `xyz`, `xywa` и `xbcde`, то второй и третий компоненты можно сократить до `xyw` и `xb` соответственно. Для проверки можно ввести следующую команду:

```
> lma$coef
   (Intercept) examsquiz[, 1]
     1.1205209      0.5899803
```

Так как `lma$coefficients` является вектором, вывести его будет несложно. Но что произойдет при выводе самого объекта `lma`?

```
> lma

Call:
lm(formula = examsquiz[, 2] ~ examsquiz[, 1])

Coefficients:
   (Intercept) examsquiz[, 1]
         1.121          0.590
```

Почему R выводит эти данные, но не другие компоненты `lma`? Дело в том, что R использует `print()` — еще одну обобщенную функцию. Как и другие обобщенные функции, `print()` поручает работу другой функции, специализирующейся на выводе объектов класса `lm` (функции `print.lm()`), а эта функция выводит именно эти данные.

Мы можем получить более подробную распечатку содержимого `lma`, вызвав обобщенную функцию, о которой говорилось ранее. Происходит вызов `summary.lm()` за кулисами, и мы получаем регрессию, специфичную для конкретной ситуации:

```
> summary(lma)

Call:
lm(formula = examsquiz[, 2] ~ examsquiz[, 1])

Residuals:
    Min      1Q  Median      3Q     Max
-3.4804 -0.1239  0.3426 0.7261 1.2225

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)      1.1205     0.6375   1.758  0.08709 .
examsquiz[, 1]   0.5900     0.2030   2.907  0.00614 **
...
```

Для этого класса также определены другие обобщенные функции. За подробностями обращайтесь к электронной справке `lm()`. (О работе с электронной документацией R рассказано в разделе 1.7.)

Для вычисления прогностического уравнения экзамена 2 по данным экзамена 1 и контрольной работы из столбца 3 используется запись со знаком +:

```
> lmb <- lm(examsquiz[,2] ~ examsquiz[,1] + examsquiz[,3])
```

Обратите внимание: знак + не означает, что мы вычисляем сумму двух величин. Это всего лишь символ-ограничитель в списке прогностических переменных.

1.6. Запуск и завершение

Как и у многих сложных программных продуктов, поведение R можно настроить при помощи файлов запуска. Кроме того, R может полностью или частично сохранить сеанс (например, протокол выполненных операций) в выходном файле. Если есть команды R, которые вам хотелось бы выполнять в начале каждого сеанса R, поместите их в файл с именем .Rprofile в домашнем каталоге или в том каталоге, из которого запускается R. Поиск файла начинается со второго каталога, что позволяет создавать специализированные профили для конкретных проектов.

Например, чтобы назначить текстовый редактор, который R будет вызывать при выполнении `edit()`, в файл .Rprofile включается следующая строка (если вы работаете в системе Linux):

```
options(editor="/usr/bin/vim")
```

Функция R `options()` используется для настройки различных параметров конфигурации. Также можно задать полный путь к вашему редактору с символом-разделителем для вашей операционной системы (/ или \).

Другой пример: в файле .Rprofile на моем домашнем компьютере с Linux присутствует следующая строка:

```
.libPaths("/home/nm/R")
```

Она автоматически добавляет в путь поиска R каталог со всеми дополнительными пакетами.

Как и многие программы, R поддерживает концепцию текущего рабочего каталога. В системе Linux или Mac при запуске это будет тот каталог, из которого R был запущен. В Windows текущим каталогом, скорее всего, будет папка

Documents. Если в ходе сеанса R вы будете обращаться к файлам, R будет предполагать, что файлы находятся в этом каталоге. Вы всегда можете проверить текущий каталог командой:

```
> getwd()
```

Чтобы сменить текущий каталог, вызовите функцию `setwd()` с нужным каталогом, заключенным в кавычки. Например, вызов:

```
> setwd("q")
```

выберет в качестве рабочего каталог `q`.

Далее в ходе интерактивного сеанса R записывает введенные команды. Если вы положительно ответите на вопрос Save workspace image? при завершении сеанса, R сохранит все объекты, созданные в ходе сеанса, и восстановит их в следующем сеансе. А это означает, что вам не придется повторять всю работу с нуля и вы сможете продолжить с того места, на котором остановились ранее.

Сохраненное рабочее пространство хранится в файле .Rdata, находящемся в каталоге, из которого был запущен сеанс R (Linux), или в каталоге установки R (Windows). Вы можете просмотреть файл .Rhistory с сохраненными командами, чтобы вспомнить, как создавалось рабочее пространство.

Если вы предпочитаете ускорить запуск/завершение, можно отказаться от загрузки всех этих файлов и сохранения сеанса. Для этого R следует запустить с ключом `vanilla`:

```
R --vanilla
```

Существуют и другие варианты в диапазоне от «не загружать ничего» и «загрузить всё». Дополнительная информация о файлах запуска содержится в электронной справке R; чтобы просмотреть ее, введите следующую команду:

```
> ?Startup
```

1.7. Получение справочной информации

Существует множество разнообразных источников информации, которые помогают в изучении R, в том числе некоторые средства R и, конечно, многие сайты в интернете.

Создатели постарались сделать R самодокументируемым. Сейчас мы рассмотрим некоторые встроенные средства R, а также ресурсы, доступные в интернете.

1.7.1. Функция help()

Чтобы получить электронную справку, выполните команду `help()`. Например, для получения информации о функции `seq()` введите команду:

```
> help(seq)
```

Вместо полного имени `help()` также можно использовать сокращенную запись — вопросительный знак (`?`):

```
> ?seq
```

Специальные символы и некоторые зарезервированные слова должны заключаться в кавычки при использовании в функции `help()`. Например, для получения справки об операторе < вводится следующая команда:

```
> ?"<"
```

А чтобы получить электронную справку по циклам `for`, введите команду:

```
> ?"for"
```

1.7.2. Функция example()

Каждый раздел справки содержит примеры. В R предусмотрена одна полезная возможность: функция `example()` может выполнять эти примеры за вас. Пример:

```
> example(seq)

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

seq> seq(stats::rnorm(20))
 [1]  1  2  3 4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20

seq> seq(1, 9, by = 2) # match
 [1] 1 3 5 7 9

seq> seq(1, 9, by = pi)# stay below
 [1] 1.000000 4.141593 7.283185

seq> seq(1, 6, by = 3)
 [1]14

seq> seq(1.575, 5.125, by=0.05)
 [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.925 1.975 2.025 2.075 2.125
[13] 2.175 2.225 2.275 2.325 2.375 2.425 2.475 2.525 2.575 2.625 2.675 2.725
[25] 2.775 2.825 2.875 2.925 2.975 3.025 3.075 3.125 3.175 3.225 3.275 3.325
```

```
[37] 3.375 3.425 3.475 3.525 3.575 3.625 3.675 3.725 3.775 3.825 3.875 3.925
[49] 3.975 4.025 4.075 4.125 4.175 4.225 4.275 4.325 4.375 4.425 4.475 4.525
[61] 4.575 4.625 4.675 4.725 4.775 4.825 4.875 4.925 4.975 5.025 5.075 5.125

seq> seq(17) # То же, что 1:17
 [1]  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
```

Функция `seq()` генерирует разные числовые последовательности в арифметической прогрессии. В результате исполнения команды `example(seq)` R выполняет некоторые примеры `seq()` прямо при вас.

А теперь представьте, насколько полезно это может быть для графики! Если вам захочется увидеть, что делает одна из превосходных графических функций R, функция `example()` выдаст графическую иллюстрацию.

Чтобы увидеть простой и очень красивый пример, попробуйте выполнить следующую команду:

```
> example(persp)
```

Команда выводит серию примеров графиков для функции `persp()`. Один из них показан на рис. 1.2. Когда вы будете готовы перейти к следующему графику,

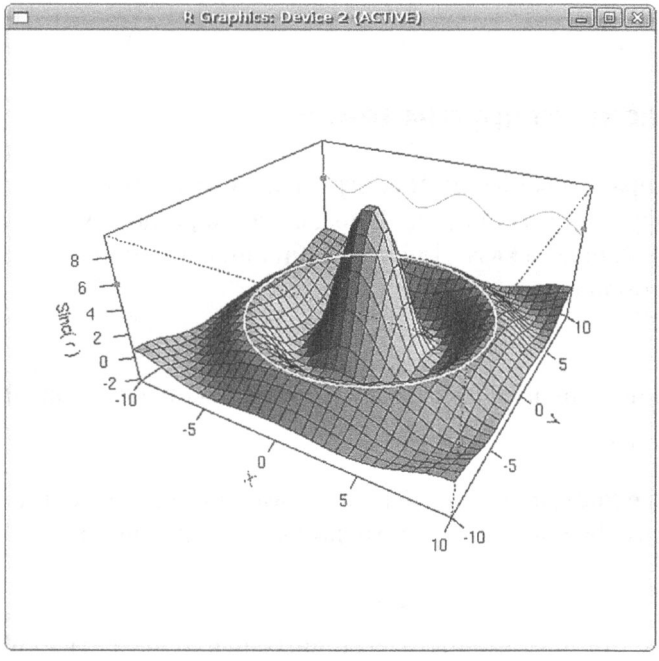


Рис. 1.2. Один из примеров persp()

нажмите Enter в консоли R. Обратите внимание: код каждого примера выводится на консоль, поэтому вы можете поэкспериментировать с настройкой аргументов.

1.7.3. Если вы в общих чертах представляете, что ищете

Функция `help.search()` используется для проведения поиска в стиле Google по документации R. Представьте, что вам нужна функция для генерирования случайных переменных с многомерными нормальными распределениями. Чтобы определить, какая функция решает эту задачу (и есть ли такая функция), можно попробовать следующую команду:

```
> help.search("multivariate normal")
```

Полученный ответ содержит следующий фрагмент:

```
mvrnorm(MASS) Simulate from a Multivariate Normal
              Distribution
```

Как видите, эту задачу решает функция `mvrnorm()` из пакета `MASS`.

Также вызов `help.search()` можно заменить сокращенной записью:

```
> ??"multivariate normal"
```

1.7.4. Справка по другим темам

Внутренние справочные файлы R не ограничиваются страницами с описаниями конкретных функций. Например, в предыдущем разделе упоминалась функция `mvrnorm()` из пакета `MASS`. Для получения информации о функции введите следующую команду:

```
> ?mvrnorm
```

Также можно получить информацию обо всем пакете следующей командой:

```
> help(package=MASS)
```

Справка также доступна и по общим темам. Например, если вам захочется больше узнать о файлах, используйте команду следующего вида:

```
> ?files
```

Функция выводит информацию о различных функциях для работы с файлами, таких как `file.create()`.

Примеры других тем:

```
Arithmetic
Comparison
Control
Dates
Extract
Math
Memory
NA
NULL
NumericaConstants
Paren
Quotes
Startup
Syntax
```

Просматривать справку по этим темам полезно даже в том случае, если вы не ищете информацию по какому-то конкретному вопросу.

1.7.5. Справка по пакетному режиму

Вспомните, что в R существуют пакетные команды, которые позволяют выполнять команды прямо из оболочки операционной системы. Чтобы получить справку по конкретной пакетной команде, введите следующую команду:

```
R CMD command --help
```

Например, для вывода описаний всех параметров команды `INSTALL` (см. приложение Б) используется следующая команда:

```
R CMD INSTALL --help
```

1.7.6. Справочная информация в интернете

В интернете можно найти много замечательных ресурсов, посвященных R. Несколько примеров:

- Руководства проекта R доступны на домашней странице R по адресу http://www.r-project.org/. Щелкните на ссылке Manuals.
- Разные системы поиска информации по R доступны на домашней странице R. Щелкните на ссылке Search.
- Пакет `sos` предоставляет мощные средства поиска материалов о R. Инструкции по установке пакетов R приведены в приложении Б.

- Я часто пользуюсь поисковой системой RSeek: http://www.rseek.org/.
- Вопросы по R можно отправить на сервер рассылки R `r-help`. Информация об этом и других серверах рассылки R находится по адресу http://www.r-project.org/mail.html. Можно использовать разные интерфейсы; мне нравится Gmane (http://www.gmane.org/).

Однобуквенное имя затрудняет поиск информации о R в поисковых системах общего назначения (таких, как Google). Впрочем, вы можете воспользоваться некоторыми полезными приемами, например критериями `filetype` Google. Чтобы провести поиск сценариев R (файлов с суффиксом .R), связанных с перестановками, введите следующую команду:

```
filetype:R permutations -rebol
```

Ключ `-rebol` приказывает Google исключить страницы со словом «rebol», поскольку язык программирования REBOL использует тот же суффикс.

CRAN (Comprehensive R Archive Network) (http://cran.r-project.org/) — репозиторий кода R, находящегося под контролем пользователей; это имя станет хорошим критерием поиска Google. Например, поиск по строке «lm CRAN» поможет найти материалы по функции R `lm()`.

2 Векторы

Основополагающим типом данных в R является *вектор*. В главе 1 были приведены вводные примеры, а теперь вы узнаете подробности. Сначала мы посмотрим, как векторы связаны с другими типами данных в R. Вы увидите, что, в отличие от языков семейства C, отдельные числа (скаляры) не обладают отдельными типами данных, а являются особой разновидностью векторов. С другой стороны, как и в языках семейства C, матрицы являются особой разновидностью векторов.

В этой главе довольно подробно рассматриваются следующие темы:

- *Переработка* — автоматическое изменение длины векторов в некоторых ситуациях.
- *Фильтрация* — извлечение подмножеств элементов из векторов.
- *Векторизация* — применение функций к каждому элементу вектора.

Все эти операции занимают центральное место в программировании на R. Они будут часто упоминаться в оставшейся части книги.

2.1. Скаляры, векторы, массивы и матрицы

Во многих языках программирования векторные переменные считаются чем-то отличающимся от скалярных (отдельных чисел). Возьмем следующий код C:

```
int x;
int y[3];
```

Компилятор должен выделить память для одной переменной с именем `x` и целочисленного массива из трех элементов (этот термин C аналогичен векторному типу R) с именем `y`. Однако в R числа считаются векторами из одного элемента, а такого понятия, как скаляр, вообще не существует.

Типы переменных R называются *режимами* (modes). Вспомните, о чем говорилось в главе 1: все элементы вектора должны иметь одинаковый режим — `integer`, `numeric` (число с плавающей точкой), `character` (строка), `logical`, `complex` и т. д. Если вам понадобится проверить режим переменной `x` в коде программы, запросите его вызовом `typeof(x)`.

В отличие от индексов векторов в языках семейства ALGOL (таких, как C и Python), индексы векторов в R начинаются с 1.

2.1.1. Добавление и удаление элементов векторов

Векторы, как и массивы в C, занимают непрерывный блок памяти, поэтому вставлять или удалять элементы в них невозможно (хотя программисты Python могли привыкнуть к этой возможности). Размер вектора определяется в момент создания, и если вы захотите добавлять или удалять элементы, вектор придется заново разместить в другой части памяти.

Например, добавим элемент в середину вектора из четырех элементов:

```
> x <- c(88,5,12,13)
> x <- c(x[1:3],168,x[4]) # Вставить 168 перед 13
>x
[1]  88   5  12 168  13
```

Здесь мы создаем вектор из четырех элементов и присваиваем его `x`. Чтобы вставить новое число 168 между третьим и четвертым элементами, мы извлекаем первые три элемента `x`, добавляем 168, а потом четвертый элемент `x`. В результате создается новый вектор из пяти элементов, а `x` на некоторое время остается неизменным. После этого новый вектор присваивается `x`.

В результате все выглядит так, словно команда действительно изменила вектор, хранящийся в `x`, но в действительности был создан новый вектор, который был сохранен в `x`. Различие на первый взгляд кажется второстепенным, но у него есть свои последствия. Например, в некоторых случаях он может ограничивать возможности ускорения вычислений в R (см. главу 14).

ПРИМЕЧАНИЕ

Для читателей с опытом работы на C: во внутренней реализации `x` является указателем, а повторное присваивание реализуется сохранением в `x` указателя на новый созданный объект.

2.1.2. Получение длины вектора

Для получения длины вектора используется функция `length()`:

```
> x <- c(1,2,4)
> length(x)
[1] 3
```

В этом примере длина `x` уже известна, поэтому запрашивать ее не нужно. Однако при написании обобщенного кода функций часто возникает необходимость в определении длин векторов-аргументов.

Предположим, вы хотите написать функцию, которая определяет индекс первого значения 1 в векторном аргументе функции (будем считать, что такое значение присутствует). Возможный (но не обязательно эффективный!) способ написания этого кода выглядит так:

```
first1 <- function(x) {
   for (i in 1:length(x)) {
      if (x[i] == 1) break # Выйти из цикла
   }
   return(i)
}
```

Без функции `length()` в `first1()` пришлось бы добавить второй аргумент — допустим, `n` — для определения длины `x`.

Обратите внимание: в этом случае цикл следующего вида *не подойдет*:

```
for (n in x)
```

Дело в том, что этот подход не позволяет получить индекс нужного элемента. Следовательно, нужен внешний цикл, для которого, в свою очередь, необходимо вычисление длины `x`.

И последнее замечание по поводу цикла: чтобы код был качественным, следует убедиться в том, что значение `length(x)` отлично от 0. Посмотрим, что произойдет с выражением `1:length(x)` в цикле `for`:

```
> x < -c()
> x
NULL
> length(x)
[1] 0
> 1:length(x)
[1] 1 0
```

Переменная `i` в цикле принимает значение 1, а потом значение 0 — конечно, это не то, что должно происходить при пустом векторе `x`.

Другое, более безопасное решение использует более сложную функцию R `seq()`, которая будет рассматриваться в разделе 2.4.4.

2.1.3. Матрицы и массивы как векторы

Как вы вскоре увидите, массивы и матрицы (и даже списки в определенном смысле) на самом деле тоже являются векторами. Они всего лишь содержат дополнительные атрибуты класса. Например, матрица содержит некоторое количество строк и столбцов. Они будут подробно рассматриваться в следующей главе, но сейчас стоит отметить, что массивы и матрицы являются векторами, а следовательно, все, что сказано о векторах, относится и к ним.

Рассмотрим пример:

```
> m
  [,1] [,2]
[1,]    1    2
[2,]    3    4
> m + 10:13
  [,1] [,2]
[1,]   11   14
[2,]   14   17
```

Матрица 2 × 2 с именем `m` хранится в форме вектора из четырех элементов по столбцам: `(1,3,2,4)`. Затем к нему прибавляется вектор `(10,11,12,13)`, в результате будет получен вектор `(11,14,14,17)`. Однако R знает, что операция выполняется с матрицей, и вы получите матрицу 2 × 2 — последние строки примера доказывают это.

2.2. Объявления

Обычно компилируемые языки требуют объявления переменных; иначе говоря, программист должен предупредить интерпретатор/компилятор о существовании переменных перед их использованием. В частности, это делалось в предшествующем примере кода C:

```
int x;
int y[3];
```

Как и в большинстве сценарных языков (таких, как Python и Perl), в R объявлять переменные не нужно. Возьмем следующую команду:

```
z <-3
```

Эта команда абсолютно законна (и обычна) даже в том случае, если имя z впервые упоминается в программе.

Но если вы обращаетесь к конкретным элементам вектора, R необходимо предупредить об этом. Допустим, мы хотим, чтобы вектор y состоял из двух компонентов со значениями 5 и 12. Следующая команда не работает:

```
> y[1] <- 5
> y[2] <- 12
```

Вместо этого необходимо сначала создать y — например, так:

```
> y <- vector(length=2)
> y[1] <- 5
> y[2] <- 12
```

Следующая команда тоже сработает:

```
> y <- c(5,12)
```

Такое решение работает, потому что в правой части создается новый вектор, с которым затем связывается переменная y.

Причина, по которой в коде R нельзя неожиданно использовать выражения вида y[2], связана с природой функционального языка R. Операции чтения и записи отдельных элементов векторов на самом деле выполняются функциями. Если R не знает, что y является вектором, у этих функций не будет информации для выполнения операций.

Аналогичным образом необъявленные переменные не ограничиваются в отношении режима (типа данных). Следующая последовательность команд вполне допустима:

```
> x <- c(1,5)
> x
[1] 1 5
> x <- "abc"
```

Сначала x связывается с числовым вектором, а затем со строкой. (Еще раз для программистов C/C++: x — всего лишь указатель, который в разное время может указывать на объекты разных типов.)

2.3. Переработка

При применении операции к двум векторам, которые должны иметь одинаковую длину, R автоматически *перерабатывает* (то есть повторяет) более короткий вектор до тех пор, пока его длина не сравняется с длиной большего вектора. Пример:

```
> c(1,2,4) + c(6,0,9,20,22)
[1] 7 2 13 21 24
Warning message:
longer object length
  is not a multiple of shorter object length in: c(1, 2, 4) + c(6,
  0, 9, 20, 22)
```

Вследствие переработки короткого вектора операция будет выполняться в следующей форме:

```
> c(1,2,4,1,2) + c(6,0,9,20,22)
```

Другой, не столь тривиальный пример:

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

И снова напомню, что матрицы в действительности являются длинными векторами. Здесь `x` — матрица 3 × 2 — тоже является вектором из шести элементов. В R ее элементы хранятся по столбцам. Иначе говоря, в отношении хранения данных матрица `x` не отличается от `c(1,2,3,4,5,6)`. Мы складываем вектор из двух элементов с вектором из шести элементов, так что прибавляемый вектор придется дважды повторить, чтобы получить шесть элементов. Фактически происходило следующее:

```
x + c(1,2,1,2,1,2)
```

Более того, вектор `c(1,2,1,2,1,2)` перед суммированием тоже был преобразован в матрицу с такими же размерами, как у `x`:

```
1 2
2 1
1 2
```

Таким образом, в конечном итоге будут выполнены следующие вычисления:

$$\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ 1 & 2 \end{pmatrix}$$

2.4. Основные операции с векторами

А теперь рассмотрим некоторые распространенные операции, связанные с векторами. Мы рассмотрим арифметические и логические операции, индексирование векторов и некоторые удобные способы создания векторов. Затем будут рассмотрены два расширенных примера работы с этими векторами.

2.4.1. Арифметические и логические операции с векторами

Вспомните, что R является функциональным языком. Каждый оператор, включая + из следующего примера, в действительности является функцией.

```
> 2+3
[1] 5
> "+"(2,3)
[1] 5
```

Вспомните, что скаляры в действительности являются одноэлементными векторами. Векторы можно складывать, а операция + может применяться к парным элементам векторов.

```
> x <- c(1,2,4)
> x + c(5,0,-1)
[1] 6 2 3
```

Если вы знакомы с линейной алгеброй, результат умножения двух векторов может оказаться неожиданным.

```
> x * c(5,0,-1)
[1] 5 0 -4
```

Из-за особенностей функции * умножение выполняется между парами элементов. Первый элемент произведения (5) равен результату умножения первого элемента x (1) на первый элемент c(5,0,1) (5) и т. д.

Остальные числовые операторы работают по тому же принципу. Вот пример:

```
> x <- c(1,2,4)
> x / c(5,4,-1)
[1]  0.2  0.5  -4.0
> x %% c(5,4,-1)
[1] 1 2 0
```

2.4.2. Индексирование векторов

Одна из самых важных и часто используемых операций R — операция *индексирования* векторов. В результате этой операции создается подвектор, полученный выбором элементов заданного вектора с указанными индексами. Запись вида вектор1[вектор2] означает, что выбираются элементы первого вектора, индексы которых содержатся в векторе 2.

```
> y <- c(1.2,3.9,0.4,0.12)
> y[c(1,3)] # Извлечь элементы 1 и 3 вектора y
[1] 1.2 0.4
> y[2:3]
[1] 3.9 0.4
> v <- 3:4
> y[v]
[1] 0.40 0.12
```

Дубликаты в списке индексов разрешены:

```
> x <- c(4,2,17,5)
> y <- x[c(1,1,3)]
> y
[1] 4 4 17
```

Отрицательные индексы означают, что указанные элементы должны быть исключены из вывода.

```
> z <- c(5,12,13)
> z[-1] # Исключить элемент 1
[1] 12 13
> z[-1:-2] # Исключить элементы с 1 по 2
[1] 13
```

В таких контекстах часто бывает удобно использовать функцию `length()`. Предположим, из вектора `z` нужно выбрать все элементы, кроме последнего. Следующий код решает эту задачу:

```
> z <- c(5,12,13)
> z[1:(length(z)-1)]
[1]  5 12
```

Или еще проще:

```
> z[-length(z)]
[1]  5 12
```

Такое решение получается более общим, чем запись `z[1:2]`. Возможно, программе придется работать не только с векторами длины 2, и второе решение обеспечит необходимую универсальность.

2.4.3. Генерирование векторов оператором :

Существует ряд операторов R, особенно полезных для создания векторов. Начнем с оператора :, представленного в главе 1. Он строит вектор, содержащий числа из заданного интервала:

```
> 5:8
[1] 5 6 7 8
> 5:1
[1] 5 4 3 2 1
```

Вспомните, что этот оператор уже использовался ранее в этой главе в контексте цикла:

```
for (i in 1:length(x)) {
```

Учитывайте приоритет операторов.

```
> i <-2
> 1:i-1 # Это означает (1:i) - 1, а не 1:(i-1)
[1] 0 1
> 1:(i-1)
[1] 1
```

В выражении `1:i-1` оператор : обладает более высоким приоритетом, чем вычитание. Таким образом, выражение `1:i` вычисляется в первую очередь, и мы получаем `1:2`. Затем R вычитает 1 из этого выражения. При этом одноэлементный вектор вычитается из двухэлементного, поэтому применяется переработка.

Одноэлементный вектор (1) расширяется до (1,1), чтобы он имел одинаковую длину с вектором 1:2. Поэлементное вычитание дает вектор (0,1).

С другой стороны, в выражении 1:(i-1) круглые скобки имеют более высокий приоритет, чем двоеточие. Потому 1 вычитается из i, что дает вектор 1:1, как видно из предыдущего примера.

ПРИМЕЧАНИЕ

Полная информация о приоритете операторов R содержится в справочной системе. Введите команду ?Syntax в приглашении командной строки.

2.4.4. Генерирование векторных последовательностей функцией seq()

Обобщенной формой : является функция seq(), генерирующая арифметические прогрессии. Например, если 3:8 дает вектор (3,4,5,6,7,8), в котором элементы генерируются с шагом 1 (4–3=1, 5–4=1 и т. д.), можно сгенерировать последовательность с шагом 3:

```
> seq(from=12,to=30,by=3)
[1] 12 15 18 21 24 27 30
```

Шаг не обязан быть целым — например, можно использовать дробное значение 0,1:

```
> seq(from=1.1,to=2,length=10)
[1] 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Функцию seq() удобно использовать для решения проблемы пустых векторов (см. раздел 2.1.2). Допустим, заголовок цикла выглядит так:

```
for (i in 1:length(x))
```

Если вектор x пуст, ни одной итерации быть не должно, но в действительности будут выполнены две итерации, так как выражение 1:length(x) дает результат (1,0). Проблему можно решить, записав команду в следующем виде:

```
for (i in seq(x))
```

Чтобы понять, почему это решение работает, немного поэкспериментируем с seq():

```
> x <- c(5,12,13)
> x
[1]  5 12 13
> seq(x)
[1] 1 2 3
> x <- NULL
> x
NULL
> seq(x)
integer(0)
```

Вы видите, что для непустых x вызов seq(x) дает такой же результат, как 1:length(x), но при пустом x оно дает правильный результат NULL, что приводит к нулевому количеству итераций в приведенном цикле.

2.4.5. Повторение векторных констант функцией rep()

Функция rep() (от *repeat*, то есть «повторить») позволяет легко заполнить длинные векторы одинаковыми константами. Вызов rep(x,times) создает вектор из times*length(x) элементов, то есть times копий x. Пример:

```
> x <- rep(8,4)
> x
[1] 8 8 8 8
> rep(c(5,12,13),3)
[1] 5 12 13  5 12 13  5 12 13
> rep(1:3,2)
[1] 1 2 3 1 2 3
```

Также имеется именованный аргумент each, который изменяет поведение функции и позволяет дублировать элементы в копиях x.

```
> rep(c(5,12,13),each=2)
[1] 5 5 12 12 13 13
```

2.5. all() и any()

Функции any() и all() представляют собой удобные сокращения для стандартных операций проверки. Они проверяют, равны ли какие-либо или все их аргументы TRUE:

```
> x <- 1:10
> any(x > 8)
[1] TRUE
```

```
> any(x > 88)
[1] FALSE
> all(x > 88)
[1] FALSE
> all(x > 0)
[1] TRUE
```

Предположим, R выполняет следующую команду:

```
> any(x > 8)
```

Сначала выполняется проверка `x > 8`; результатом будет следующий вектор:

```
(FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,TRUE,TRUE)
```

Функция `any()` проверяет, равно ли *хотя бы одно* из этих значений `TRUE`. Функция `all()` работает аналогично и сообщает, равны ли `TRUE` *все* значения.

2.5.1. Расширенный пример: поиск серий последовательных единиц

Предположим, вы хотите найти серии последовательных единиц в векторах, содержащих только значения 0 и 1. Например, в векторе `(1,0,0,1,1,1,0,1,1)` присутствует серия длины 3, начинающаяся с индекса 3, и серии длины 2, начинающиеся с индексов 4, 5 и 8. Таким образом, вызов приведенной ниже функции `findruns(c(1,0,0,1,1,1,0,1,1),2)` вернет `(4,5,8)`. Код функции:

```
1 findruns <- function(x,k) {
2    n <- length(x)
3    runs <- NULL
4    for (i in 1:(n-k+1)) {
5       if (all(x[i:(i+k-1)]==1)) runs <- c(runs,i)
6    }
7    return(runs)
8 }
```

В строке 5 требуется определить, равны ли 1 все `k` значений, начиная с `x[i]`, то есть все значения `x[i]`, `x[i+1]`, ..., `x[i+k-1]`. Выражение `x[i:(i+k-1)]` выделяет этот диапазон в `x`, а последующее применение `all()` сообщает, содержит ли он серию единиц.

Протестируем программу:

```
> y <- c(1,0,0,1,1,1,0,1,1)
> findruns(y,3)
```

```
[1] 4
> findruns(y,2)
[1]458
> findruns(y,6)
NULL
```

Хотя использование `all()` в этом коде уместно, решение с накоплением векторов `runs` оставляет желать лучшего. Выделение памяти под вектор требует времени. Каждое выполнение следующей команды замедляет работу программы, так как команда создает новый вектор вызовом `c(runs,i)`. (Тот факт, что новый вектор присваивается `runs`, роли не играет; память для хранения вектора выделяется в любом случае.)

```
runs <- c(runs,i)
```

Вероятно, в коротком цикле проблем не будет, но если быстродействие приложения для вас критично, есть и лучшие решения.

Один из вариантов — предварительное выделение памяти:

```
 1 findruns1 <- function(x,k) {
 2    n <- length(x)
 3    runs <- vector(length=n)
 4    count <- 0
 5    for (i in 1:(n-k+1)) {
 6       if (all(x[i:(i+k-1)]==1)) {
 7          count <- count + 1
 8          runs[count] <- i
 9       }
10    }
11    if (count > 0) {
12       runs <- runs[1:count]
13    } else runs <- NULL
14    return(runs)
15 }
```

В строке 3 выделяется память для вектора длины `n`. Тем самым предотвращаются новые выделения памяти во время выполнения цикла — мы просто заполняем `runs` в строке 8. Непосредственно перед выходом из функции `runs` переопределяется в строке 12 для удаления неиспользуемой части вектора.

Такое решение лучше исходного, поскольку количество операций выделения памяти сократилось до 2 (тогда как в первой версии кода их могло быть много). Если скорость действительно критична, можно рассмотреть возможность написания кода на C (см. главу 14).

2.5.2. Расширенный пример: прогнозирование временных рядов с дискретными значениями

Допустим, имеются наблюдения данных с возможными значениями 0 и 1, по одному значению за период времени. Для конкретности предположим, что это ежедневные погодные данные: 1 — идет дождь, 0 — дождя нет. Нужно спрогнозировать, будет ли дождь завтра, на основании того, насколько часто он шел в последнее время. А именно для некоторого числа k завтрашняя погода будет прогнозироваться на основании погодных данных за последние k дней. Мы будем использовать правило большинства: если количество 1 за предыдущие k периодов времени не меньше k/2, прогнозируется следующее значение 1; в противном случае прогнозируется 0. Например, если k = 3 и данные трех последних периодов равны 1, 0, 1, то на следующий период прогнозируется 1.

Но как выбрать k? Очевидно, если значение будет слишком малым, выборка для прогнозирования окажется недостаточной. При слишком большом значении прогнозирование будет учитывать данные из далекого прошлого, которые не имеют прогностической ценности.

Стандартное решение этой задачи заключается в том, чтобы взять известные данные — так называемый *обучающий набор* (training set), — а затем проверить, насколько эффективно работает прогнозирование при разных значениях k.

Предположим, в примере с погодой имеются данные за 500 дней, и рассматривается возможность использования k = 3. Чтобы оценить прогностическую способность этого значения k, мы «прогнозируем» каждый день данные на основании трех предшествующих дней, а затем сравниваем прогнозы с известными значениями. После того как это будет сделано для всех данных, будет известна частота ошибок для k = 3. То же самое делается для k = 1, k = 2, k = 4 и так далее вплоть до некоторого максимального значения k, которое считается достаточным. После этого то значение k, которое лучше всего работало для обучающих данных, используется для будущих прогнозов.

Как запрограммировать это решение на R? Наивный подход может выглядеть так:

```
1 preda <- function(x,k) {
2    n <- length(x)
3    k2 <- k/2
4    # Вектор pred будет содержать прогнозируемые значения
5    pred <- vector(length=n-k)
6    for (i in 1:(n-k)) {
7       if (sum(x[i:(i+(k-1))]) >= k2) pred[i] <- 1 else pred[i] <- 0
```

```
 8    }
 9    return(mean(abs(pred-x[(k+1):n])))
10 }
```

Все самое главное происходит в строке 7. Здесь прогноз для дня `i+k` (который будет сохранен в `pred[i]`) определяется на основании `k` предыдущих дней, то есть дней `i,...,i+k-1`. Таким образом, среди этих дней нужно посчитать 1. Так как мы работаем с данными 0 и 1, количество 1 равно сумме `x[j]` за эти дни. Нужное значение можно удобно получить следующим образом:

```
sum(x[i:(i+(k-1))])
```

Использование `sum()` и индексирования вектора позволяет выполнить вычисления компактно, без явного программирования цикла, так что решение получается более простым и быстрым. Это типично для кода R.

То же самое можно сказать и о следующем выражении в строке 9:

```
mean(abs(pred-x[(k+1):n]))
```

Здесь `pred` содержит прогнозы, а `x[(k+1):n]` — фактические значения для соответствующих дней. Вычитание второго из первого дает значения 0, 1 или –1. Здесь 1 или –1 соответствует ошибкам прогнозирования в одном или в другом направлении: прогнозом 0, когда истинное значение было равно 1, или наоборот. Получая абсолютные значения (модули) вызовом `abs()`, мы получаем 0 и 1; второй случай соответствует ошибкам.

Теперь мы знаем, в какие дни были получены ошибки. Остается вычислить пропорцию ошибок. Для этого используется вызов `mean()` и тот математический факт, что среднее значение данных 0 или 1 равно части 1. Этот прием часто встречается в R.

Приведенная выше реализация функции `preda()` прямолинейна, и к ее преимуществам можно отнести простоту и компактность. Тем не менее, вероятно, она будет неэффективной. Можно попытаться ускорить ее за счет векторизации цикла, как объяснялось в разделе 2.6, но это не решит главную проблему — повторяющиеся вычисления в коде. Для последовательных значений `i` в цикле функция `sum()` вызывается для векторов, отличающихся только двумя элементами. Это может серьезно замедлить программу (кроме очень малых `k`).

Перепишем код так, чтобы воспользоваться результатами ранее выполненных вычислений. При каждой итерации цикла мы будем обновлять ранее полученную сумму, вместо того чтобы вычислять новую сумму с нуля.

```
 1 predb <- function(x,k) {
 2    n <- length(x)
 3    k2 <- k/2
 4    pred <- vector(length=n-k)
 5    sm <- sum(x[1:k])
 6    if (sm >= k2) pred[1] <- 1 else pred[1] <- 0
 7    if (n-k >= 2) {
 8       for (i in 2:(n-k)) {
 9          sm <- sm + x[i+k-1] - x[i-1]
10          if (sm >= k2) pred[i] <- 1 else pred[i] <- 0
11       }
12    }
13    return(mean(abs(pred-x[(k+1):n])))
14 }
```

Ключевой является строка 9. Здесь мы обновляем `sm`, вычитая самый старый элемент с получением суммы (`x[i-1]`) и прибавляя новый (`x[i+k-1]`).

Впрочем, есть и другое решение: можно воспользоваться функцией R `cumsum()`, которая вычисляет накапливаемые суммы для элементов вектора. Пример:

```
> y <- c(5,2,-3,8)
> cumsum(y)
[1]  5  7  4 12
```

Здесь накапливаемые суммы для `y` будут равны 5 = 5,5 + 2 = 7,5 + 2+(–3) = 4 и 5 + 2 + (–3) + 8 = 12; эти значения будут возвращены функцией `cumsum()`.

Выражение `sum(x[i:(i+(k-1))` в функции `preda()` из примера наводит на мысль о возможном использовании разностей `cumsum()`:

```
predc <- function(x,k) {
   n <- length(x)
   k2 <- k/2
   # Вектор pred будет содержать прогнозируемые значения
   pred <- vector(length=n-k)
   csx <- c(0,cumsum(x))
   for (i in 1:(n-k)) {
      if (csx[i+k] - csx[i] >= k2) pred[i] <- 1 else pred[i] <- 0
   }
   return(mean(abs(pred-x[(k+1):n])))
}
```

Вместо применения `sum()` к окну из `k` последовательных элементов в `x`:

```
sum(x[i:(i+(k-1))
```

та же сумма будет вычисляться как разность накапливаемых сумм в начале и конце окна:

```
csx[i+k] - csx[i]
```

Обратите внимание на присоединение 0 перед вектором накапливаемых сумм:

```
csx <- c(0,cumsum(x))
```

Это необходимо для правильной обработки случая `i=1`.

Код `predc()` требует всего одной операции вычитания на каждую итерацию цикла (в отличие от двух в `predb()`).

2.6. Векторизованные операции

Предположим, имеется функция `f()`, которая должна быть применена ко всем элементам вектора `x`. Во многих случаях задача решается простым вызовом `f()` для самого вектора `x`. Это может упростить код и, более того, обеспечить кардинальное улучшение быстродействия в 100 и более раз.

Один из самых эффективных способов повышения скорости кода R заключается в использовании *векторизованных* операций — это означает, что функция, применяемая к вектору, в действительности применяется к каждому элементу по отдельности.

2.6.1. Вектор на входе, вектор на выходе

Примеры векторизованных функций уже встречались вам ранее в этой главе (операторы `+` и `*`). Другой пример — оператор `>`:

```
> u <- c(5,2,8)
> v <- c(1,3,9)
> u > v
[1] TRUE FALSE FALSE
```

Функция `>` применяется к `u[1]` и `v[1]` с результатом `TRUE`, затем к `u[2]` и `v[2]` с результатом `FALSE` и т. д.

Здесь важно то, что если функция R использует векторизованные операции, она тоже является векторизованной; тем самым открывается потенциальная возможность для ускорения. Пример:

```
> w <- function(x) return(x+1)
> w(u)
[1] 6 3 9
```

Здесь `w()` использует векторизованную операцию `+`, так что `w()` также является векторизованной. Как легко догадаться, существует неограниченное количество векторизованных функций, так как сложные функции строятся из более простых.

Учтите, что векторизованы даже трансцендентные функции — квадратный корень, логарифм, тригонометрические функции и т. д.

```
> sqrt(1:9)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427
[9] 3.000000
```

Это относится ко многим встроенным функциям R. Например, применим функцию округления до ближайшего целого к целочисленному вектору `y`:

```
> y <- c(1.2,3.9,0.4)
> z <- round(y)
> z
[1]140
```

Функция `round()` применяется по отдельности к каждому элементу вектора `y`. Помните, что скаляры в действительности являются одноэлементными векторами, так что «обычное» использование `round()` для одного числа — всего лишь частный случай.

```
> round(1.2)
[1] 1
```

Здесь используется встроенная функция `round()`, но то же самое можно сделать с функциями, которые вы написали самостоятельно.

Как упоминалось ранее, даже такие операторы, как `+`, в действительности являются функциями. Например, возьмем следующий код:

```
> y <- c(12,5,13)
> y+4
[1] 16  9 17
```

Поэлементное добавление 4 работает, потому что `+` в действительности является функцией! Вот то же самое в явном виде:

```
> '+'(y,4)
[1] 16 9 17
```

Также заметьте, что переработка играет здесь ключевую роль, так как 4 расширяется в вектор `(4,4,4)`.

Так как мы знаем, что в R нет обычных скаляров, рассмотрим векторизованные функции, которые на первый взгляд получают скалярные аргументы.

```
>f
function(x,c) return((x+c)^2)
> f(1:3,0)
[1]149
> f(1:3,1)
[1]  4  9 16
```

В нашем определении `f()` переменная `c` явно задумана скалярной, но, конечно, на самом деле это вектор длины 1. Даже если при вызове `f()` передается одно число, оно будет расширено в вектор для вычисления `x+c` внутри `f()`. Таким образом, в вызове `f(1:3,1)` из нашего примера значение `x+c` преобразуется в следующую форму:

$$\begin{pmatrix}1\\2\\3\end{pmatrix}+\begin{pmatrix}1\\1\\1\end{pmatrix}$$

Но при этом встает вопрос безопасности кода. В `f()` нет ничего, что помешало бы передать «обычный» вектор в `c`, как в следующем примере:

```
> f(1:3,1:3)
[1]  4 16 36
```

Проследите за выполняемыми вычислениями и убедитесь в том, что функция действительно получает вектор `(4,16,36)`.

Если вы действительно хотите ограничить `c` скалярами, вставьте соответствующую проверку — например, такую:

```
> f
function(x,c) {
if (length(c) != 1) stop("vector c not allowed")
   return((x+c)^2)
}
```

2.6.2. Вектор на входе, матрица на выходе

Векторизованные функции, с которыми мы работали до настоящего момента, возвращали скалярные значения. При вызове `sqrt()` для числа будет получено число. Если эта функция будет применена к вектору из восьми элементов, вы получите на выходе восемь чисел, то есть другой вектор из восьми элементов.

Но что, если сама функция возвращает векторное значение, как `z12()` в следующем примере:

```
z12 <- function(z) return(c(z,z^2))
```

При вызове `z12()` для аргумента 5 будет получен вектор из двух элементов (`5,25`). Если же вызвать эту функцию для вектора из восьми элементов, она выдаст 16 чисел:

```
x <- 1:8
> z12(x)
[1]  1  2  3  4  5  6  7  8  1  4  9  16  25  36  49  64
```

Более естественно было бы организовать результат в виде матрицы 8 × 2. Это можно сделать при помощи функции `matrix()`:

```
> matrix(z12(x),ncol=2)
     [,1] [,2]
[1,]    1    1
[2,]    2    4
[3,]    3    9
[4,]    4   16
[5,]    5   25
[6,]    6   36
[7,]    7   49
[8,]    8   64
```

Впрочем, запись можно упростить при помощи функции `sapply()` (от *simplify apply*, то есть «упростить применение»). Вызов `sapply(x,f)` применяет функцию `f()` к каждому элементу `x`, а затем преобразует результат в матрицу. Пример:

```
> z12 <- function(z) return(c(z,z^2))
> sapply(1:8,z12)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    2    3    4    5    6    7    8
[2,]    1    4    9   16   25   36   49   64
```

Мы получим матрицу 2 × 8, а не 8 × 2, но в этом варианте она не менее полезна. Функция `sapply()` более подробно рассматривается в главе 4.

2.7. NA и NULL

Читателям с опытом программирования на других сценарных языках могут быть знакомы «неопределенные» значения, такие как `None` в Python или `undefined` в Perl. В R предусмотрено два таких значения: `NA` и `NULL`.

В статистических наборах данных часто встречаются пропущенные значения, которые в R представляются значением `NA`. С другой стороны, `NULL` указывает, что такого значения просто не существует — в отличие от значений существующих, но неизвестных. Посмотрим, как эти значения работают в конкретных ситуациях.

2.7.1. Значение NA

При вызове многих статистических функций R можно приказать функции пропустить любые отсутствующие значения, то есть `NA`. Пример:

```
> x <- c(88,NA,12,168,13)
>x
[1]  88  NA  12 168  13
> mean(x)
[1] NA
> mean(x,na.rm=T)
[1] 70.25
> x <- c(88,NULL,12,168,13)
> mean(x)
[1] 70.25
```

При первом вызове `mean()` отказывается вычислять результат, так как в `x` присутствует значение `NA`. Но если задать необязательному аргументу `na.rm` (*NA remove*, то есть «удалить NA») значение `TRUE` (`T`), функция вычислит среднее значение остальных элементов. При этом R автоматически пропускает значение `NULL`, которое будет описано в следующем разделе.

Существует несколько значений `NA`, по одному для каждого режима:

```
> x <- c(5,NA,12)
> mode(x[1])
[1] "numeric"
> mode(x[2])
[1] "numeric"
> y <- c("abc","def",NA)
> mode(y[2])
[1] "character"
> mode(y[3])
[1] "character"
```

2.7.2. Значение NULL

NULL часто применяется при построении векторов в циклах, когда при каждой итерации в вектор добавляется новый элемент. В следующем простом примере строится вектор нечетных чисел:

```
# Построение вектора с нечетными числами в интервале 1:10
> z <- NULL
> for (i in 1:10) if (i %%2 == 0) z <- c(z,i)
> z
[1]  2  4  6  8 10
```

Как упоминалось в главе 1, оператор %% возвращает остаток от деления. Например, выражение 13 %% 4 дает 1, так как остаток от деления 13 на 4 равен 1. (Список арифметических и логических операторов приведен в разделе 7.2.) Таким образом, цикл в этом примере начинает с вектора NULL, а потом добавляет в него элемент 2, затем элемент 4 и т. д.

Конечно, это искусственный пример, и у этой задачи есть намного лучшие решения. Приведу еще два способа нахождения нечетных чисел в интервале 1:10:

```
> seq(2,10,2)
[1]  2  4  6  8 10
> 2*1:5
[1]  2  4  6  8 10
```

Но этот пример нужен только для демонстрации различий между NA и NULL. Если использовать NA вместо NULL в приведенном примере, мы получим нежелательный элемент NA:

```
> z <-NA
> for (i in 1:10) if (i %%2 == 0) z <- c(z,i)
> z
[1] NA  2  4  6  8 10
```

Значения NULL действительно интерпретируются как несуществующие, что видно из следующего примера:

```
> u <- NULL
> length(u)
[1] 0
> v <-NA
> length(v)
[1] 1
```

NULL — специальный объект R, не имеющий режима.

2.8. Фильтрация

Еще одна возможность, отражающая функциональную природу языка R, — *фильтрация*. Этот механизм позволяет выбрать элементы вектора, удовлетворяющие некоторым условиям. Фильтрация — одна из самых распространенных операций в R, так как статистический анализ часто работает с данными, удовлетворяющими некоторым заданным условиям.

2.8.1. Генерирование индексов фильтрации

Начнем с простого примера:

```
> z <- c(5,2,-3,8)
> w <- z[z*z > 8]
> w
[1] 5  -3  8
```

Задавшись обычным вопросом «Что мы пытаемся сделать?», мы видим, что здесь мы приказываем R извлечь из `z` все элементы, квадраты которых больше восьми, и присвоить этот подвектор `w`.

Однако фильтрация в R играет настолько важную роль, что нам стоит изучить подробности того, как R достигает нужной цели. Давайте посмотрим, как это делается, шаг за шагом:

```
> z <- c(5,2,-3,8)
> z
[1] 5  2 -3  8
> z*z > 8
[1]  TRUE FALSE  TRUE  TRUE
```

Вычисление результата выражения `z*z > 8` дает вектор логических значений! Очень важно понимать, как это происходит.

Для начала отметим, что в выражении `z*z > 8` все компоненты являются векторами или векторными операторами:

- Так как `z` является вектором, это значит, что `z*z` также будет вектором (с такой же длиной, как у `z`).
- Из-за переработки число 8 (или вектор длины 1) превращается в вектор `(8,8,8,8)`.
- Оператор `>`, как и `+`, в действительности является функцией.

Следующий пример подтверждает последний пункт:

```
> ">"(2,1)
[1] TRUE
> ">"(2,5)
[1] FALSE
```

Таким образом, следующая запись:

```
z*z > 8
```

в действительности означает:

```
">"(z*z,8)
```

Иначе говоря, мы применяем функцию к векторам — еще один случай векторизации, не отличающийся от тех, что встречались вам ранее. А следовательно, результат является вектором — в данном случае вектором логических значений. Затем полученные логические значения используются для извлечения нужных элементов z:

```
> z[c(TRUE,FALSE,TRUE,TRUE)]
[1]  5  -3  8
```

В следующем примере все станет еще нагляднее. Здесь мы снова определяем условие извлечения в контексте z, но затем результаты используются для извлечения элементов из другого вектора (вместо извлечения из z):

```
> z <- c(5,2,-3,8)
> j <- z*z > 8
> j
[1]  TRUE FALSE  TRUE  TRUE
> y <- c(1,2,30,5)
> y[j]
[1]  1 30  5
```

Или в более компактном виде можно использовать следующую запись:

```
> z <- c(5,2,-3,8)
> y <- c(1,2,30,5)
> y[z*z > 8]
[1]  1 30  5
```

Еще раз: в этом примере один вектор z используется для определения индексов, по которым должен фильтроваться другой вектор y. В предыдущем примере вектор z использовался для фильтрации самого себя.

Другой пример — на этот раз с присваиванием. Допустим, имеется вектор x, в котором все элементы больше 3 нужно заменить нулем. Это можно сделать очень компактно; собственно, достаточно всего одной строки:

```
> x[x > 3]<-0
```

Проверяем:

```
> x <- c(1,3,8,2,20)
> x[x > 3]<-0
> x
[1]13020
```

2.8.2. Фильтрация с использованием функции subset()

Фильтрация также может выполняться с использованием функции `subset()`. Применительно к векторам различия между этой функцией и обычной фильтрацией заключаются в способе обработки значений NA.

```
> x <- c(6,1:3,NA,12)
> x
[1]  6  1  2  3 NA 12
> x[x > 5]
[1] 6 NA 12
> subset(x,x > 5)
[1]  6 12
```

Когда мы выполняем обычную фильтрацию в предыдущем разделе, R фактически говорит: «Что ж, значение `x[5]` неизвестно, но также неизвестно, больше ли его квадрат 5». Но возможно, вы не хотите присутствия NA в результатах. Если вы хотите исключить значения NA, функция `subset()` избавляет вас от хлопот с ручным удалением NA.

2.8.3. Функция выбора which()

Как вы уже видели, фильтрация заключается в извлечении элементов вектора z, удовлетворяющих некоторому условию. Впрочем, в некоторых случаях достаточно найти позиции z, для которых выполняется условие. Это можно сделать при помощи функции `which()`:

```
> z <- c(5,2,-3,8)
> which(z*z > 8)
[1] 1 3 4
```

Результат означает, что у элементов 1, 3 и 4 вектора z квадрат превышает 8. Как и с фильтрацией, важно точно понимать, что происходит в процессе выбора. Выражение

```
z*z > 8
```

дает результат (TRUE,FALSE,TRUE,TRUE). Функция which() затем просто сообщает, для каких элементов последнее выражение равно TRUE.

Удобное (хотя и несколько расточительное) применение функции which() — определение первой позиции вектора, для которой выполняется некоторое условие. Вспомните, как в коде на с. 55 производился поиск первого значения 1 в векторе x:

```
first1 <- function(x) {
   for (i in 1:length(x)) {
      if (x[i] == 1) break # Выйти из цикла
   }
   return(i)
}
```

Альтернативный способ программирования этой задачи:

```
first1a <- function(x) return(which(x == 1)[1])
```

Вызов which() возвращает индексы вхождений 1 в x. Эти индексы задаются в форме вектора, и мы запрашиваем индекс элемента 1 в этом векторе, который равен индексу первой единицы.

Такое решение получается куда более компактным. С другой стороны, оно недостаточно эффективно, потому что находятся *все* вхождения 1 в x, тогда как нам нужно только первое. Итак, несмотря на векторизацию решения, которая может ускорить его выполнение, если первая единица в x встречается близко к началу, такое решение может работать медленнее.

2.9. Векторизованная конструкция if-then-else: функция ifelse()

Кроме обычной конструкции if-then-else, присутствующей в большинстве языков, R также включает ее векторизованную версию: функцию ifelse(). Функция вызывается в следующей форме:

```
ifelse(b,u,v)
```

где b — логический вектор, а u и v — векторы.

Возвращаемое значение само по себе является вектором; элемент `i` равен `u[i]`, если значение `b[i]` истинно, или `v[i]`, если значение `b[i]` ложно. Концепция выглядит довольно абстрактно, поэтому перейдем сразу к примеру:

```
> x <- 1:10
> y <- ifelse(x %% 2 == 0,5,12) # %% — оператор вычисления остатка
> y
 [1] 12  5 12  5 12  5 12  5 12  5
```

В данном случае мы хотим создать вектор, который содержит 5 для четных элементов `x` или 12 для нечетных значений `x`. Итак, фактический аргумент, соответствующий формальному аргументу `b`, имеет вид `(F,T,F,T,F,T,F,T,F,T)`. Второй фактический аргумент 5, соответствующий `u`, интерпретируется как `(5,5,...)` (десять пятерок) из-за переработки. Третий аргумент 12 тоже перерабатывается в `(12,12,...)`.

Другой пример:

```
> x <- c(5,2,9,12)
> ifelse(x > 6,2*x,3*x)
[1] 15  6 18 24
```

Мы возвращаем вектор с элементами `x`, умноженными на 2 или на 3 в зависимости от того, превышает ли элемент 6.

И снова полезно продумать все, что здесь происходит. Выражение `x > 6` дает вектор с логическими элементами. Если `i`-й компонент истинен, то `i`-му элементу возвращаемого значения присваивается `i`-й элемент `2*x[i]`, в противном случае ему присваивается `3*x[i]`, и т. д.

Преимущество функции `ifelse()` перед стандартной конструкцией `if-then-else` — ее векторизация, которая может существенно ускорить выполнение.

2.9.1. Расширенный пример: мера связи

При оценке статистического соотношения двух переменных у стандартной меры корреляции (коэффициент корреляции Пирсона) есть много альтернатив. Возможно, некоторые читатели слышали о методе ранговой корреляции Спирмена. Эти альтернативные меры обладают разными свойствами (например, устойчивостью к *выбросам*, или *резко отклоняющимся значениям*, — аномальным и, возможно, ошибочным элементам данных).

Здесь мы предложим новую меру такого рода, и не для получения наград по статистике (хотя она и связана с одной популярной мерой такого рода — коэф-

фициентом корреляции (или тау) Кендалла), а для демонстрации некоторых методов программирования R, представленных в этой главе, особенно `ifelse()`.

Возьмем векторы `x` и `y`, содержащие временные ряды, — допустим, для измерений температуры воздуха и давления за каждый час. Мы определим нашу меру связи между ними как долю времени, в течение которого `x` и `y` возрастают или убывают одновременно, то есть долю `i`, для которых `y[i+1]-y[i]` имеет такой же знак, как `x[i+1]-x[i]`. Код выглядит так:

```
 1 # findud() преобразует вектор v в значения 1 и 0, которые показывают,
 2 # возрастает или нет значение элемента относительно
 3 # предыдущего; длина вывода на 1 меньше длины входного вектора.
 4 findud <- function(v) {
 5    vud <- v[-1] - v[-length(v)]
 6    return(ifelse(vud > 0,1,-1))
 7 }
 8
 9 udcorr <- function(x,y) {
10    ud <- lapply(list(x,y),findud)
11    return(mean(ud[[1]] == ud[[2]]))
12 }
```

Пример:

```
> x
 [1]  5 12 13  3  6  0  1  15 16  8 88
> y
 [1]  4  2  3 23  6 10 11 12  6  3  2
> udcorr(x,y)
[1] 0.4
```

В этом примере `x` и `y` одновременно растут в трех из десяти возможных случаев (первый раз с 12 до 13 и с 2 до 3) и одновременно убывают в одном случае. Таким образом, метрика связи равна 4/10 = 0,4.

Посмотрим, как это работает. Прежде всего нужно перекодировать `x` и `y` последовательностями 1 и –1 (значение 1 обозначает возрастание текущего наблюдения по сравнению с последним). Это было сделано в строках 5 и 6.

Например, подумайте, что просходит в строке 5 при вызове `findud()` с вектором `v` длиной, предположим, 16 элементов. Вектор `v[-1]` будет состоять из 15 элементов, начиная со второго элемента `v`. В результате мы вычитаем исходные серии из серий, полученных сдвигом вправо на один период времени. Разность дает последовательность состояний возрастания/убывания за каждый период времени — именно то, что нам нужно.

Затем эти разности нужно заменить 1 и –1 в зависимости от того, какой является разность — положительной или отрицательной. Вызов `ifelse()` решает эту задачу просто, компактно и с меньшим временем выполнения, чем версия кода с циклом.

Можно было записать два вызова `findud()`: для `x` и для `y`. Но если поместить `x` и `y` в список, а затем использовать `lapply()`, можно сделать это без дублирования кода. Если бы та же операция применялась ко многим векторам, а не только к двум (особенно с переменным количеством векторов), такое использование `lapply()` сделало бы код более ясным и компактным, а возможно, слегка ускорило бы его выполнение.

Затем вычисляется доля совпадений:

```
return(mean(ud[[1]] == ud[[2]]))
```

Обратите внимание: `lapply()` возвращает список. Компонентами являются наши векторы с кодировкой 1/–1. Выражение `ud[[1]] == ud[[2]]` возвращает вектор значений `TRUE` и `FALSE`, которые `mean()` интерпретирует как значения 1 и 0. В результате мы получаем нужную дробную величину.

Более сложная версия могла бы использовать функцию R `diff()`, которая выполняет *операции со сдвигом* для векторов. Например, можно сравнить каждый элемент с элементом, следующим через три позиции после него (*сдвиг 3*). Сдвиг по умолчанию составляет один временной период — то, что нам нужно в данном случае.

```
> u
[1] 1 6 7 2 3 5
> diff(u)
[1] 5 1 -5 1 2
```

Строка 5 в предыдущем примере приходит к следующему виду:

```
vud <- diff(v)
```

Чтобы код стал действительно компактным, используйте другую расширенную функцию R `sign()`, которая преобразует числа своего вектора-аргумента в 1, 0 или –1 в зависимости от их знака (положительные, нуль, отрицательные). Пример:

```
> u
[1] 1 6 7 2 3 5
> diff(u)
[1] 5 1 -5  1  2
> sign(diff(u))
[1] 1 1 -1  1  1
```

Функция sign() позволяет сократить функцию udcorr() до одной строки:

```
> udcorr <- function(x,y) mean(sign(diff(x)) == sign(diff(y)))
```

Конечно, эта функция намного короче исходной. Но лучше ли она? Вероятно, большинству программистов потребуется больше времени на то, чтобы ее написать. И хотя код получился коротким, пожалуй, понять его стало труднее.

Все программисты R должны искать свою «золотую середину» в соотношении между краткостью и ясностью кода.

2.9.2. Расширенный пример: перекодирование набора данных

Ввиду векторной природы аргументов возможны вложенные операции ifelse(). В следующем примере, использующем набор данных моллюсков-абалонов[1], пол кодируется в форме M, F или I (для молодых особей). Требуется закодировать эти характеристики в виде 1, 2 или 3. Реальный набор данных состоит из 4000 наблюдений, но для своего примера мы ограничимся несколькими наблюдениями, хранящимися в переменной g:

```
> g
[1] "M" "F" "F" "I" "M" "M" "F"
> ifelse(g == "M",1,ifelse(g == "F",2,3))
[1] 1 2 2 3 1 1 2
```

Что же на самом деле происходит во вложенном вызове ifelse()? Разберемся подробнее.

Во-первых, для конкретности выясним имена формальных аргументов функции ifelse():

```
> args(ifelse)
function (test, yes, no)
NULL
```

Напомню, что для каждого истинного элемента test функция дает соответствующий элемент yes. Если же значение test[i] ложно, функция дает no[i]. Все значения, сгенерированные таким образом, возвращаются совместно в виде вектора. В нашем примере R сначала выполнит внешний вызов ifelse(), в котором проверяется условие g=="M", а yes содержит 1 (с переработкой); no будет

[1] https://www.kaggle.com/yuridias/abalone-dataset

содержать результат выполнения `ifelse(g=="F",2,3)`. Теперь, поскольку элемент `test[1]` истинен, генерируется значение `yes[1]`, которое равно 1. Итак, первый элемент возвращаемого значения внешнего вызова равен 1.

Затем R проверяет `test[2]`. Это условие ложно, так что R потребуется `no[2]`. R теперь нужно выполнить внутренний вызов `ifelse()`. Ранее это еще не делалось, потому что до настоящего момента такой необходимости не было. R использует принцип *отложенного вычисления* — иначе говоря, выражение не вычисляется до того момента, пока в нем не возникнет необходимость в программе. Теперь R вычисляет `ifelse(g=="F",2,3)` и получает результат `(3,2,2,3,3,3,2)`; это вектор `no` для внешнего вызова `ifelse()`, так что вторым возвращаемым элементом последнего будет второй элемент `(3,2,2,3,3,3,2)`, или 2.

Когда внешний вызов `ifelse()` добирается до `test[4]`, он видит, что значение ложно, и возвращает `no[4]`. Так как вектор `no` уже был вычислен, у R есть нужное значение, то есть 3.

Вспомните, что задействованные векторы могут быть столбцами матриц — это очень распространенная ситуация. Допустим, данные абалонов хранятся в матрице `ab`, а пол приведен в первом столбце. Если вы захотите перекодировать данные по аналогии с предыдущим примером, это можно сделать так:

```
> ab[,1] <- ifelse(ab[,1] == "M",1,ifelse(ab[,1] == "F",2,3))
```

Предположим, требуется сгруппировать данные в соответствии с полом. Можно воспользоваться функцией `which()` для нахождения номеров элементов, соответствующих значениям `M`, `F` и `I`:

```
> m <- which(g == "M")
> f <- which(g == "F")
> i <- which(g == "I")
> m
[1] 1 5 6
> f
[1] 2 3 7
> i
[1] 4
```

Сделаем следующий шаг и сохраним эти группы в файле:

```
> grps <- list()
> for (gen in c("M","F","I")) grps[[gen]] <- which(g==gen)
> grps
$M
[1] 1 5 6
```

```
$F
[1] 2 3 7

$I
[1] 4
```

Здесь используется тот факт, что цикл `for()` в R может перебирать элементы строкового вектора. (Более эффективное решение будет показано в разделе 4.4.)

На основании перекодированных данных можно построить графики для анализа различных переменных в наборе данных абалонов. Чтобы природа переменных стала более понятной, добавим в файл следующий заголовок:

```
Gender,Length,Diameter,Height,WholeWt,ShuckedWt,ViscWt,ShellWt,Rings
```

Например, можно построить график зависимости диаметра от длины для мужских и женских особей, для чего будет использоваться следующий код:

```
aba <- read.csv("abalone.data",header=T,as.is=T)
grps <- list()
for (gen in c("M","F")) grps[[gen]] <- which(aba[,1]==gen)
abam <- aba[grps$M,]
abaf <- aba[grps$F,]
plot(abam$Length,abam$Diameter)
plot(abaf$Length,abaf$Diameter,pch="x",new=FALSE)
```

Сначала мы читаем набор данных и присваиваем его переменной `aba` (чтобы напомнить, что это данные абалонов). Вызов `read.csv()` аналогичен вызову `read.table()`, использованному в главе 1 (об этом будет более подробно рассказано в главах 6 и 10). Затем формируются `abam` и `abaf` — подматрицы `aba`, включающие мужские и женские особи соответственно.

На следующем шаге строятся графики. Первый вызов строит диаграмму разброса для зависимости диаметра от длины для мужских особей. Второй вызов предназначен для женских особей. Так как новая диаграмма должна быть наложена на уже существующую, передается аргумент `new=FALSE`, который приказывает R не создавать новый график. Аргумент `pch="x"` означает, что диаграмма для женских особей должна отображаться символами `x` вместо используемых по умолчанию символов `o`.

Диаграмма (для всего набора данных) изображена на рис. 2.1. Кстати говоря, она не идеальна: очевидно, между диаметром и длиной существует сильная корреляция, так что точки плотно заполняют часть пространства, а диаграммы для мужских и женских особей практически совпадают. (Впрочем, у мужских особей наблюдается больший разброс.) Это одна из стандартных проблем ста-

тистической графики. Более детализированный графический анализ мог бы стать более поучительным, но по крайней мере мы видим признаки сильной корреляции и то, что связь не так сильно изменяется между полами.

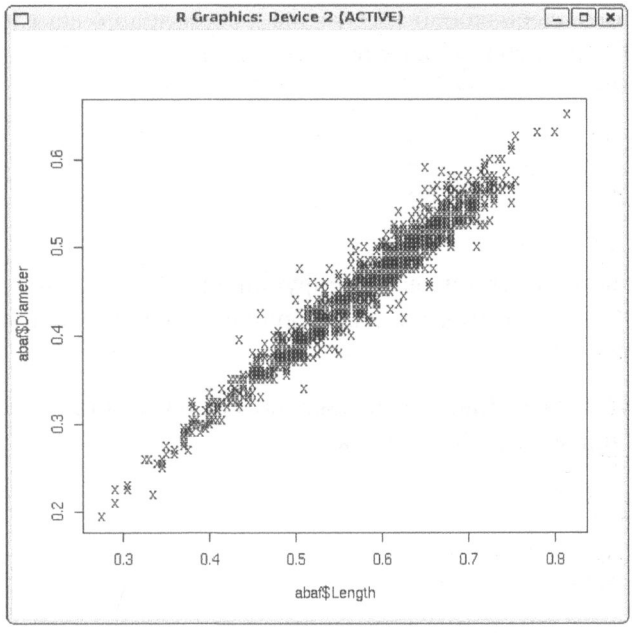


Рис. 2.1. Зависимость диаметра от длины для абалонов разных полов

Код построения диаграмм можно сделать более компактным при помощи еще одного использования `ifelse`. При этом используется тот факт, что параметр `pch` при выводе диаграммы может быть вектором вместо отдельного символа. Другими словами, R позволяет задать новый выводимый символ для каждой точки.

```
pchvec <- ifelse(aba$Gender == "M","o","x")
plot(aba$Length,aba$Diameter,pch=pchvec)
```

(Здесь перекодирование в 1, 2 и 3 опущено, но при желании вы можете сохранить его.)

2.10. Проверка равенства векторов

Предположим, нужно проверить два вектора на равенство. Наивное решение `==` не сработает.

```
> x <- 1:3
> y <- c(1,3,4)
> x==y
[1] TRUE FALSE FALSE
```

Что произошло? Здесь важно то, что мы имеем дело с векторизацией. Как и почти все в R, оператор == является функцией.

```
> "=="(3,2)
[1] FALSE
> i <-2
> "=="(i,2)
[1] TRUE
```

Вернее, == является векторизованной функцией. Выражение `x==y` применяет функцию `==()` к элементам `x` и `y`, а вектор представляет собой вектор логических значений.

Что еще можно сделать? Как вариант, можно воспользоваться векторизованной природой == и применить функцию `all()`:

```
> x <-1:3
> y <- c(1,3,4)
> x==y
[1] TRUE FALSE FALSE
> all(x == y)
[1] FALSE
```

Применяя `all()` к результату ==, вы спрашиваете, истинны ли все элементы последнего, что равносильно вопросу об идентичности `x` и `y`.

А еще лучше воспользоваться функцией `identical`:

```
> identical(x,y)
[1] FALSE
```

Но будьте осторожны, потому что *идентичность* в имени функции понимается буквально. Возьмем следующий фрагмент кода R:

```
> x <- 1:2
> y <- c(1,2)
> x
[1]12
> y
[1]12
> identical(x,y)
```

```
[1] FALSE
> typeof(x)
[1] "integer"
> typeof(y)
[1] "double"
```

Итак, : генерирует целые числа, а `c()` генерирует числа с плавающей точкой. Кто бы мог подумать?

2.11. Имена элементов векторов

Элементам векторов при желании можно присвоить имена. Например, допустим, что у вас имеется вектор из 50 элементов с населением каждого штата США. Элементам можно присвоить имена, соответствующие названиям штатов, — `"Montana"`, `"New Jersey"` и т. д. Например, это позволит выводить названия штатов на диаграммах.

Для назначения или чтения имен элементов векторов мы можем назначить или запросить имена элементов вектора через функцию `names()`:

```
> x <- c(1,2,4)
> names(x)
NULL
> names(x) <- c("a","b","ab")
> names(x)
[1] "a" "b" "ab"
> x
 a  b  ab
 1  2   4
```

Чтобы удалить имена из вектора, присвойте `NULL`:

```
> names(x) <- NULL
> x
[1]124
```

К элементам вектора даже можно обращаться по именам:

```
> x <- c(1,2,4)
> names(x) <- c("a","b","ab")
> x["b"]
b
2
```

2.12. Подробнее о c()

В этом разделе рассматриваются некоторые факты, относящиеся к функции конкатенации `c()`, которые часто оказываются полезными.

Если `c()` передаются аргументы разных режимов, они приводятся к «наименьшему общему знаменателю»:

```
> c(5,2,"abc")
[1] "5"   "2"   "abc"
> c(5,2,list(a=1,b=4))
[[1]]
[1] 5

[[2]]
[1] 2

$a
[1] 1

$b
[1] 4
```

В первом примере смешиваются режимы `integer` и `character`, а в такой комбинации R выбирает приведение ко второму режиму. Во втором примере R считает режим `list` наименее приоритетным в смешанных выражениях. Эта тема более подробно рассматривается в разделе 4.3.

Пожалуй, вам не стоит писать код, в котором создаются подобные комбинации, но вы можете встретить такой код, написанный другими разработчиками, поэтому важно понимать его эффект.

Также следует помнить еще об одном обстоятельстве: `c()` «сглаживает» содержимое векторов, как в следующем примере:

```
> c(5,2,c(1.5,6))
[1] 5.0 2.0 1.5 6.0
```

Пользователи с опытом работы на других языках (таких, как Python) могут предположить, что этот код создаст двухуровневый объект. С векторами R этого не происходит, хотя создать двухуровневый список возможно (см. главу 4).

Следующая глава посвящена очень важному частному случаю векторов, а именно матрицам и массивам.

3 Матрицы и массивы

Матрица представляет собой вектор, который содержит два дополнительных атрибута: количество строк и количество столбцов. Поскольку матрицы являются векторами, с ними также связывается режим — `numeric`, `character` и т. д. (С другой стороны, векторы не являются матрицами из одного столбца или одной строки.)

Матрицы являются особыми случаями более общего типа объектов в R — *массивов* (arrays). Массивы могут быть многомерными. Например, трехмерный массив состоит из строк, столбцов и слоев (а не только из строк и столбцов, как у матриц). Большая часть этой главы будет посвящена матрицам, но мы также кратко обсудим многомерные массивы в завершающем разделе.

Мощь R в значительной мере происходит от разнообразных операций, которые могут выполняться с матрицами. Эти операции будут рассмотрены в этой главе, и прежде всего аналоги векторных операций сегментирования и векторизации.

3.1. Создание матриц

Индексы строк и столбцов матриц начинаются с 1. Например, элемент в левом верхнем углу матрицы `a` обозначается `a[1,1]`. Во внутреннем представлении матрица размещается в памяти *по столбцам*: сначала идут все элементы столбца 1, затем все элементы столбца 2 и т. д., как было показано в разделе 2.1.3.

Для создания матрицы можно воспользоваться функцией `matrix()`:

```
> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

При вызове осуществляется конкатенация предполагаемых элементов первого столбца (1 и 2) и элементов, которые должны оказаться во втором столбце (3 и 4). Таким образом, данные имеют вид `(1,2,3,4)`. Затем задается количество строк и столбцов. Тот факт, что R использует порядок хранения по столбцам, определяет размещение этих четырех чисел в матрице.

Так как все четыре элемента матрицы были заданы явно, указывать `ncol` вместе с `nrow` необязательно; было бы достаточно одного из двух значений. Если элементов всего четыре, а строк две, из этого следует, что матрица состоит из двух столбцов:

```
> y <- matrix(c(1,2,3,4),nrow=2)
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

Обратите внимание на то, что при выводе `y` R выводит свои обозначения строк и столбцов. Например, `[,2]` обозначает весь столбец 2, как показывает следующая проверка:

```
> y[,2]
[1] 3 4
```

Другой способ построения матрицы `y` основан на перечислении отдельных элементов:

```
> y <- matrix(nrow=2,ncol=2)
> y[1,1] <- 1
> y[2,1] <- 2
> y[1,2] <- 3
> y[2,2] <- 4
> y
  [,1] [,2]
[1,] 1    3
[2,] 2    4
```

Обратите внимание: вы должны заранее предупредить R о том, что `y` является матрицей, и указать количество строк и столбцов.

Хотя во внутреннем представлении матрица размещается в памяти по столбцам, вы можете передать в аргументе `byrow` при вызове `matrix()` истинное значение, чтобы показать, что данные следуют по строкам. Пример использования `byrow`:

```
> m <- matrix(c(1,2,3,4,5,6),nrow=2,byrow=T)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

При этом матрица все равно хранится по столбцам — аргумент `byrow` только сообщает, что *входные данные* следуют по строкам. В частности, этот порядок может быть более удобным при чтении данных из файла, организованного подобным образом.

3.2. Общие операции с матрицами

Разобравшись с основами создания матриц, рассмотрим некоторые стандартные операции, связанные с матрицами. К их числу относятся операции линейной алгебры, индексирования и фильтрации матриц.

3.2.1. Выполнение операций линейной алгебры с матрицами

С матрицами могут выполняться различные операции линейной алгебры: умножение матриц, умножение матрицы на скаляр и сложение матриц. Вот как выполняются эти три операции с матрицей `y` из предыдущего примера:

```
> y %*% y # математическое умножение матриц
[,1] [,2]
[1,] 7 15
[2,]10 22
> 3*y # математическое умножение матрицы на скаляр
[,1] [,2]
[1,] 3 9
[2,] 6 12
> y+y # математическое сложение матриц
[,1] [,2]
[1,] 2 6
[2,] 4 8
```

За дополнительной информацией об операциях линейной алгебры с матрицами обращайтесь к разделу 8.4.

3.2.2. Индексирование матриц

Те же операции, которые рассматривались для векторов в разделе 2.4.2, могут применяться и к матрицам. Пример:

```
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> z[,2:3]
  [,1] [,2]
[1,] 1    1
[2,] 1    0
[3,] 0    1
[4,] 0    0
```

Здесь запрашивается подматрица z, состоящая из всех элементов с номерами столбцов 2 и 3 и любыми номерами строк. Иначе говоря, извлекаются второй и третий столбцы.

Пример извлечения строк вместо столбцов:

```
> y
  [,1] [,2]
[1,]11   12
[2,]21   22
[3,]31   32
> y[2:3,]
  [,1] [,2]
[1,]21   22
[2,]31   32
> y[2:3,2]
[1] 22 32
```

Также можно присваивать значения подматрицам:

```
> y
[,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
> y[c(1,3),] <- matrix(c(1,1,8,12),nrow=2)
> y
[,1] [,2]
```

```
[1,] 1 8
[2,] 2 5
[3,] 1 12
```

Здесь первой и третьей строке y присваиваются новые значения.

Другой пример присваивания подматрицам:

```
> x <- matrix(nrow=3,ncol=3)
> y <- matrix(c(4,5,2,3),nrow=2)
> y
     [,1] [,2]
[1,]    4    2
[2,]    5    3
> x[2:3,2:3] <- y
> x
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA    4    2
[3,]   NA    5    3
```

Отрицательные индексы, используемые с векторами для исключения некоторых элементов, аналогичным образом работают с матрицами:

```
> y
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> y[-2,]
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

Вторая команда запрашивает все строки y, кроме второй.

3.2.3. Расширенный пример: обработка графических изображений

Графические изображения по своей сути являются матрицами, поскольку пикселы организованы в строках и столбцах. Например, для изображения в оттенках серого хранится интенсивность (яркость) данного пиксела изображения. Таким образом, интенсивность пиксела, расположенного в строке 28 и столбце 88 изображения, хранится в строке 28 и столбце 88 матрицы. Для

цветного изображения хранятся три матрицы с интенсивностями красной, зеленой и синей составляющей, но мы ограничимся оттенками серого.

В этом примере будет использоваться изображение Национального мемориала горы Рашмор в Соединенных Штатах. Для чтения данных изображения будет использоваться библиотека `pixmap`. (О том, как загрузить и установить библиотеки, рассказано в приложении Б.)

```
> library(pixmap)
> mtrush1 <- read.pnm("mtrush1.pgm")
> mtrush1
Pixmap image
  Type : pixmapGrey
  Size : 194x259
  Resolution : 1x1
  Bounding box : 0 0 259 194
> plot(mtrush1)
```

Мы читаем файл с именем `mtrush1.pgm`, получая объект класса `pixmap`. После этого данные выводятся в графическом виде, как показано на рис. 3.1.

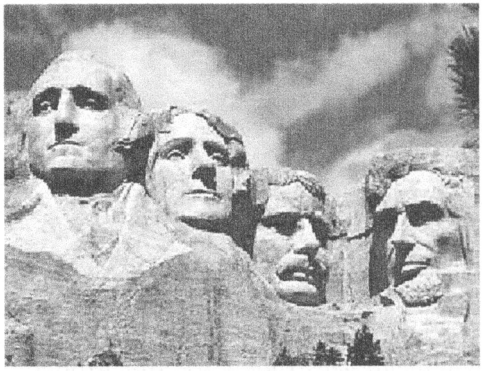


Рис. 3.1. Чтение данных изображения горы Рашмор

Теперь посмотрим, из чего состоит этот класс:

```
> str(mtrush1)
Formal class 'pixmapGrey' [package "pixmap"] with 6 slots
  ..@ grey : num [1:194, 1:259] 0.278 0.263 0.239 0.212 0.192 ...
  ..@ channels: chr "grey"
  ..@ size : int [1:2] 194 259
...
```

Класс относится к типу S4, поэтому его компоненты обозначаются символом `@` вместо `$`. Классы S3 и S4 будут рассматриваться в главе 9, а сейчас нас интересует матрица интенсивности `mtrush1@grey`. В нашем примере матрица состоит из 194 строк и 259 столбцов.

Интенсивности хранятся в виде чисел в диапазоне от 0,0 (черный) до 1,0 (белый); промежуточные значения буквально являются оттенками серого. Например, пиксел в строке 28 и столбце 88 имеет достаточно высокую яркость:

```
> mtrush1@grey[28,88]
[1] 0.7960784
```

Чтобы продемонстрировать матричные операции, сотрем президента Рузвельта (прости, Тедди, ничего личного). Для определения нужных строк и столбцов можно воспользоваться функцией R `locator()`. При вызове этой функции R ждет, когда пользователь щелкнет на точке внутри изображения, и возвращает точные координаты этой точки. Так я определил, что часть изображения с портретом Рузвельта расположена в строках с 84-й по 163-ю и столбцах с 135-го по 177-й. (Внимание: номера строк в объектах `pixmap` возрастают сверху вниз, тогда как `locator()` использует обратное направление.) Итак, чтобы стереть эту часть изображения, мы присваиваем всем пикселам в указанном диапазоне значение 1,0:

```
> mtrush2 <- mtrush1
> mtrush2@grey[84:163,135:177] <- 1
> plot(mtrush2)
```

Результат показан на рис. 3.2.

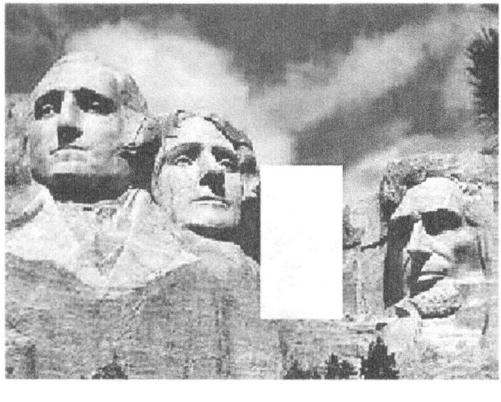


Рис. 3.2. Гора Рашмор без президента Рузвельта

А если вместо полного стирания мы захотим просто замаскировать личность президента Рузвельта? Это можно сделать, добавив в изображение случайный шум. Код выглядит так:

```
# Добавляет в img случайный шум в диапазоне rows,cols; img и возвращаемое
# значение являются объектами класса pixmap; параметр q
# управляет весом шума (используется доля 1-q от исходного изображения
# и q от случайного шума.)
blurpart <- function(img,rows,cols,q) {
   lrows <- length(rows)
   lcols <- length(cols)
   newimg <- img
   randomnoise <- matrix(nrow=lrows,ncol=lcols,runif(lrows*lcols))
   newimg@grey[rows,cols] <- (1-q) * img@grey[rows,cols] + q * randomnoise
   return(newimg)
}
```

Как видно из комментариев, мы генерируем случайный шум и затем берем средневзвешенное значение целевого пиксела и шума. Параметр q управляет весом шума, с большими значениями q повышается уровень размывки. Случайный шум представляет собой выборку из U(0,1) — равномерного распределения в интервале (0,1). Обратите внимание: следующая команда является матричной операцией:

```
newimg@grey[rows,cols] <- (1-q) * img@grey[rows,cols] + q * randomnoise
```

Посмотрим, как работает этот код:

```
> mtrush3 <- blurpart(mtrush1,84:163,135:177,0.65)
> plot(mtrush3)
```

Результат показан на рис. 3.3.

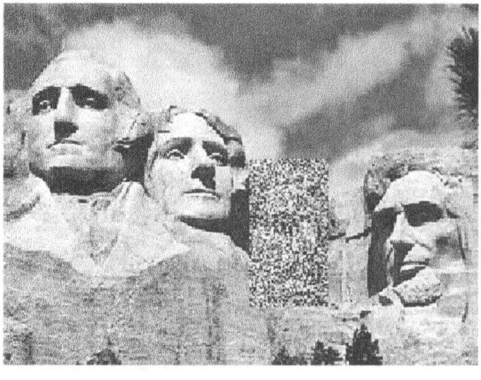


Рис. 3.3. Гора Рашмор с размывкой президента Рузвельта

3.2.4. Фильтрация матриц

Фильтрация может применяться не только к векторам, но и к матрицам. Впрочем, при этом необходима осторожность. Начнем с простого примера:

```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> x[x[,2] >= 3,]
     x
[1,] 2 3
[2,] 3 4
```

И снова проанализируем код так же, как это было сделано при первом знакомстве с фильтрацией в главе 2:

```
> j <- x[,2] >= 3
> j
[1] FALSE TRUE TRUE
```

Здесь мы рассмотрим вектор `x[,2]` — второй столбец `x` — и определим, какой из элементов больше либо равен 3. Результат, присваиваемый `j`, представляет собой логический вектор.

Теперь используем `j` в `x`:

```
> x[j,]
x
[1,] 2 3
[2,] 3 4
```

Здесь мы вычисляем `x[j,]`, то есть строки `x`, соответствующие истинным элементам `j`; в результате будут получены строки, соответствующие элементам столбца 2, не меньшим 3. Отсюда поведение, продемонстрированное ранее:

```
> x
     x
[1,] 1 2
[2,] 2 3
[3,] 3 4
> x[x[,2] >= 3,]
     x
[1,] 2 3
[2,] 3 4
```

По соображениям быстродействия стоит заметить, что вычисление `j` является полностью векторизированной операцией, поскольку все следующие утверждения истинны:

- Объект `x[,2]` является вектором.
- Оператор `>=` сравнивает два вектора.
- Число 3 было преобразовано в вектор с элементом 3.

Также следует заметить, что хотя `j` определяется в контексте `x`, а затем используется для извлечения элементов из `x`, это, в принципе, не обязательно. Критерий фильтрации может базироваться совсем не на той переменной, к которой фильтрация будет применена. Пример с той же матрицей `x`, что и выше:

```
> x[z %% 2 == 1,]
     [,1] [,2]
[1,] 1 2
[2,] 3 4
```

Здесь выражение `z %% 2 == 1` проверяет каждый элемент `z` на нечетность, что дает вектор `(TRUE,FALSE,TRUE)`. В результате извлекаются первая и третья строки `x`.

Другой пример:

```
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m[m[,1] > 1 & m[,2] > 5,]
[1] 3 6
```

Здесь используется тот же принцип, но с чуть более сложным набором условий извлечения строк (с извлечением столбцов или, в более общем случае, — любой подматрицы дело обстоит аналогично). Сначала выражение `m[,1] > 1` сравнивает каждый элемент первого столбца `m` с 1 и возвращает `(FALSE,TRUE,TRUE)`. Затем `(FALSE,TRUE,TRUE)` и `(FALSE,FALSE,TRUE)` объединяются логической операцией AND, которая дает результат `(FALSE,FALSE,TRUE)`. Последний используется для индексирования строк `m`, и мы получаем третью строку `m`.

Обратите внимание: в данном случае должен использоваться векторный оператор AND `&` вместо скалярного оператора `&&`, который бы следовало использовать в команде `&&`. Полный список таких операторов приведен в разделе 7.2.

Внимательный читатель мог заметить одну странность в предыдущем примере. Наша фильтрация должна была дать подматрицу размером 1 × 2, но вместо этого

дает вектор из двух элементов. Элементы были правильными, а тип данных — нет. Это создало бы проблемы при передаче данных другой матричной функции. Проблема решается аргументом `drop`, который приказывает R сохранить двумерную природу данных. Об аргументе `drop` мы подробно поговорим в разделе 3.6, когда будем анализировать непреднамеренное снижение размерности.

Так как матрицы являются векторами, к ним можно применять векторные операции. Пример:

```
> m
     [,1] [,2]
[1,]    5   -1
[2,]    2   10
[3,]    9   11
> which(m > 2)
[1] 1 3 5 6
```

R сообщает, что с точки зрения индексирования векторов элементы с индексами 1, 3, 5 и 6 матрицы `m` больше 2. Например, элемент с индексом 5 (строка 2, столбец 2) равен 10, что в самом деле больше 2.

3.2.5. Расширенный пример: генерирование ковариационной матрицы

Пример демонстрирует использование функций R `row()` и `col()`, аргументами которых являются матрицы. Например, для матрицы `a` вызов `row(a[2,8])` возвращает номер строки этого элемента `a`, который равен 2. Хотя мы и так знали, что `row(a[2,8])` относится к строке 2, не так ли? Тогда зачем нужна эта функция?

Рассмотрим пример. При написании кода моделирования многомерных нормальных распределений — например, с использованием `mvrnorm()` из библиотеки `MASS` — необходимо задать ковариационную матрицу. Для нас здесь важно прежде всего то, что матрица является симметричной: например, элемент в строке 1, столбце 2 равен элементу в строке 2, столбце 1.

Предположим, мы работаем с `n`-мерным нормальным распределением. Наша матрица состоит из `n` строк и `n` столбцов, и мы хотим, чтобы каждая из `n` переменных имела дисперсию 1 с корреляцией `rho` между парами переменных. Например, для `n` = 3 и `rho` = 0,2 матрица выглядит так:

$$\begin{pmatrix} 1 & 0{,}2 & 0{,}2 \\ 0{,}2 & 1 & 0{,}2 \\ 0{,}2 & 0{,}2 & 1 \end{pmatrix}$$

Приведу код для генерирования таких матриц:

```
1 makecov <- function(rho,n) {
2    m <- matrix(nrow=n,ncol=n)
3    m <- ifelse(row(m) == col(m),1,rho)
4    return(m)
5 }
```

Посмотрим, как работает этот код. Сначала, как вы, возможно, уже догадались, `col()` возвращает номер столбца своего аргумента (а `row()` делает то же самое для номера строки). Затем выражение `row(m)` в строке 3 возвращает матрицу целочисленных значений, каждое из которых указывает номер строки соответствующего элемента `m`. Пример:

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

Таким образом, выражение `row(m) == col(m)` в той же строке возвращает матрицу со значениями `TRUE` и `FALSE` — значения `TRUE` соответствуют значениям на диагонали матрицы, а значения `FALSE` — всем остальным элементам. И снова следует помнить, что бинарные операции, в данном случае `==`, являются функциями. Конечно, `row()` и `col()` тоже являются функциями, так что выражение

```
row(m) == col(m)
```

применяет эту функцию к каждому элементу матрицы `m` и возвращает матрицу `TRUE/FALSE` того же размера, что и `m`. Выражение `ifelse()` также является вызовом функции:

```
ifelse(row(m) == col(m),1,rho)
```

В данном случае, аргументом только что описанной матрицы является `TRUE/FALSE`, в результате значения 1 и `rho` размещаются в правильных позициях выходной матрицы.

3.3. Применение функций к строкам и столбцам матриц

Одна из самых известных и популярных возможностей R — семейство функций `*apply()` (таких, как `apply()`, `tapply()` и `lapply()`). Здесь будет рассмотрена функция `apply()`, которая приказывает R вызвать функцию, определенную пользователем, для каждой строки или каждого столбца матрицы.

3.3.1. Использование функции apply()

Обобщенная форма `apply` для матриц выглядит так:

```
apply(m,dimcode,f,fargs)
```

Аргументы имеют следующий смысл:

- `m` — матрица;
- `dimcode` — код измерения (1, если функция применяется к строкам, 2 для столбцов);
- `f` — применяемая функция;
- `fargs` — необязательный набор аргументов, применяемых к `f`.

В следующем примере функция R `mean()` применяется к каждому столбцу матрицы `z`:

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(z,2,mean)
[1] 2 5
```

В этом случае также можно было воспользоваться функцией `colMeans()`, но я хотел привести простой пример использования `apply()`.

Функции, написанные вами, так же законны для использования с `apply()`, как и любая встроенная функция R, такая как `mean()`. Пример использования пользовательской функции `f`:

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
  [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

Функция `f()` делит вектор из двух элементов на вектор `(2,8)`. (Если длина `x` больше 2, будет применена переработка.) Вызов `apply()` приказывает R вызвать `f()` для каждой из строк `z`. Первой такой строкой является строка `(1,4)`, и в вызове `f()` фактический аргумент, соответствующий формальному аргументу `x`, равен `(1,4)`. Таким образом, R вычисляет значение `(1,4)/(2,8)`, которое в поэлементной векторной арифметике R равно `(0.5,0.5)`. Вычисления для двух остальных строк выглядят аналогично.

Возможно, вас удивило, что результат имеет размеры 2 × 3 вместо 3 × 2. Первый результат `(0.5,0.5)` становится первым столбцом вывода `apply()` — не первой строкой! Но так уж работает `apply()`. Если применяемая функция возвращает вектор из `k` компонентов, то результат `apply()` будет состоять из `k` строк. При необходимости можно изменить ее функцией транспонирования матриц `t()`:

```
> t(apply(z,1,f))
     [,1]  [,2]
[1,]  0.5 0.500
[2,]  1.0 0.625
[3,]  1.5 0.750
```

Если функция возвращает скаляр (который, как мы знаем, представляет собой вектор из одного элемента), то итоговым результатом будет вектор, а не матрица.

Как видите, применяемая функция должна получать как минимум один аргумент. Здесь формальный аргумент будет соответствовать фактическому аргументу — одной строке или одному столбцу матрицы (см. выше). В некоторых случаях функции нужны дополнительные аргументы, которые могут следовать за именем функции в вызове `apply()`.

Предположим, имеется матрица из 1 и 0 и вы хотите создать вектор следующим образом: для каждой строки матрицы соответствующий элемент вектора будет

равен 1 или 0 в зависимости от того, какие значения составляют большинство в первых d элементах — 1 или 0. Здесь параметр d, возможно, потребуется изменять. Это можно сделать так:

```
> copymaj
function(rw,d) {
   maj <- sum(rw[1:d]) / d
   return(if(maj > 0.5) 1 else 0)
}
>x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    1    1    0
[2,]    1    1    1    1    0
[3,]    1    0    0    1    1
[4,]    0    1    1    1    0
> apply(x,1,copymaj,3)
[1] 1 1 0 1
> apply(x,1,copymaj,2)
[1] 0 1 0 0
```

Здесь значения 3 и 2 являются фактическими аргументами для формального аргумента d в `copymaj()`. Посмотрим, что происходит в случае строки 1 матрицы x. Строка состоит из элементов `(1,0,1,1,0)`, первыми d из которых были `(1,0,1)`. Большинство среди этих элементов составляли 1, поэтому `copymaj()` возвращает 1, а следовательно, первым элементом вывода `apply()` будет 1.

Вопреки распространенному мнению, использование `apply()` в общем случае не ускоряет код. Основные преимущества — исключительная компактность кода, который проще читается и изменяется, и предотвращение возможных ошибок при написании кода для циклов. Более того, по мере того как R подходит все ближе к параллельной обработке, такие функции, как `apply()`, начинают играть все более важную роль. Например, функция `clusterApply()` из пакета `snow` наделяет R некоторыми возможностями параллельной обработки за счет распределения данных подматриц по различным сетевым узлам; при этом каждый узел фактически применяет заданную функцию к своей подматрице.

3.3.2. Расширенный пример: поиск выбросов

В статистике *выбросами* называются точки данных, сильно отличающиеся от других наблюдений. Они обычно рассматриваются как подозрительные (воз-

можная ошибка ввода) или нерепрезентативные (как доход Билла Гейтса на уровне доходов граждан штата Вашингтон). Ученые разработали много методов выявления выбросов. Здесь мы создадим очень простую реализацию.

Допустим, в матрице `rs` хранятся данные розничных продаж. Каждая строка данных представляет отдельный магазин, а наблюдения в строке представляют данные ежедневных продаж. Воспользуемся простым (и несомненно, излишне простым) методом и напишем код для выявления большинства аномальных наблюдений для каждого магазина. Выброс будет определяться как наблюдение, в наибольшей степени отклоняющееся от медианного значения для этого магазина. Код выглядит так:

```
1 findols <- function(x) {
2    findol <- function(xrow) {
3       mdn <- median(xrow)
4       devs <- abs(xrow-mdn)
5       return(which.max(devs))
6    }
7    return(apply(x,1,findol))
8 }
```

А вот как выглядит вызов:

```
findols(rs)
```

Как он работает? Сначала нужно указать функцию в вызове `apply()`.

Так как функция будет применяться к каждой строке матрицы продаж, из описания следует, что она должна сообщать индекс самого аномального наблюдения в заданной строке. Наша функция `findol()` делает это в строках 4 и 5. (Обратите внимание: здесь одна функция определяется внутри другой; это обычная практика, если внутренняя функция достаточно короткая.) В выражении `xrow-mdn` число, то есть одноэлементный вектор, вычитается из вектора, длина которого в общем случае выше 1. Соответственно, перед вычитанием `mdn` расширяется до размеров `xrow`.

Затем в строке 5 используется функция R `which.max()`. Вместо поиска максимального значения в векторе, что делает функция `max()`, `which.max()` сообщает, где находится это максимальное значение, то есть индекс вхождения. Это именно то, что нам нужно.

Наконец, в строке 7 мы приказываем R применить `findol()` к каждой строке `x` и получаем индексы самого аномального наблюдения в каждой строке.

3.4. Добавление и удаление строк и столбцов матриц

С технической точки зрения матрицы имеют фиксированную длину и размеры, поэтому добавлять и удалять строки и столбцы невозможно. Однако к матрицам можно применять *повторное присваивание*, а это позволяет достичь того же эффекта, как если бы операции добавления и удаления выполнялись с ними напрямую.

3.4.1. Изменение размера матрицы

Вспомните, как производилось повторное присваивание векторов для изменения их размеров:

```
> x
[1] 12 5 13 16 8
> x <- c(x,20) # Присоединить 20
> x
[1] 12  5 13 16  8 20
> x <- c(x[1:3],20,x[4:6]) # Вставить 20
> x
[1] 12  5 13 20 16  8 20
> x <- x[-2:-4] # Удалить элементы со 2 по 4
> x
[1] 12 16  8 20
```

В первом случае x изначально имеет длину 5, которая увеличивается до 6 посредством конкатенации с последующим повторным присваиванием. Длина x не изменяется напрямую; вместо этого мы создаем новый вектор на базе x и присваиваем x этот новый вектор.

ПРИМЕЧАНИЕ

Повторное присваивание происходит даже в том случае, если вы его не замечаете (см. главу 14). Например, даже при безобидном на первый взгляд присваивании `x[2]<-12` в действительности происходит повторное присваивание.

Аналогичные операции могут использоваться для изменения размера матрицы. Например, функции `rbind()` (*row bind*) и `cbind()` (*column bind*) позволяют добавлять строки и столбцы в матрицу.

```
> one
[1] 1 1 1 1
> z
  [,1] [,2] [,3]
[1,] 1    1    1
[2,] 2    1    0
[3,] 3    0    1
[4,] 4    0    0
> cbind(one,z)
[1,]1 1 1 1
[2,]1 2 1 0
[3,]1 3 0 1
[4,]1 4 0 0
```

Здесь `cbind()` создает новую матрицу, объединяя единичный столбец со столбцами `z`. Мы выбрали простой вывод, но с таким же успехом можно было присвоить результат `z` (или другой переменной):

```
z <- cbind(one,z)
```

Также стоит заметить, что мы могли воспользоваться переработкой:

```
> cbind(1,z)
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    2    1    0
[3,]    1    3    0    1
[4,]    1    4    0    0
```

Здесь значение 1 было расширено в вектор из четырех значений 1.

Функции `rbind()` и `cbind()` также могут использоваться для быстрого создания небольших матриц. Пример:

```
> q <- cbind(c(1,2),c(3,4))
> q
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Впрочем, при использовании `rbind()` и `cbind()` необходима осторожность. Создание матрицы, как и создание вектора, является затратной по времени операцией (в конце концов, матрицы представляют собой векторы). В следующем коде `cbind()` создает новую матрицу:

```
z <- cbind(one,z)
```

Новая матрица присваивается z; другими словами, мы назначаем ей имя z — такое же, как у исходной матрицы, которая перестала существовать. Однако при этом на создание матрицы тратится лишнее время. Если эта операция будет многократно выполняться в цикле, накопленные затраты будут достаточно заметными.

Итак, если вы последовательно добавляете строки/столбцы в цикле, лучше сразу выделить память под большую матрицу. Сначала она будет пустой, но строки или столбцы будут заполняться раз за разом (вместо долгого выделения памяти под матрицу при каждой итерации).

Строки и столбцы также можно удалять повторным присваиванием:

```
> m <- matrix(1:6,nrow=3)
> m
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> m <- m[c(1,3),]
> m
     [,1] [,2]
[1,]    1    4
[2,]    3    6
```

3.4.2. Расширенный пример: поиск ближайшей пары вершин в графе

Определение расстояний между вершинами графа — типичный пример, используемый в учебном курсе информатики и в статистике/теории обработки данных. В частности, подобные задачи встречаются в алгоритмах кластеризации и в задачах геномики.

Здесь мы рассмотрим стандартный пример определения расстояний между городами, поскольку оно проще описывается, чем, скажем, определение расстояний между цепями ДНК.

Допустим, вы хотите написать функцию, которая получает на входе матрицу расстояний (элемент в строке i и столбце j определяет расстояние между городом i и городом j) и выдает минимальное расстояние одного перехода между городами и пару городов, для которых достигается минимум. Код решения выглядит так:

```
# Возвращает минимальное значение d[i,j], i != j, и строку/столбец,
# для которых достигается минимум; случай равенства не обрабатывается.
mind <- function(d) {
   n <- nrow(d)
   # добавить столбец для определения номера строки для apply()
   dd <- cbind(d,1:n)
   wmins <- apply(dd[-n,],1,imin)
   # Размер wmins равен 2xn: 1 строка – индексы, 2 – значения
   i <- which.min(wmins[2,])
   j <- wmins[1,i]
   return(c(d[i,j],i,j))
}

# Найти позицию и значение минимума в строке x
imin <- function(x) {
   lx <- length(x)
   i <- x[lx] # Исходный номер строки
   j <- which.min(x[(i+1):(lx-1)])
   k<-i+j
   return(c(k,x[k]))
}
```

Пример практического применения новой функции:

```
> q
     [,1] [,2] [,3] [,4] [,5]
[1,]    0   12   13    8   20
[2,]   12    0   15   28   88
[3,]   13   15    0    6    9
[4,]    8   28    6    0   33
[5,]   20   88    9   33    0
> mind(q)
[1] 6 3 4
```

Минимальное значение 6 находится в строке 3, столбец 4. Как видите, вызов `apply()` играет важную роль.

Задача достаточно проста: нужно найти в матрице минимальный ненулевой элемент. Мы находим минимум в каждой строке (один вызов `apply()` делает это для всех строк), после чего находим наименьшее значение среди минимумов. Но как вы увидите, логика кода становится довольно нетривиальной.

Важно, что матрица является симметричной, потому что расстояние от города `i` до города `j` равно расстоянию от `j` до `i`. Таким образом, чтобы найти минимальное значение в `i`-й строке, нужно проверить только элементы `i + 1`, `i + 2`, ...,

n, где n — количество строк и столбцов в матрице. Кроме того, это позволяет пропустить последнюю строку d при вызове apply() в строке 7.

Так как матрица может быть очень большой (с тысячами городов матрица будет содержать миллионы элементов), нам стоит воспользоваться симметрией и избавиться от лишней работы.

Однако при этом возникает проблема. Чтобы выполнить основные вычисления, функция, вызываемая apply(), должна знать номер строки в исходной матрице, — apply() эту информацию не предоставляет. Таким образом, в строке 6 матрица дополняется лишним столбцом с номерами строк, чтобы они могли использоваться функцией, вызываемой apply().

Такой функцией будет функция imin(), которая начинается в строке 15 и находит минимум в строке, заданной в формальном аргументе x. Она возвращает не только минимум в заданной строке, но и индекс, в котором этот минимум был обнаружен. Так, при вызове imin() для строки 1 матрицы q из приведенного примера минимальное значение 8 имеет индекс 4. Задача решается при помощи очень удобной функции R which.min() (строка 18).

Строка 19 также заслуживает внимания. Вспомните, что из-за симметричности матрицы начальная часть каждой строки пропускается, как видно в выражении (i+1):(lx-1) в строке 18. Но это означает, что вызов which.min() в этой строке вернет индекс минимума относительно диапазона (i+1):(lx-1). В строке 3 матрицы q из нашего примера будет получен индекс 1 вместо 4. А значит, в результат нужно внести поправку и прибавить i, что делается в строке 19.

Наконец, правильно использовать вывод apply() тоже непросто. Взгляните на приведенную выше матрицу q. Вызов apply() вернет матрицу wmins:

$$\begin{pmatrix} 4 & 3 & 4 & 5 \\ 8 & 15 & 6 & 33 \end{pmatrix}$$

Как упоминалось в комментариях, вторая строка этой матрицы содержит минимумы разных строк d по верхней диагонали, тогда как первая строка содержит индексы этих значений. Например, первый столбец wmins дает информацию о первой строке q и сообщает, что наименьшее значение в этой строке 8 находится в элементе с индексом 4 этой строки.

Таким образом, строка 9 выберет i-е число в строке, содержащей наименьшее значение во всей матрице, — 6 в нашем примере с q. Строка 10 дает позицию j этой строки, в которой находится минимум (4 в случае q). Другими словами,

общий минимум находится в строке i и столбце j — информация, которая будет использоваться в строке 11.

В строке 1 вывода apply() содержатся индексы строк, в которых достигаются минимумы по строкам. И это очень полезная информация, потому что теперь мы можем определить, какой другой город задействован в лучшей паре. Мы знаем, что город 3 является одним из них, поэтому переходим к элементу 3 строки 1 вывода и узнаем, что это 4. Таким образом, пару городов, расположенных ближе всего друг к другу, составляют города 3 и 4. Строки 9 и 10 реализуют эту логику.

Если минимальный элемент матрицы уникален, то существует альтернативное, намного более простое решение:

```
minda <- function(d) {
   smallest <- min(d)
   ij <- which(d == smallest,arr.ind=TRUE)
   return(c(smallest,ij))
}
```

Такое решение работает, но у него есть некоторые потенциальные недостатки. В новом коде самой важной является следующая строка:

```
ij <- which(d == smallest,arr.ind=TRUE)
```

Она определяет индекс элемента d, при котором достигается минимум. Аргумент arr.ind=TRUE указывает, что возвращаемый индекс является индексом матрицы, то есть состоит из строки и столбца. Без этого аргумента d будет рассматриваться как вектор.

Как упоминалось выше, этот код работает только в том случае, если минимум уникален. Если это условие не выполнено, which() вернет несколько пар «строка/столбец», что противоречит нашей основной цели. С другой стороны, если бы использовался исходный код, а d имело несколько минимальных элементов, то возвращался бы только один из них.

Другой проблемой является быстродействие. Новый код фактически выполняет два (внешне незаметных) цикла по матрице: один вычисляет минимум, а другой содержится в вызове which(). Скорее всего, он будет медленнее исходного кода.

Из этих двух решений исходный код выбирается в том случае, если скорость выполнения критична или матрица содержит несколько минимумов. В остальных случаях следует выбирать альтернативный код; простота последнего решения существенно упрощает понимание и сопровождение кода.

3.5. Чем векторы отличаются от матриц

В начале этой главы я сказал, что матрица представляет собой обычный вектор с двумя дополнительными атрибутами: количеством строк и количеством столбцов. Здесь векторная природа матриц будет рассмотрена более подробно. Пример:

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

Поскольку z все равно является вектором, вы можете запросить его длину:

```
> length(z)
[1] 8
```

Но z как матрица представляет собой несколько большее, чем вектор:

```
> class(z)
[1] "matrix"
> attributes(z)
$dim
[1] 4 2
```

Другими словами, здесь действительно существует *класс* `matrix` в смысле объектно-ориентированного программирования. Как упоминалось в главе 1, в R в основном используются классы S3, компоненты которых обозначаются знаком `$`. Класс `matrix` имеет один атрибут с именем `dim`, который представляет собой вектор с количеством строк и столбцов матрицы. Классы будут подробно рассматриваться в главе 9.

Также значение `dim` можно получить при помощи функции `dim()`:

```
> dim(z)
[1] 4 2
```

Количество строк и столбцов можно получить по отдельности при помощи функций `nrow()` и `ncol()`:

```
> nrow(z)
[1] 4
> ncol(z)
[1] 2
```

Эти функции всего лишь используют `dim()`, в чем нетрудно убедиться. Вспомните, что для вывода объекта в интерактивном режиме достаточно ввести его имя:

```
> nrow
function (x)
dim(x)[1]
```

Эти функции полезны в том случае, если вы пишете библиотечную функцию общего назначения, аргументом которой является матрица. Имея возможность определить количество строк и столбцов в коде, вы избавляете сторону вызова от хлопот с передачей этой информации в двух дополнительных аргументах. Это одно из преимуществ объектно-ориентированного программирования.

3.6. Предотвращение непреднамеренного снижения размерности

В мире статистики сокращение размерности считается положительным явлением, а для его качественного выполнения существует много статистических процедур. Если вы работаете, скажем, с десятью переменными и это число можно сократить до трех, которые отражают суть данных, — то это очень хорошо.

Однако в R под сокращением размерности может пониматься другое явление, нежелательное в некоторых случаях. Допустим, у нас есть матрица из четырех строк, из которой извлекается одна строка:

```
> z
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
> r <- z[2,]
> r
[1] 2 6
```

Вроде бы выглядит безобидно, но обратите внимание на формат, в котором R выводит `r`. Это формат вектора, а не матрицы. Другими словами, `r` является вектором длины 2, а не матрицей 1 × 2. В этом можно убедиться парой способов:

```
> attributes(z)
$dim
[1] 4 2
```

```
> attributes(r)
NULL
> str(z)
int [1:4, 1:2] 1 2 3 4 5 6 7 8
> str(r)
int [1:2] 2 6
```

Здесь R сообщает о том, что у `z` есть номера строк и столбцов, а у `r` их нет. Аналогичным образом `str()` сообщает, что индексы `z` лежат в диапазонах `1:4` и `1:2`, а индексы `r` лежат в диапазоне `1:2`. Никаких сомнений нет — `r` является вектором, а не матрицей.

Все выглядит естественно, но во многих случаях это создаст проблемы в программах, выполняющих множество матричных операций. Может оказаться, что в целом ваш код нормально работает, но перестает работать в каком-то частном случае. Представьте, что ваш код извлекает подматрицу из заданной матрицы, после чего выполняет некоторые матричные операции с подматрицей. Если подматрица состоит только из одной строки, R превратит ее в вектор, что нарушит ваши вычисления.

К счастью, в R можно предотвратить такое снижение размерности: аргументом `drop`. В следующем примере используется матрица `z` из предыдущего примера:

```
> r <- z[2,, drop=FALSE]
> r
     [,1] [,2]
[1,]    2    6
> dim(r)
[1] 1 2
```

Теперь `r` становится матрицей 1 × 2, а не вектором из двух элементов. По этим причинам вам стоит привыкнуть к тому, чтобы включать аргумент `drop=FALSE` в код всех операций с матрицами.

Почему мы говорим о `drop` как об аргументе? Потому что `[` в действительности является функцией, как и в случае с операторами типа `+`. Возьмем следующий код:

```
> z[3,2]
[1] 7
> "["(z,3,2)
[1] 7
```

Если у вас имеется вектор, который должен интерпретироваться как матрица, используйте функцию `as.matrix()`:

```
> u
[1] 1 2 3
> v <- as.matrix(u)
> attributes(u)
NULL
> attributes(v)
$dim
[1] 3 1
```

3.7. Назначение имен для строк и столбцов матриц

К элементам матриц естественно обращаться по номерам строк и столбцов матрицы. Тем не менее строкам и столбцам также можно назначить имена. Пример:

```
> z
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> colnames(z)
NULL
> colnames(z) <- c("a","b")
> z
     a b
[1,] 1 3
[2,] 2 4
> colnames(z)
[1] "a" "b"
> z[,"a"]
[1] 1 2
```

Как видно из примера, эти имена могут использоваться для обращения к конкретным столбцам. Функция `rownames()` работает аналогично.

Имена строк и столбцов обычно не настолько важны при написании кода R для приложений общего назначения, но они могут пригодиться при анализе конкретных наборов данных.

3.8. Массивы более высокой размерности

В статистическом контексте строки типичной матрицы R соответствуют наблюдениям (допустим, данным различных людей), а столбцы соответствуют

переменным (например, весу и кровяному давлению). Но предположим, измерения также выполняются в разное время, по одной точке данных для каждого человека в каждый момент времени. Время становится третьим измерением в дополнение к строкам и столбцам. В R такие наборы данных называются *массивами* (arrays).

Простой пример: возьмем пример со студентами и оценками за тесты. Допустим, каждый тест состоит из двух частей, поэтому для каждого теста каждого студента сохраняются две оценки. Для удобства будем считать, что имеются всего два теста и три студента. Данные первого теста:

```
> firsttest
     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   50
```

Студент 1 получил 46 и 30 баллов по результатам первого теста, студент 2 получил 21 и 25 баллов и т. д. А вот оценки тех же студентов на втором тесте:

```
> secondtest
     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   50
```

Теперь объединим данные обоих тестов в одну структуру данных с именем `tests`. Она будет состоять из двух «уровней» (по одному на каждый тест) с тремя строками и двумя столбцами на каждый уровень. Данные `firsttest` будут храниться на первом уровне, а данные `secondtest` — на втором.

На уровне 1 три строки содержат данные трех студентов по результатам первого теста, по два столбца на каждую строку для двух частей теста. Для создания структуры данных используется функция R `array`:

```
> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))
```

В аргументе `dim=c(3,2,2)` определяются два уровня (вторая 2), каждый из которых состоит из трех строк и двух столбцов. Далее это значение становится атрибутом структуры данных:

```
> attributes(tests)
$dim
[1] 3 2 2
```

Каждый элемент `tests` теперь имеет три индекса вместо двух, как в случае с матрицей. Первый индекс соответствует первому элементу вектора `$dim`, второй — второму элементу вектора и т. д. Например, результат второй части теста 1 для студента 3 может быть получен следующим образом:

```
> tests[3,2,1]
[1] 4 8
```

Функция R `print` для массивов выводит данные по уровням:

```
> tests
, , 1

     [,1] [,2]
[1,]   46   30
[2,]   21   25
[3,]   50   48

, , 2

     [,1] [,2]
[1,]   46   43
[2,]   41   35
[3,]   50   49
```

Подобно тому как трехмерный массив был построен объединением двух матриц, можно строить четырехмерные массивы объединием двух и более трехмерных массивов и т. д.

Массивы очень часто используются для вычисления таблиц. Пример трехмерной таблицы приведен в разделе 6.3.

4 Списки

В отличие от вектора, все элементы которого должны иметь одинаковый режим, структура списка `list` может объединять объекты разных типов. Для читателей, знакомых с Python, списки R напоминают словари Python (или хеши Perl). Программисты C заметят сходство со структурами `struct` языка C. Списки играют центральную роль в R: они закладывают основу для кадров данных, объектно-ориентированного программирования и т. д.

В этой главе вы узнаете, как создавать списки и как работать с ними. Как и при работе с векторами и матрицами, одной из распространенных операций со списками является индексирование. Механизм индексирования списков имеет много общего с индексированием векторов и матриц, но есть и принципиальные различия. Как и у матриц, у списков существует аналог функции `apply()`. В этой главе мы обсудим эти и другие возможности списков, включая средства разбора списков на элементы, что иногда бывает удобно.

4.1. Создание списков

Технически список является вектором. Обычные векторы, то есть векторы типа, который использовался до настоящего момента, называются *атомными* векторами, поскольку их компоненты не могут быть разбиты на меньшие компоненты. Списки иногда называются *рекурсивными векторами*.

Для того, чтобы начать работу со списками, рассмотрим базу данных работников. Для каждого работника в базе данных должно храниться имя, зарплата и логический признак принадлежности к профсоюзу. Так как в данных используются три разных режима — `character`, `numeric` и `logical`, — для их хранения идеально подойдут списки. Вся база данных может быть реализована в виде списка списков или другой разновидности списка (например, кадра данных, хотя эта возможность здесь не рассматривается).

Например, данные работника Джо в списке могут выглядеть так:

```
j <- list(name="Joe", salary=55000, union=T)
```

После этого вы сможете вывести объект j полностью или по компонентам:

```
> j
$name
[1] "Joe"
$salary
[1] 55000
$union
[1] TRUE
```

На самом деле имена компонентов (в литературе по R они называются *тегами*), такие как salary, необязательны. С таким же успехом можно поступить так:

```
> jalt <- list("Joe", 55000, T)
> jalt
[[1]]
[1] "Joe"

[[2]]
[1] 55000

[[3]]
[1] TRUE
```

Тем не менее обычно считается, что код с именами вместо числовых индексов более понятен и менее подвержен риску ошибок.

Имена компонентов списков можно сокращать насколько угодно, лишь бы это не создавало неоднозначности:

```
> j$sal
[1] 55000
```

Так как списки являются векторами, они могут создаваться вызовом vector():

```
> z <- vector(mode="list")
> z[["abc"]] <- 3
> z
$abc
[1] 3
```

4.2. Общие операции со списками

Итак, мы рассмотрели простой пример создания списков. Теперь посмотрим, как обращаться к содержимому списков и работать с ними.

4.2.1. Индексирование списков

К компонентам списков можно обращаться несколькими разными способами:

```
> j$salary
[1] 55000
> j[["salary"]]
[1] 55000
> j[[2]]
[1] 55000
```

К компонентам списков можно обращаться по индексам, рассматривая список как вектор. Однако обратите внимание на то, что в этом случае используются двойные квадратные скобки вместо одинарных.

Итак, существуют три способа обратиться к отдельному компоненту `c` списка `lst` и вернуть его в типе данных `c`:

- `lst$c`;
- `lst[["c"]]`;
- `lst[[i]]`, где `i` — индекс `c` в `lst`.

Все эти способы полезны в разных ситуациях, как вы увидите в последующих примерах. Но обратите внимание на уточнение: «вернуть его в типе данных `c`»; во втором и третьем из представленных способов также можно использовать одиночные квадратные скобки вместо двойных:

- `lst["c"]`;
- `lst[i]`, где `i` — индекс `c` в `lst`.

Как с одинарными, так и с двойными квадратными скобками мы обращаемся к элементам списков по аналогии с векторами. Однако в данном случае существует важное отличие от индексирования обычных (атомных) векторов: при использовании одинарных квадратных скобок `[]` результат представляет собой другой список — подсписок исходного. В продолжение предыдущего примера получаем следующее:

```
> j[1:2]
$name
[1] "Joe"

$salary
[1] 55000
> j2 <- j[2]
> j2
$salary
[1] 55000
> class(j2)
[1] "list"
> str(j2)
List of 1
 $ salary: num 55000
```

Операция сегментирования вернула другой список, состоящий из первых двух компонентов исходного списка `j`. Обратите внимание: слово «вернула» здесь вполне оправданно, поскольку квадратные скобки индексирования являются функциями. Происходящее аналогично другим ситуациям с операторами, которые на первый взгляд не похожи на функции, — например, `+`.

С другой стороны, двойные квадратные скобки `[[ ]]` могут использоваться для обращения к одному компоненту, при этом тип результата соответствует типу этого компонента.

```
> j[[1:2]]
Error in j[[1:2]] : subscript out of bounds
> j2a <- j[[2]]
> j2a
[1] 55000
> class(j2a)
[1] "numeric"
```

4.2.2. Добавление и удаление элементов списка

Операции добавления и удаления элементов списка встречаются в чрезвычайно широком наборе контекстов. Это особенно справедливо для структур данных, построенных на основе списков, таких как кадры данных и классы R.

Новые компоненты могут добавляться после создания списка:

```
> z <- list(a="abc",b=12)
> z
$a
[1] "abc"
```

```
$b
[1] 12

> z$c <- "sailing" # Добавить компонент c
> # Проверить, появился ли компонент?
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] "sailing"
```

Добавление компонентов также может осуществляться индексированием вектора:

```
> z[[4]] <- 28
> z[5:7] <- c(FALSE,TRUE,TRUE)
> z
$a
[1] "abc"

$b
[1] 12

$c
[1] "sailing"

[[4]]
[1] 28

[[5]]
[1] FALSE

[[6]]
[1] TRUE

[[7]]
[1] TRUE
```

Чтобы удалить компонент списка, присвойте ему `NULL`:

```
> z$b <- NULL
> z
$a
[1] "abc"
$c
[1] "sailing"
```

```
[[3]]
[1] 28

[[4]]
[1] FALSE

[[5]]
[1] TRUE

[[6]]
[1] TRUE
```

Учтите, что после удаления `z$b` индексы элементов, следующих за ним, сдвигаются вверх на 1. Например, бывший элемент `z[[4]]` превращается в `z[[3]]`.

Также возможна конкатенация списков:

```
> c(list("Joe", 55000, T),list(5))
[[1]]
[1] "Joe"

[[2]]
[1] 55000

[[3]]
[1] TRUE

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

4.2.3. Получение размера списка

Так как список является вектором, для получения количества компонентов в списке можно воспользоваться функцией `length()`.

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

4.2.4. Расширенный пример: поиск слов

В наши дни в области веб-поиска и других видов глубокого анализа текстовых данных ведутся активные исследования. Наш следующий пример работы со списками в R относится к этой теме.

Мы напишем функцию с именем `findwords()`, которая определит, какие слова содержатся в текстовом файле, и скомпилирует список позиций вхождений

для каждого слова. Например, такой список может принести пользу при контекстном анализе.

Предположим, входной файл testconcord.txt содержит следующий текст (с фрагментом этой книги):

```
The [1] here means that the first item in this line of output is
item 1. In this case, our output consists of only one line (and one
item), so this is redundant, but this notation helps to read
voluminous output that consists of many items spread over many
lines. For example, if there were two rows of output with six items
per row, the second row would be labeled [7].
```

Для идентификации слов мы заменим все неалфавитные символы пробелами и приведем все символы к единому регистру. Для этого можно воспользоваться строковыми функциями, представленными в главе 11, но для простоты этот код здесь не приводится. Новый файл testconcorda.txt выглядит так:

```
the here means that the first item in this line of output is
item in this case our output consists of only one line and one
item so this is redundant but this notation helps to read
voluminous output that consists of many items spread over many
lines for example if there were two rows of output with six items
per row the second row would be labeled
```

Например, слово `item` встречается в позициях 7, 14 и 27; это означает, что оно является 7-м, 14-м и 27-м словом в файле.

Фрагмент списка, возвращаемого при вызове функции `findwords()` для этого файла:

```
> findwords("testconcorda.txt")
Read 68 items
$the
[1] 1 5 63

$here
[1] 2

$means
[1] 3

$that
[1] 4 40

$first
[1] 6
```

```
$item
[1] 7 14 27
...
```

Список содержит один компонент для каждого слова в файле; компонент обозначает позиции в файле, в которых встречается это слово. Вывод подтверждает, что слово `item` встречается в позициях 7, 14 и 27.

Прежде чем рассматривать код, поговорим о выборе списковой структуры. Одно из возможных решений — использование матрицы, в которой каждому слову текста соответствует строка. Функция `rownames()` используется для назначения имен строк, а элементы строки обозначают позиции слова. Например, строка `item` состоит из чисел 7, 14, 27 и числа 0, которое является признаком конца строки. Тем не менее у решения с матрицей есть несколько серьезных недостатков:

- Возникает проблема с количеством столбцов, выделяемых для хранения матрицы. Если максимальная частота, с которой слово встречается в тексте, равна 10, понадобится 10 столбцов. Но это количество неизвестно заранее. Новый столбец можно добавлять функцией `cbind()` каждый раз, когда обнаруживается новое слово (в дополнение к вызову `rbind()`, добавляющему строку для этого слова). Также можно выполнить предварительную обработку входного файла для определения максимальной частоты. Любое из этих решений обернется возрастанием сложности кода и потенциальным ростом времени выполнения.
- Такая схема хранения данных неэффективно использует память, поскольку большая часть строк будет состоять из множества нулей. Другими словами, матрица будет разреженной — такая ситуация также часто встречается в контексте числового анализа.

Следовательно, для применения списковой структуры есть веские причины. Посмотрим, как запрограммировать это решение.

```
1 findwords <- function(tf) {
2    # Прочитать слова из файла в вектор с режимом character
3    txt <- scan(tf,"")
4    wl <- list()
5    for (i in 1:length(txt)) {
6       wrd <- txt[i] # i-e слово входного файла
7       wl[[wrd]] <- c(wl[[wrd]],i)
8    }
9    return(wl)
10 }
```

Чтение слов из файла («словом» считается любая группа алфавитных символов, ограниченная пробелами) осуществляется вызовом `scan()`. Подробности

чтения и записи файлов рассматриваются в главе 10, но здесь важно то, что `txt` становится вектором строк: по одной строке на каждый экземпляр слова в файле. Вот как выглядит `txt` после чтения:

```
> txt
 [1] "the"         "here"       "means"      "that"       "the"
 [6] "first"       "item"       "in"         "this"       "line"
[11] "of"          "output"     "is"         "item"       "in"
[16] "this"        "case"       "our"        "output"     "consists"
[21] "of"          "only"       "one"        "line"       "and"
[26] "one"         "item"       "so"         "this"       "is"
[31] "redundant"   "but"        "this"       "notation"   "helps"
[36] "to"          "read"       "voluminous" "output"     "that"
[41] "consists"    "of"         "many"       "items"      "spread"
[46] "over"        "many"       "lines"      "for"        "example"
[51] "if"          "there"      "were"       "two"        "rows"
[56] "of"          "output"     "with"       "six"        "items"
[61] "per"         "row"        "the"        "second"     "row"
[66] "would"       "be"         "labeled"
```

Списковые операции в строках с 4-й по 8-ю строят главную переменную — список `wl` (от «word list», то есть «список слов»). Все слова длинной строки перебираются в цикле, при этом `wrd` является текущим словом.

Посмотрим, что происходит с кодом в строке 7 при `i=4`, так как в файле testconcorda.txt из нашего примера `wrd ="that"`. В этот момент `wl[["that"]]` еще не существует. Как упоминалось ранее, по правилам R в этом случае `wl[["that"]] = NULL`, а значит, в строке 7 может использоваться для конкатенации! `wl[["that"]]` становится одноэлементным вектором `(4)`. Позднее, при `i=40` `wl[["that"]]` превращается в `(4,40)`, отражая тот факт, что слова 4 и 40 в файле содержат значение `"that"`.

Обратите внимание, как удобно использовать индексирование по строкам, заключенным в кавычки, — например, `wl[["that"]]`.

В улучшенной, более элегантной версии этого кода используется функция R `split()`; она будет рассмотрена в разделе 6.2.2.

4.3. Обращение к компонентам списков и значениям

Если компонентам списка назначены теги (такие, как `name`, `salary` и `union` для `j` в разделе 4.1), их можно получить вызовом `names()`:

```
> names(j)
[1] "name" "salary" "union"
```

Для получения значений используется функция `unlist()`:

```
> ulj <- unlist(j)
> ulj
   name   salary   union
  "Joe"  "55000"  "TRUE"
> class(ulj)
[1] "character"
```

Возвращаемое значение `unlist()` представляет собой вектор — в данном случае содержащий символьные строки. При этом имена элементов в векторе берутся из компонентов исходного списка.

С другой стороны, если список содержит числовую информацию, вы получите числа:

```
> z <- list(a=5,b=12,c=13)
> y <- unlist(z)
> class(y)
[1] "numeric"
> y
 a  b  c
 5 12 13
```

Таким образом, результат `unlist()` в данном случае представляет собой числовой вектор. А как насчет смешанного случая?

```
> w <- list(a=5,b="xyz")
> wu <- unlist(w)
> class(wu)
[1] "character"
> wu
    a     b
  "5" "xyz"
```

Здесь R выбирает «наименьший общий знаменатель»: символьные строки. Создается впечатление, что существует некоторая система приоритетов, и это действительно так. Как указано в справке по состояниям `unlist()`:

«Компоненты списков приводятся к общему режиму в процессе развертывания списка, и результатом часто оказывается символьный вектор. Векторы приводятся к наивысшему типу компонентов в иерархии NULL<raw<logical<integer <real<complex<character<list<expression: парные списки интерпретируются как списки».

Однако это еще не все. Хотя wu представляет собой вектор, а не список, R присваивает каждому элементу имя. Чтобы удалить имена, можно присвоить им NULL, как было показано в разделе 2.11.

```
> names(wu) <- NULL
> wu
[1] "5"   "xyz"
```

Также имена элементов можно удалить напрямую вызовом unname():

```
> wun <- unname(wu)
> wun
[1] "5" "xyz"
```

К преимуществам этого способа относится то, что он не уничтожает имена в wu на тот случай, если они понадобятся позднее. Если же они не понадобятся, можно просто выполнить присваивание wu вместо wun в предыдущей команде.

4.4. Применение функций к спискам

Две функции lapply и sapply хорошо подходят для применения функций к спискам.

4.4.1. Функции lapply() и sapply()

Функция lapply() (сокращение от «list apply») работает так же, как и матричная функция apply(): она вызывает заданную функцию для каждого компонента списка (или вектора, преобразованного к списку) и возвращает другой список. Пример:

```
> lapply(list(1:3,25:29),median)
[[1]]
[1] 2

[[2]]
[1] 27
```

R применяет median() к 1:3 и 25:29, возвращая список с элементами 2 и 27. В некоторых случаях (как в этом примере) список, возвращаемый lapply(), может быть упрощен до вектора или матрицы. Именно это делает функция sapply() (от «simplified [l]apply», то есть «упрощенная [l]apply»):

```
> sapply(list(1:3,25:29),median)
[1] 2 27
```

Пример матричного вывода был показан в разделе 2.6.2. По этой причине мы применили векторизованную функцию с векторными значениями — функцию c, возвращаемое значение которой является вектором, каждый из компонентов которого векторизован, — к векторным входным данным. Использование `sapply()` (вместо прямого применения функции) дает нужную матричную форму вывода.

4.4.2. Расширенный пример: поиск слов (продолжение)

Функция поиска слов `findwords()`, которая была разработана в разделе 4.2.4, возвращала список позиций слов, индексированный по словам. Было бы удобно иметь возможность сортировать этот список разными способами.

Вспомните, что для входного файла testconcorda.txt был получен следующий вывод:

```
$the
[1] 1 5 63

$here
[1] 2

$means
[1] 3

$that
[1]  4 40
...
```

Следующий код предназначен для вывода списка в алфавитном порядке слов:

```
1 # Сортирует wrdlst – вывод findwords() – по словам в алфавитном порядке
2 alphawl <- function(wrdlst) {
3    nms <- names(wrdlst) # Слова
4    sn <- sort(nms) # Те же слова в алфавитном порядке
5    return(wrdlst[sn]) # Вернуть отсортированную версию
6 }
```

Так как слова являются именами компонентов списка, их можно извлечь простым вызовом `names()`. Данные сортируются по алфавиту, после чего в строке 5 отсортированная версия используется в качестве входных данных для индексирования списка, и мы получаем отсортированную версию списка. Обратите внимание на использование одинарных квадратных скобок вместо двойных, так как они необходимы для сегментации списков. (Также можно подумать

об использовании `order()` вместо `sort()`; эта возможность рассматривается в разделе 8.3.)

Проверим, как работает программа.

```
> alphawl(wl)
$and
[1] 25

$be
[1] 67

$but
[1] 32

$case
[1] 17

$consists
[1] 20 41

$example
[1] 50
...
```

Все отлично работает. Сначала выводятся данные для *and*, затем данные для *be* и т. д.

Также аналогичным способом можно отсортировать данные по частоте слов:

```
1 # Упорядочивает вывод findwords() по частоте слов
2 freqwl <- function(wrdlst) {
3    freqs <- sapply(wrdlst,length) # Получить частоты слов
4    return(wrdlst[order(freqs)])
5 }
```

В строке 3 используется тот факт, что каждый элемент `wrdlst` представляет собой вектор чисел, представляющих позиции входного файла, в которых было обнаружено заданное слово. Вызов `length()` для этого вектора возвращает количество вхождений заданного слова в файле. В результате вызова `sapply()` будет получен вектор частот слов.

Здесь снова можно было использовать `sort()`, но вызов `order()` более прямолинеен. Последняя функция возвращает индексы отсортированного вектора по отношению к исходному вектору. Пример:

```
> x <- c(12,5,13,8)
> order(x)
[1] 2 4 1 3
```

Вывод показывает, что `x[2]` — наименьший элемент `x`, `x[4]` — второй элемент от конца и т. д. В нашем случае функция `order()` используется для определения того, какое слово встречается реже всех остальных, находится на втором месте от конца и т. д. Объединяя эти индексы со списком слов, мы получаем тот же список, но в порядке частоты.

Проверим, как работает код.

```
> freqwl(wl)
$here
[1] 2

$means
[1] 3

$first
[1] 6
...
$that
[1] 4 40

$`in`
[1] 8 15

$line
[1] 10 24
...
$this
[1] 9162933

$of
[1] 11 21 42 56

$output
[1] 12 19 39 57
```

Да, данные упорядочены по возрастанию частот.

Также можно построить диаграмму наиболее часто встречающихся слов. Я применил следующий код к статье о R, опубликованной в New York Times, — «Data Analysts Captivated by R's Power» (6 января 2009 года).

```
> nyt <- findwords("nyt.txt")
Read 1011 items
> snyt <- freqwl(nyt)
> nwords <- length(ssnyt)
> freqs9 <- sapply(ssnyt[round(0.9*nwords):nwords],length)
> barplot(freqs9)
```

Я хотел вывести частоты для верхних 10 % слов в статье. Результат показан на рис. 4.1.

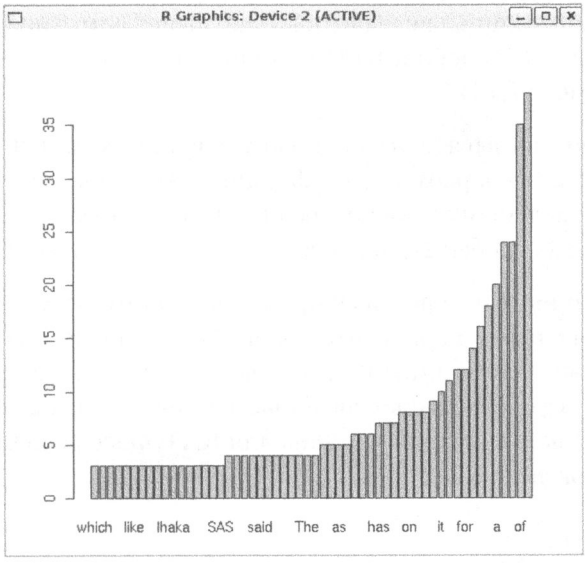


Рис. 4.1. Диаграмма частот наиболее часто встречающихся слов в статье, посвященной R

4.4.3. Расширенный пример: возвращение к данным абалонов

Применим функцию `lapply()` в примере с полом моллюсков-абалонов из раздела 2.9.2. Вспомните, что в определенный момент работы над примером нам потребовалось узнать индексы наблюдений для мужских, женских и молодых особей. Для простой демонстрации воспользуемся тем же тестовым сценарием: вектором полов.

```
g <- c("M","F","F","I","M","M","F")
```

Более компактное решение выглядит так:

```
> lapply(c("M","F","I"),function(gender) which(g==gender))
[[1]]
[1] 1 5 6

[[2]]
[1] 2 3 7

[[3]]
[1] 4
```

Функция `lapply()` ожидает, что ее первый аргумент будет списком. В данном случае он был вектором, но `lapply()` преобразует вектор в списковую форму. Кроме того, `lapply()` ожидает, что второй аргумент является функцией. Это может быть имя функции, как было показано выше, или фактический код, как в данном случае (здесь используется анонимная функция, о них вы узнаете подробнее в разделе 7.13).

Функция `lapply()` вызывает эту анонимную функцию для `"M"`, затем для `"F"` и, наконец, для `"I"`. В первом случае функция вычисляет `which(g=="M")` и получает вектор индексов мужских особей в `g`. После определения индексов для женских и молодых особей `lapply()` возвращает три вектора в виде списка.

Обратите внимание: хотя нас здесь прежде всего интересует вектор полов `g`, он не передается в первом аргументе вызова `lapply()` в примере. Вместо этого аргумент содержит безобидный на первый взгляд вектор трех возможных кодировок пола, а `g` очень кратко упоминается в функции как второй фактический аргумент. Такая ситуация типична для R. Лучшее решение задачи будет представлено в разделе 6.2.2.

4.5. Рекурсивные списки

Списки могут быть *рекурсивными*; это означает, что они могут содержать другие списки. Пример:

```
> b <- list(u = 5,v=12)
> c <- list(w = 13)
> a <- list(b,c)
> a
[[1]]
[[1]]$u
[1] 5

[[1]]$v
[1] 12

[[2]]
[[2]]$w
[1] 13


> length(a)
[1] 2
```

Этот код преобразует a в список из двух компонентов, и каждый компонент также является списком.

Функция конкатенации c() получает необязательный аргумент recursive, который управляет тем, происходит ли сглаживание при объединении рекурсивных списков.

```
> c(list(a=1,b=2,c=list(d=5,e=9)))
$a
[1] 1

$b
[1] 2

$c
$c$d
[1] 5

$c$e
[1] 9
> c(list(a=1,b=2,c=list(d=5,e=9)),recursive=T)
  a   b c.d c.e
  1   2   5   9
```

В первом случае мы принимаем значение по умолчанию recursive, равное FALSE, и получаем рекурсивный список; компонент c главного списка сам является другим списком. Во втором вызове recursive присваивается значение TRUE, и в результате мы получаем только один список (хотя немного странно, что при присваивании recursive значения TRUE возвращается *нерекурсивный* список).

Вспомните базу данных работников из первого примера. Я говорил, что поскольку каждый работник представлен списком, вся база данных представляет собой список списков. Это конкретный пример использования рекурсивных списков.

5 Кадры данных

На интуитивном уровне *кадр данных* (data frame) похож на матрицу: он тоже имеет двумерную структуру из строк и столбцов. С другой стороны, кадр данных отличается от матриц тем, что все столбцы могут иметь разные режимы. Скажем, один столбец может содержать числа, а другой — символьные строки. Если списки являются разнородными аналогами векторов в одном измерении, кадры данных являются разнородными аналогами матриц для двумерных данных.

На техническом уровне кадр данных представляет собой список, компоненты которого являются векторами равной длины. На самом деле R допускает, чтобы компоненты были объектами других типов, включая другие кадры данных. Отсюда появляются аналоги массивов с разнородными данными в нашей аналогии. Тем не менее такое использование кадров данных на практике встречается редко, и в этой книге мы будем считать, что все компоненты кадра данных являются векторами.

В этой главе мы рассмотрим достаточно много примеров кадров данных, чтобы вы получили представление о разнообразии их применения в R.

5.1. Создание кадров данных

Для начала еще раз рассмотрим пример простого кадра данных из раздела 1.4.5:

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d # структура, сходная с матрицей
  kids ages
1 Jack   12
2 Jill   10
```

Первые два аргумента вызова `data.frame()` понятны: мы хотим построить кадр данных из двух векторов, `kids` и `ages`. Тем не менее третий аргумент, `stringsAsFactors=FALSE`, требует более подробных пояснений.

Если именованный аргумент `stringsAsFactors` не указан, то по умолчанию используется значение `TRUE`. (Также можно использовать `options()` для выбора противоположного значения по умолчанию.) Это означает, что при создании кадра данных на основе символьного вектора — в данном случае `kids` — R преобразует этот вектор в *фактор*. Так как наша работа с символьными данными в основном будет происходить с использованием векторов, а не факторов, мы присвоим `stringsAsFactors` значение `FALSE`. Факторы рассматриваются в главе 6.

5.1.1. Обращение к кадрам данных

Кадр данных создан, теперь можно немного поэкспериментировать. Так как `d` является списком, к нему можно обращаться как по индексам компонентов, так и по именам компонентов:

```
> d[[1]]
[1] "Jack" "Jill"
> d$kids
[1] "Jack" "Jill"
```

Но с ним также можно работать по тем же правилам, что и с матрицами. Например, можно просмотреть столбец 1:

```
> d[,1]
[1] "Jack" "Jill"
```

Это сходство с матрицами также проявляется при разборе `d` с использованием `str()`:

```
> str(d)
'data.frame': 2 obs. of 2 variables:
$ kids: chr "Jack" "Jill"
$ ages: num 12 10
```

R сообщает, что `d` состоит из двух наблюдений (две строки), в которых хранятся данные двух переменных (два столбца).

Рассмотрим три способа обращения к первому столбцу приведенного выше кадра данных: `d[[1]]`, `d[,1]` и `d$kids`. Третий способ обычно считается более

понятным и, что еще важнее, — более безопасным, чем первые два. Он лучше идентифицирует столбец и снижает вероятность случайного обращения к другому столбцу. Но при написании обобщенного кода — допустим, написании пакета R — необходима матричная запись `d[,1]`, что особенно удобно при извлечении подкадров данных (вы убедитесь в этом, когда мы будем рассматривать подкадры данных в разделе 5.2).

5.1.2. Расширенный пример: регрессионный анализ экзаменационных оценок, продолжение

Вспомните набор данных оценок в разделе 1.5. Заголовок в этом наборе отсутствует, но для этого примера мы его создадим, и несколько первых записей в файле теперь выглядят так:

```
"Exam 1" "Exam 2" Quiz
2.0      3.3      4.0
3.3      2.0      3.7
4.0      4.0      4.0
2.3      0.0      3.3
2.3      1.0      3.3
3.3      3.7      4.0
```

Как видите, каждая строка содержит три оценки для одного студента. Это классическое представление двумерного файла, как в предыдущем примере `str()`. Здесь каждая строка файла содержит данные одного наблюдения в статистическом наборе данных. Концепция кадра данных заключается в том, чтобы инкапсулировать такие данные (вместе с именами переменных) в одном объекте.

Обратите внимание на то, что поля в данном случае разделяются пробелами. Также можно выбрать другие ограничители, прежде всего запятые для файлов в формате CSV (Comma-Separated Values, то есть «значения, разделенные запятыми»), — об этом будет рассказано в разделе 5.2.5. Имена переменных, заданные в первой записи, должны разделяться тем же символом, который используется для данных, — в данном случае это пробел. Если сами имена содержат внутренние пробелы, как в нашем примере, они должны быть заключены в кавычки.

Файл читается так же, но на этот раз указывает, что в нем присутствует строка заголовков:

```
examsquiz <- read.table("exams",header=TRUE)
```

Теперь в выводе появляются имена столбцов, а пробелы заменяются точками:

```
> head(examsquiz)
  Exam.1 Exam.2 Quiz
1    2.0    3.3  4.0
2    3.3    2.0  3.7
3    4.0    4.0  4.0
4    2.3    0.0  3.3
5    2.3    1.0  3.3
6    3.3    3.7  4.0
```

5.2. Другие матричные операции

Различные матричные операции также применимы к кадрам данных. Самая главная и полезная операция такого рода — фильтрация для извлечения различных подкадров данных, представляющих интерес для аналитика.

5.2.1. Извлечение подкадров данных

Как упоминалось ранее, кадр данных может рассматриваться в контексте строк и столбцов. В частности, из него можно извлекать подкадры данных по строкам и столбцам. Пример:

```
> examsquiz[2:5,]
  Exam.1 Exam.2 Quiz
2    3.3      2  3.7
3    4.0      4  4.0
4    2.3      0  3.3
5    2.3      1  3.3
> examsquiz[2:5,2]
[1]2401
> class(examsquiz[2:5,2])
[1] "numeric"
> examsquiz[2:5,2,drop=FALSE]
  Exam.2
2      2
3      4
4      0
5      1
> class(examsquiz[2:5,2,drop=FALSE])
[1] "data.frame"
```

Во втором вызове, поскольку `examsquiz[2:5,2]` является вектором, R создает вектор вместо другого кадра данных. Передавая аргумент `drop=FALSE`, как опи-

сано в примере для матрицы в разделе 3.6, можно сохранить результат в виде (одностолбцового) кадра данных.

Также можно выполнить фильтрацию данных. В следующем примере извлекается подкадр данных всех студентов, у которых первая оценка не ниже 3,8:

```
> examsquiz[examsquiz$Exam.1 >= 3.8,]
   Exam.1 Exam.2 Quiz
 3      4    4.0  4.0
 9      4    3.3  4.0
11      4    4.0  4.0
14      4    0.0  4.0
16      4    3.7  4.0
19      4    4.0  4.0
22      4    4.0  4.0
25      4    4.0  3.3
29      4    3.0  3.7
```

5.2.2. Об интерпретации значений NA

Предположим, вторая оценка для первого студента отсутствует в данных. Тогда при подготовке файла данных в эту строку следует включить следующее значение:

```
2.0 NA 4.0
```

В любом последующем статистическом анализе R старается как можно лучше справиться с отсутствующими данными. Однако в некоторых ситуациях необходимо передать аргумент `na.rm=TRUE`, явно приказывая R игнорировать значения `NA`. Например, если оценка отсутствует, то при вычислении средней оценки за экзамен 2 вызовом функции R `mean()` первый студент будет пропущен при вычислении. В противном случае R просто выдаст `NA` вместо средней оценки.

Небольшой пример:

```
> x <- c(2,NA,4)
> mean(x)
[1] NA
> mean(x,na.rm=TRUE)
[1] 3
```

В разделе 2.8.2 была представлена функция `subset()`, которая избавляет от хлопот с передачей `na.rm=TRUE`. Эта функция может применяться к кадрам данных

для выбора строк. Имена столбцов рассматриваются в контексте конкретного кадра данных. В нашем примере вместо команды:

```
> examsquiz[examsquiz$Exam.1 >= 3.8,]
```

можно выполнить следующую команду:

```
> subset(examsquiz,Exam.1 >= 3.8)
```

Обратите внимание: не нужно использовать следующую запись:

```
> subset(examsquiz,examsquiz$Exam.1 >= 3.8)
```

В некоторых ситуациях можно решить, что из кадра данных следует исключить все наблюдения, имеющие хотя бы одно значение NA. Для этого существует удобная функция complete.cases().

```
> d4
     kids states
1    Jack     CA
2    <NA>     MA
3 Jillian     MA
4    John   <NA>
> complete.cases(d4)
[1] TRUE FALSE TRUE FALSE
> d5 <- d4[complete.cases(d4),]
> d5
     kids states
1    Jack     CA
3 Jillian     MA
```

Данные наблюдений 2 и 4 были неполными; отсюда значения FALSE в выводе complete.cases(d4). Затем этот вывод используется для выбора строк с полными данными.

5.2.3. Использование rbind() и cbind() и альтернативных функций

Матричные функции rbind() и cbind(), представленные в разделе 3.4, будут работать и с кадрами данных — конечно, при условии совместимости размеров. Например, при помощи cbind() можно добавить новый столбец с такой же длиной, как у существующих столбцов.

При добавлении строки вызовом rbind() строка обычно добавляется в форме другого кадра данных или списка.

```
> d
  kids ages
1 Jack   12
2 Jill   10
> rbind(d,list("Laura",19))
  kids ages
1 Jack   12
2 Jill   10
3 Laura  19
```

Также можно создавать новые столбцы на основе уже существующих. Например, можно добавить переменную для разности между оценками за экзамены 1 и 2:

```
> eq <- cbind(examsquiz,examsquiz$Exam.2-examsquiz$Exam.1)
> class(eq)
[1] "data.frame"
> head(eq)
  Exam.1 Exam.2 Quiz examsquiz$Exam.2 - examsquiz$Exam.1
1 2.0 3.3 4.0 1.3
2 3.3 2.0 3.7 -1.3
3 4.0 4.0 4.0 0.0
4 2.3 0.0 3.3 -2.3
5 2.3 1.0 3.3 -1.3
6 3.3 3.7 4.0 0.4
```

Новое имя получилось неудобным: оно длинное и содержит внутренние пробелы. Его можно изменить функцией `names()`, но лучше воспользоваться списковой природой кадров данных и добавить в кадр данных столбец (той же длины) для этого результата:

```
> examsquiz$ExamDiff <- examsquiz$Exam.2 - examsquiz$Exam.1
> head(examsquiz)
  Exam.1 Exam.2 Quiz ExamDiff
1    2.0    3.3  4.0      1.3
2    3.3    2.0  3.7     -1.3
3    4.0    4.0  4.0      0.0
4    2.3    0.0  3.3     -2.3
5    2.3    1.0  3.3     -1.3
6    3.3    3.7  4.0      0.4
```

Что здесь происходит? Так как в уже существующий список в любой момент можно добавить новый компонент, мы так и поступаем: компонент `ExamDiff` добавляется в список/кадр данных `examsquiz`.

Вы даже можете воспользоваться механизмом переработки и добавить столбец, который отличается по длине от столбцов в кадре данных:

```
> d
  kids ages
1 Jack   12
2 Jill   10
> d$one <- 1
> d
  kids ages one
1 Jack   12   1
2 Jill   10   1
```

5.2.4. Применение apply()

Функция `apply()` может применяться к кадрам данных, если все столбцы относятся к одному типу. Например, можно определить максимальную оценку для каждого студента:

```
> apply(examsquiz,1,max)
[1] 4.0 3.7 4.0 3.3 3.3 4.0 3.7 3.3 4.0 4.0 4.0 3.3 4.0 4.0 3.7 4.0 3.3 3.7 4.0
[20] 3.7 4.0 4.0 3.3 3.3 4.0 4.0 3.3 3.3 4.0 3.7 3.3 3.3 3.7 2.7 3.3 4.0 3.7 3.7
[39] 3.7
```

5.2.5. Расширенный пример: анализ зарплаты

В ходе одного исследования инженеров и программистов я искал ответ на вопрос: «Какие из этих работников являются самыми лучшими и одаренными, то есть людьми выдающихся способностей?» (Некоторые подробности в данных были изменены.)

Правительственные данные, которыми я располагал, были ограниченны. Один (честно говоря, не идеальный) способ выявления выдающихся способностей основан на анализе отношения фактической зарплаты к типичной зарплате для этой должности и географического места. Если отношение заметно выше 1,0, можно не без основания считать, что работник действительно талантлив.

Я использовал R для подготовки и анализа данных, а здесь приведу выдержки из своего подготовительного кода. Сначала я прочитал файл данных:

```
all2006 <- read.csv("2006.csv",header=TRUE,as.is=TRUE)
```

Функция `read.csv()` фактически идентична `read.table()`, если не считать того, что входные данные хранятся в формате CSV, экспортируемом электронными таблицами (в этом формате набор данных был подготовлен Министерством труда США). Аргумент `as.is` инвертирует `stringsAsFactors`, как было показано

в разделе 5.1. Таким образом, присваивание ему TRUE просто является альтернативным способом stringsAsFactors=FALSE.

На этот момент у меня был кадр данных all2006, состоящий из всех данных за 2006 год. Затем я выполнил фильтрацию:

```
all2006 <- all2006[all2006$Wage_Per=="Year",] # Исключить временных работников
all2006 <- all2006[all2006$Wage_Offered_From > 20000,] # Исключить аномалии
all2006 <- all2006[all2006$Prevailing_Wage_Amount > 200,] # Исключить
                                                  # почасовую типичную зарплату
```

Здесь выполняется типичная чистка данных. В большинстве крупных наборов данных присутствуют аномальные значения — очевидные ошибки, значения в других единицах измерения и т. д. Я должен был исправить ситуацию перед тем, как приступать к анализу.

Также нужно было создать новый столбец для отношения фактической зарплаты к типичной:

```
all2006$rat <- all2006$Wage_Offered_From / all2006$Prevailing_Wage_Amount
```

Так как я знал, что мне придется вычислять медиану по новому столбцу для многих поднаборов данных, я определил для этого функцию:

```
medrat <- function(dataframe) {
   return(median(dataframe$rat,na.rm=TRUE))
}
```

Обратите внимание на исключение значений NA, часто встречающихся в правительственных наборах данных.

Меня особенно интересовали три профессии, поэтому я выделил их подкадры данных для удобства анализа:

```
se2006 <- all2006[grep("Software Engineer",all2006),]
prg2006 <- all2006[grep("Programmer",all2006),]
ee2006 <- all2006[grep("Electronics Engineer",all2006),]
```

Здесь я воспользовался функцией R grep() для нахождения строк с названием должности. Более подробное описание функции приведено в главе 11.

Также представлял интерес анализ данных в пределах компаний. Я написал следующую функцию для извлечения подкадра данных для заданной компании:

```
makecorp <- function(corpname) {
   t <- all2006[all2006$Employer_Name == corpname,]
   return(t)
}
```

Затем были созданы подкадры данных для нескольких компаний (ниже приведены лишь отдельные примеры):

```
corplist <- c("MICROSOFT CORPORATION","ms","INTEL CORPORATION","intel","
SUN MICROSYSTEMS, INC.","sun","GOOGLE INC.","google")

for (i in 1:(length(corplist)/2)) {
   corp <- corplist[2*i-1]
   newdtf <- paste(corplist[2*i],"2006",sep="")
   assign(newdtf,makecorp(corp),pos=.GlobalEnv)
}
```

В этом коде есть несколько мест, заслуживающих внимания. Прежде всего обратите внимание на то, что все переменные создаются на верхнем (то есть глобальном) уровне, — это типично для интерактивного анализа. Кроме того, имена новых переменных создаются на основе символьных строк — например, `"intel2006"`. По этим причинам функция `assign()` отлично подойдет для указанной цели. Это позволило мне присвоить значение переменной по имени в виде строки и выбрать верхний уровень (см. раздел 7.8.2).

Функция `paste()` позволяет выполнять конкатенацию строк, при этом аргумент `sep=""` сообщает, что составляющие строки не должны разделяться дополнительными символами в результате конкатенации.

5.3. Слияние кадров данных

В мире реляционных баз данных одной из важнейших операций является операция *соединения* (join), при которой происходит слияние двух таблиц по значениям общей переменной. В R два кадра данных объединяются аналогичным образом при помощи функции `merge()`.

Простейшая форма этой функции выглядит так:

```
merge(x,y)
```

Вызов объединяет кадры данных `x` и `y`. Предполагается, что два кадра данных содержат один или несколько столбцов с одинаковыми именами. Пример:

```
> d1
     kids states
1    Jack     CA
2    Jill     MA
3 Jillian     MA
4    John     HI
```

```
> d2
  ages    kids
1   10    Jill
2    7 Lillian
3   12    Jack
> d <- merge(d1,d2)
> d
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

Здесь два кадра данных содержат общую переменную `kids`. R находит строки, у которых эта переменная имеет одинаковое значение в двух кадрах данных (`Jack` и `Jill`). Затем создается кадр данных с соответствующими строками и столбцами из обоих кадров данных (`kids`, `states` и `ages`).

Функция `merge()` содержит именованные аргументы `by.x` и `by.y` для ситуаций, в которых переменные содержат похожую информацию, но имеют разные имена в кадрах данных. Пример:

```
> d3
  ages    pals
1   12    Jack
2   10    Jill
3    7 Lillian
> merge(d1,d3,by.x="kids",by.y="pals")
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
```

Хотя переменная называлась `kids` в одном кадре данных и `pals` в другом, эти столбцы предназначены для хранения одной и той же информации, поэтому слияние имеет смысл.

Дубликатные совпадения будут полностью присутствовать в результате — вероятно, в нежелательной форме.

```
> d1
     kids states
1    Jack     CA
2    Jill     MA
3    Jillian  MA
4    John     HI
> d2a <- rbind(d2,list(15,"Jill"))
> d2a
  ages    kids
1   12    Jack
```

```
2   10    Jill
3    7 Lillian
4   15    Jill
> merge(d1,d2a)
  kids states ages
1 Jack     CA   12
2 Jill     MA   10
3 Jill     MA   15
```

В d2a присутствуют две записи Jill. Одна запись из d1 относится к штату Массачусетс (MA), а у другой штат не определен. В предыдущем примере merge(d1,d2) запись Jill была только одна, и предполагалось, что она соответствует одному человеку в обоих кадрах данных. Но при вызове merge(d1,d2a) может оказаться, что в Массачусетсе живет только одна из двух Jill. Этот пример наглядно показывает, что переменные для соединения следует выбирать с большой осторожностью.

5.3.1. Расширенный пример: база данных работников

Следующий пример представляет собой видоизмененный вариант одного из моих консультационных проектов. В этом проекте требовалось сравнить эффективность труда пожилых и молодых работников. Я располагал данными по нескольким показателям, таким как возраст и оценка производительности; они использовались в моем сравнении. В данных также содержались идентификаторы работников, необходимые для соединения двух файлов данных: DA и DB.

Файл DA содержал следующий заголовок:

```
"EmpID","Perf 1","Perf 2","Perf 3","Job Title"
```

Это имена для идентификатора сотрудника, три оценки производительности и название должности. В DB заголовка нет. Имена переменных снова начинаются с идентификатора, за которым идут даты приема и увольнения.

Оба файла содержат данные в формате CSV. Один из этапов чистки данных заключался в проверке того, что каждая запись содержит правильное количество полей. Например, в DA каждая запись должна была состоять из пяти полей. Проверка выглядит так:

```
> count.fields("DA",sep=",")
 [1] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
 5 5 5 5
...
```

Здесь я указываю, что файл DA содержит поля, разделенные запятыми. Затем функция выводит количества полей во всех записях файла; к счастью, все они были равны 5.

Также можно было воспользоваться функцией all() вместо визуальной проверки:

```
all(count.fields("DA",sep=",") >= 5)
```

Возвращаемое значение TRUE будет означать, что все прошло нормально. Также можно было использовать следующую форму:

```
table(count.fields("DA",sep=","))
```

Вызов вернет количество записей с пятью полями, четырьмя полями, шестью полями и т. д.

После этой проверки я читаю файлы как кадры данных:

```
da <- read.csv("DA",header=TRUE,stringsAsFactors=FALSE)
db <- read.csv("DB",header=FALSE,stringsAsFactors=FALSE)
```

Я также хотел проверить все возможные ошибки в разных полях, поэтому я выполнил следующий код:

```
for (col in 1:6)
   print(unique(sort(da[,col])))
```

Так был получен список уникальных значений в каждом столбце, чтобы я мог визуально обнаружить возможные ошибки.

Два кадра данных должны были объединяться по идентификатору работника, поэтому я выполнил следующий код:

```
mrg <- merge(da,db,by.x=1,by.y=1)
```

Я указал, что первый столбец в обоих случаях будет переменной слияния. (Как упоминалось ранее, вместо чисел также можно было использовать имена полей.)

5.4. Применение функций к кадрам данных

Как и в случае со списками, функции lapply и sapply также могут использоваться с кадрами данных.

5.4.1. Функции lapply() и sapply() с кадрами данных

Следует помнить, что кадры данных являются особой разновидностью списков; компонентами списков являются столбцы кадра данных. Следовательно, если вызвать `lapply()` для кадра данных с функцией `f()`, функция `f()` будет вызвана для каждого столбца кадра, а возвращаемые значения будут помещены в список.

Например, для нашего предыдущего примера функция `lapply` может использоваться следующим образом:

```
> d
  kids ages
1 Jack   12
2 Jill   10
> dl <- lapply(d,sort)
> dl
$kids
[1] "Jack" "Jill"

$ages
[1] 10 12
```

Итак, `dl` представляет собой список, состоящий из двух векторов: отсортированных версий `kids` и `ages`.

Следует помнить, что `dl` — всего лишь список, а не кадр данных. Его можно преобразовать в кадр данных:

```
as.data.frame(dl)
  kids ages
1 Jack   10
2 Jill   12
```

Но это не имеет смысла, так как соответствия между именами и возрастом были потеряны. Например, для `Jack` указан возраст 10 лет вместо 12. (Но если бы мы захотели отсортировать кадр данных по одному из столбцов с сохранением соответствий, можно было бы воспользоваться методом, представленным на с. 171.)

5.4.2. Расширенный пример: применение моделей логистической регрессии

Применим модель логистической регрессии к данным абалонов из раздела 2.9.2 и попробуем спрогнозировать пол по каждой из остальных восьми переменных: высоте, весу, количеству колец и т. д.

Логистическая модель используется для прогнозирования случайной переменной Y со значениями 0/1 по одной или нескольким независимым переменным. Значение функции определяет вероятность того, что Y=1 при заданных независимых переменных. Скажем, имеется всего одна такая переменная X. Модель выглядит так:

$$Pr(Y=1|X=t)=\frac{1}{1+\exp[-(\beta_0+\beta_1 t)]}.$$

Как и для линейных регрессионных моделей, значения β_i вычисляются на основании данных, с использованием функции `glm()` с аргументом `family=binomial`.

Мы можем использовать `sapply()` для подгонки восьми моделей с одной независимой переменной — по одной для каждой из восьми переменных, кроме пола, в наборе данных — всего в одной строке кода:

```
1 aba <- read.csv("abalone.data",header=T)
2 abamf <- aba[aba$Gender != "I",] # Исключить молодых особей из анализа
3 lftn <- function(clmn) {
4    glm(abamf$Gender ~ clmn, family=binomial)$coef
5 }
6 loall <- sapply(abamf[,-1],lftn)
```

В строках 1 и 2 мы читаем кадр данных, а затем исключаем наблюдения для молодых особей. В строке 6 функция `sapply()` вызывается для подкадра данных, из которого был исключен столбец 1 (с именем `Gender`). Другими словами, это подкадр из восьми столбцов, состоящих из восьми независимых переменных. Таким образом, `lftn()` вызывается для каждого столбца этого подкадра.

Получая в качестве входных данных столбец из подкадра, для обращения к которому используется формальный аргумент `clmn`, строка 4 выполняет подгонку логистической модели, прогнозирующей пол по этому столбцу, а следовательно, по этой независимой переменной. Вспомните, о чем говорилось в разделе 1.5: обычная функция регрессии `lm()` возвращает объект класса `"lm"`, содержащий много компонентов, одним из которых является `$coefficients` — вектор вычисленных значений β_i. Этот компонент также присутствует в возвращаемом значении `glm()`. Также вспомните, что имена компонентов списков можно сокращать так, чтобы при этом не возникло неоднозначности. Здесь мы сократили `coefficients` до `coef`.

В конечном итоге строка 6 вернула восемь пар вычисленных значений β_i. Проверим, как они работают:

```
> loall
             Length    Diameter  Height    WholeWt    ShuckedWt  ViscWt
(Intercept)  1.275832  1.289130  1.027872  0.4300827  0.2855054  0.4829153
clmn        -1.962613 -2.533227 -5.643495 -0.2688070 -0.2941351 -1.4647507
             ShellWt    Rings
(Intercept)  0.5103942  0.64823569
col         -1.2135496 -0.04509376
```

Естественно, мы получаем матрицу 2 × 8, в которой `j`-й столбец содержит пару вычисленных значений β_j, полученных при применении логистической регрессии с `j`-й независимой переменной.

Того же результата можно было добиться с обычной функцией `apply()`, применяемой к матрицам/кадрам данных, но когда я попытался это сделать, обнаружилось, что этот метод работает медленнее. Это расхождение могло объясняться затратами времени на выделение памяти. Обратите внимание на класс возвращаемого значения `glm()`:

```
> class(loall)
[1] "glm" "lm"
```

Как видите, `loall` в действительности имеет два класса: `"glm"` и `"lm"`. Дело в том, что класс `"glm"` является подклассом `"lm"`. Классы более подробно рассматриваются в главе 9.

5.4.3. Расширенный пример: изучение китайских диалектов

Стандартный китайский язык, часто называемый «мандаринским наречием» за пределами Китая, официально называется «путунхуа» или «гоюй». Сегодня на нем говорит подавляющее большинство жителей Китая и многих этнических китайцев по всему миру, но его диалекты, такие как кантонский или шанхайский, также получили широкое распространение. Таким образом, если китайский бизнесмен из Пекина захочет вести бизнес в Гонконге, ему стоит заняться изучением кантонского диалекта. Аналогичным образом многим жителям Гонконга хотелось бы улучшить свое знание мандаринского наречия. Посмотрим, как можно было бы ускорить процесс обучения и какую помощь в этом может оказать R.

Различия между диалектами иногда оказываются весьма значительными. Иероглиф 下 (обозначающий «низ») читается «xia» на мандаринском наречии, «ha» на кантонском диалекте и «wu» на шанхайском. Из-за таких различий, а также

различий в грамматике многие лингвисты считают их разными языками, а не диалектами. Мы будем использовать китайский термин *фаньянь* («региональный язык»).

Посмотрим, как R может помочь людям, говорящим на одном фаньяне, в изучении другого. Здесь ключевой момент заключается в том, что между фаньянями часто существуют закономерности и соответствия. Например, достаточно часто встречается преобразование начального согласного «x» в «h», как в предыдущем примере («xia» → «ha»); оно также встречается в таких иероглифах, как 香 («душистый»), который произносится «xiang» на мандаринском наречии и «heung» на кантонском диалекте. Также обратите внимание на преобразование «iang» → «eung» в неначальных частях этих произношений, они тоже типичны. Знание таких преобразований поможет ускорить изучение, особенно кантонского диалекта, людьми, владеющими мандаринским наречием; мы будем рассматривать именно такую ситуацию.

И мы еще не сказали ни слова о тональности. Все фаньяни являются тональными, и иногда в этой области также проявляются закономерности, которые предоставляют дополнительные средства обучения. Тем не менее мы не пойдем по этому пути. Вы увидите, что наш код должен в некоторой степени использовать тональность, но мы не будем пытаться анализировать преобразования тонов с одного фаньяня на другой. Для простоты мы также не будем рассматривать иероглифы, начинающиеся с гласных; иероглифы, имеющие несколько чтений; бестоновые иероглифы и другие нюансы. Хотя начальный согласный «c» в мандаринском наречии часто преобразуется в «h», как показано ранее, он также часто преобразуется в другие гласные. Например, иероглиф «xie» 謝 в известном мандаринском выражении «xiexie» («спасибо») на кантонском диалекте произносится «je». Здесь происходит замена «x → j» согласного звука.

Список преобразований и их частот был бы очень полезен для человека, изучающего диалект. И эта задача словно создана для R! Функция `mapsound()`, приведенная далее в этой главе, решает эту задачу. В ней используются некоторые вспомогательные функции, которые также будут представлены ниже.

Чтобы объяснить, что делает функция `mapsound()`, сначала выработаем некую терминологию на приведенном выше примере ($x \rightarrow h$). Будем называть x «исходным значением», а h, s и т. д. — «отображенными значениями».

Несколько формальных параметров:

- `df`: кадр данных, состоящий из данных произношения двух фаньяней.
- `fromcol` и `tocol`: имена исходного и отображенного столбца в `df`.

- `sourceval`: исходное отображаемое значение (такое, как `x` в предыдущем примере).

Начало типичного кадра данных для двух фаньяней, `canman8`, который будет использоваться для `df`:

```
> head(canman8)
  Ch char    Can   Man Can cons Can sound Can tone Man cons Man sound Man tone
1       ·   yat1    yi1       y        at        1        y         i 1
2      丁  ding1  ding1       d       ing        1        d       ing 1
3      七  chat1    qi1      ch        at        1        q         i 1
4      丈 jeung6 zhang4       j      eung        6       zh       ang 4
5      上 seung5 shang4       s      eung        5       sh       ang 3
6      下    ha5   xia4       h         a        5        x        ia 4
```

Функция возвращает список с двумя компонентами:

- `counts`: вектор целых чисел, индексируемый по отображаемым значениям и содержащий счетчики этих значений. Имена элементов вектора назначаются в соответствии с отображаемыми значениями.
- `images`: список символьных векторов. И снова индексами списка являются отображаемые значения, а каждый вектор содержит все иероглифы, соответствующие заданному отображаемому значению.

Чтобы описание стало более конкретным, рассмотрим практический пример:

```
> m2cx <- mapsound(canman8,"Man cons","Can cons","x")
> m2cx$counts
ch  f  g  h  j  k  k  w  n  s  y
15  2  1 87  1  2  4  2  1 81 21
```

Как видите, `x` отображается на `ch` 15 раз, на `f` — 2 раза и т. д. Обратите внимание: мы могли вызвать `sort()` для `m2cx$counts`, чтобы просмотреть отображения по порядку с убыванием частоты.

Если вы захотите узнать кантонское произношение слова, которое в латинской записи на мандаринском наречии начинается с буквы `x`, кантонская запись почти наверняка будет начинаться с `h` или `s`. Подобные маленькие подсказки основательно упрощают процесс изучения.

Чтобы попытаться выявить новые закономерности, читатель может попытаться определить, какие символы `x` отображаются, например, на `ch`. Из результата предыдущего примера видно, что таких символов всего шесть. Что это за символы?

Эта информация хранится в images. Последняя переменная, как упоминалось ранее, представляет собой список векторов. Нас интересует вектор, соответствующий ch.

```
> head(m2cx$images[["ch"]])
  Ch char    Can  Man Can cons Can sound Can tone Man cons Man sound Man tone
613    嗅  chau3 xiu4       ch        au        3         x       iu 4
982    尋  cham4 xin2       ch        am        4         x       in 2
1050   巡  chun3 xun2       ch        un        3         x       un 2
1173   徐  chui4  xu2       ch        ui        4         x       u  2
1184   循  chun3 xun2       ch        un        3         x       un 2
1566   斜   che4 xie2       ch         e        4         x       ie 2
```

А теперь рассмотрим код. Прежде чем просматривать код mapsound(), рассмотрим другую необходимую вспомогательную функцию. Предполагается, что кадр данных df, который подается на ввод mapsound(), создается слиянием двух кадров для разных фаньяней. В этом случае, например, начало входного кадра для кантонского диалекта выглядит так:

```
> head(can8)
  Ch char   Can
1       一 yat1
2       乙 yuet3
3       丁 ding1
4       七 chat1
5       乃 naai5
6       九  gau2
```

Кадр данных для мандаринского наречия выглядит аналогично. Необходимо объединить эти два кадра в приведенный выше кадр canman8. Я написал код так, чтобы эта операция не только объединяла кадры, но и разделяла транслитерацию на начальный согласный, остаток и номер тона. Например, ding1 разделяется на d, ing и 1.

Аналогичным образом можно исследовать преобразования в другом направлении, из кантонского в мандаринское наречие, основанное на гласных остатках. Например, можно определить, какие иероглифы содержат eung в гласной части кантонского произношения:

```
> c2meung <- mapsound(canman8,c("Can cons","Man cons"),"eung")
```

Затем можно проанализировать ассоциированные звуки мандаринского наречия.

Код для решения этой задачи:

```
# Объединяет кадры данных для 2 фаньяней
merge2fy <- function(fy1,fy2) {
   outdf <- merge(fy1,fy2)
   # Отделить тон от звука и создать новые столбцы
   for (fy in list(fy1,fy2)) {
      # saplout будет матрицей: начальный согласный в строке 1, остатки
      # в строке 2, тональность в строке
      saplout <- sapply((fy[[2]]),sepsoundtone)
      # Преобразовать в кадр данных
      tmpdf <- data.frame(fy[,1],t(saplout),row.names=NULL,
         stringsAsFactors=F)
      # Добавить имена в столбцы
      consname <- paste(names(fy)[[2]]," cons",sep="")
      restname <- paste(names(fy)[[2]]," sound",sep="")
      tonename <- paste(names(fy)[[2]]," tone",sep="")
      names(tmpdf) <- c("Ch char",consname,restname,tonename)
      # Необходимо использовать merge(), а не cbind(), из-за возможности
      # разного упорядочения fy, outdf
      outdf <- merge(outdf,tmpdf)
   }
   return(outdf)
}

# Разделяет транслитерацию произношения pronun на начальный согласный
# (если есть) остаток звука и тон (если есть).
sepsoundtone <- function(pronun) {
   nchr <- nchar(pronun)
   vowels <- c("a","e","i","o","u")
   # Сколько начальных согласных?
   numcons <- 0
   for (i in 1:nchr) {
      ltr <- substr(pronun,i,i)
      if (!ltr %in% vowels) numcons <- numcons + 1 else break
   }
   cons <- if (numcons > 0) substr(pronun,1,numcons) else NA
   tone <- substr(pronun,nchr,nchr)
   numtones <- tone %in% letters # T is 1, F is 0
   if (numtones == 1) tone <- NA
   therest <- substr(pronun,numcons+1,nchr-numtones)
   return(c(cons,therest,tone))
}
```

Код слияния не так уж прост. И даже этот код делает некоторые упрощающие допущения, и из него исключены некоторые важные случаи. Анализ текста — занятие не для слабонервных!

Как и следовало ожидать, процесс слияния начинается с вызова `merge()` в строке 3. Он создает новый кадр данных `outdf`, к которому будут присоединяться новые столбцы для отделенных компонентов звука.

Таким образом, настоящая работа заключается в разделении транслитерации на звуковые компоненты. Для этого в строке 5 запускается цикл по двум кадрам данных. При каждой итерации текущий кадр данных разбивается на звуковые компоненты, а результат присоединяется к `outdf` в строке 19. Обратите внимание на комментарий перед этой строкой, в котором говорится, что `cbind()` не подходит для этой ситуации.

Фактическое разбиение на компоненты происходит в главе 8. Здесь мы получаем столбец транслитераций следующего вида:

```
yat1
yuet3
ding1
chat1
naai5
gau2
```

Он разбивается на три столбца для начального согласного, остатка звука и тона. Например, `yat1` разбивается на `y`, `at` и 1. Эта задача естественно подходит для одной из разновидностей `appy`, и действительно, в строке 8 применяется `sapply()`. Конечно, для этого вызова необходимо написать применяемую функцию (при некотором везении могла бы отыскаться готовая работающая функция R, но здесь нам не повезло). Мы будем использовать функцию `sepsoundtone()`, начинающуюся в строке 26.

Функция `sepsoundtone()` интенсивно использует функцию R `substr()` (от «substring», то есть «подстрока»), которая подробно описана в главе 11. В строке 31, например, в цикле перебираются символы, пока не будут обнаружены все начальные согласные (например, ch). Возвращаемое значение в строке 40 состоит из трех компонентов звука, извлеченных из транслитерированной формы (формальный параметр `pronun`).

Обратите внимание на использование встроенной константы R `letters` в строке 37. С ее помощью мы проверяем, является ли заданный символ цифровым; это означало бы, что это тон. Некоторые транслитерации не имеют тона.

Затем строка 8 возвращает матрицу 3 × 1, которая содержит по одной строке для каждого из трех компонентов звука. Мы хотим преобразовать ее в кадр

данных для слияния с outdf в строке 19, и подготовка к преобразованию выполняется в строке 10.

Обратите внимание на вызов функции транспонирования матрицы t() для размещения информации в столбцах вместо строк. Это необходимо из-за того, что в кадрах данных информация хранится по столбцам. Кроме того, мы включаем столбец fy[,1] (сами китайские иероглифы), чтобы иметь общий столбец для вызова merge() в строке 19. А теперь обратимся к коду mapsound(), который на самом деле проще предыдущего кода слияния.

```
 1 mapsound <- function(df,fromcol,tocol,sourceval) {
 2    base <- which(df[[fromcol]] == sourceval)
 3    basedf <- df[base,]
 4    # Определить, какие строки basedf соответствуют различным
 5    # отображаемым значениям.
 6    sp <- split(basedf,basedf[[tocol]])
 7    retval <- list()
 8    retval$counts <- sapply(sp,nrow)
 9    retval$images <- sp
10    return(retval)
11 }
```

Напомню, что аргумент df представляет собой кадр данных с двумя фаньянями — вывод merge2fy(). Аргументы fromcol и tocol содержат имена исходного и отображенного столбцов. Строка sourceval содержит исходное отображаемое значение. Для конкретности рассмотрим предыдущие примеры, в которых переменная sourceval содержала x.

Прежде всего нужно определить, какие строки df соответствуют sourceval. Эта информация затем используется в строке 3 для извлечения нужного подкадра данных. В последнем кадре обратите внимание на форму, которую basedf[[tocol]] принимает в главе 6. Это будут значения, на которые отображается x, то есть ch, h и т. д. Строка 6 должна определить, какие строки basedf содержат те или иные отображаемые значения. Здесь используется функция R split(). Функция split() подробно рассматривается в разделе 6.2.2, но здесь важно то, что sp будет содержать список кадров данных: один для ch, один для h и т. д.

На этом завершается подготовка к главе 8. Так как sp содержит список кадров данных (по одному для каждого отображаемого значения), применение функции nrow() вызовом sapply() предоставит счетчики количества иероглифов для каждого из отображаемых значений — например, количество иероглифов, в которых происходит отображение x → ch (15, как видно из примера).

Из-за сложности кода стоит сказать пару слов о стиле программирования. Некоторые читатели могут справедливо заметить, что строки 2 и 3 можно заменить одной строкой:

```
basedf <- df[df[[fromcol]] == sourceval,]
```

Но, с моей точки зрения, эта строка с многочисленными квадратными скобками хуже читается. Лично я предпочитаю разбивать операции, если они становятся слишком сложными.

Аналогичным образом несколько последних строк кода можно сжать в еще одну однострочную конструкцию:

```
list(counts=sapply(sp,nrow),images=sp)
```

Среди прочего эта конструкция избавляется от вызова `return()`, что ощутимо ускоряет работу кода. Вспомните, что в R последнее значение, вычисленное функцией, автоматически возвращается функцией без вызова `return()`. Впрочем, такая экономия времени невелика и обычно не критична; еще раз скажу, что, по моему личному мнению, включение вызова `return()` выглядит более понятно.

6 Факторы и таблицы

Факторы лежат в основе многих мощных операций R, включая многие операции, выполняемые с табличными данными. Побудительные причины для применения факторов исходят от концепции *номинальных* (или *категорийных*) переменных в статистике. Такие значения по своей природе являются нечисловыми; они соответствуют категориям («демократ», «республиканец», «беспартийный» и т. д.), хотя и могут кодироваться в числовом виде.

В этой главе мы начнем с дополнительной информации, содержащейся в факторах, а затем сосредоточимся на функциях, используемых с факторами. Также будут рассмотрены таблицы и основные операции с таблицами.

6.1. Факторы и уровни

Фактор (factor) R можно рассматривать просто как вектор с дополнительной информацией (хотя, как вы вскоре увидите, во внутреннем представлении это не так). Эта дополнительная информация состоит из реестра различных значений в этом векторе, которые называются *уровнями* (levels). Пример:

```
> x <- c(5,12,13,12)
> xf <- factor(x)
> xf
[1] 5  12 13 12
Levels: 5 12 13
```

Уровнями здесь являются различные значения в `xf` — 5, 12 и 13.

Давайте заглянем вовнутрь:

```
> str(xf)
 Factor w/ 3 levels "5","12","13":1232
> unclass(xf)
```

```
[1] 1 2 3 2
attr(,"levels")
[1] "5" "12" "13"
```

А это уже интересно. В `xf` хранятся не значения (5, 12, 13, 12), а значения (1, 2, 3, 2). Таким образом, данные состоят сначала из значения уровня 1, затем из значений уровней 2 и 3 и, наконец, еще одного значения уровня 2. Таким образом, данные были перекодированы по уровню. Разумеется, сами уровни тоже сохранены, хотя и в виде символов (`"5"` вместо 5).

Длина фактора по-прежнему определяется в контексте длины данных, а не в контексте количества уровней или еще в каком-нибудь виде:

```
> length(xf)
[1] 4
```

Также можно резервировать будущие новые уровни, как в следующем примере:

```
> x <- c(5,12,13,12)
> xff <- factor(x,levels=c(5,12,13,88))
> xff
[1] 5  12 13 12
Levels: 5 12 13 88
> xff[2] <- 88
> xff
[1] 5  88 13 12
Levels: 5 12 13 88
```

В исходном виде `xff` не содержит значение 88, но при его определении допускается такая будущая возможность. Позднее это значение действительно было добавлено. Попытка подкинуть «незаконный» уровень по тому же принципу завершается неудачей. Вот как это выглядит:

```
> xff[2] <- 28
Warning message:
In `[<-.factor`(`*tmp*`, 2, value = 28) :
  invalid factor level, NAs generated
```

6.2. Типичные функции, используемые с факторами

С факторами используется еще один представитель семейства функций `apply` — `tapply`. Мы рассмотрим эту функцию, а также две другие функции, часто используемые с факторами: `split()` и `by()`.

6.2.1. Функция tapply()

Допустим, имеется вектор x с возрастами избирателей и фактор f с некоторой нечисловой характеристикой избирателей — например, партийная принадлежность (демократ, республиканец, беспартийный). Требуется найти средний возраст для каждой из партийных групп в x.

В типичном варианте использования вызов tapply(x,f,g) имеет вектор x, фактор или список факторов f и функцию g. Функцией g() в нашем маленьком примере будет встроенная функция R mean(). Если вы хотите сгруппировать данные по партийной принадлежности и другому показателю (скажем, полу), фактор f должен включать два других фактора (партийная принадлежность и пол).

Каждый фактор в f должен иметь такую же длину, как и x. В контексте приведенного выше примера с избирателями это логично: количество вариантов партийной принадлежности должно совпадать с количеством возрастов. Если компонент f является вектором, он будет преобразован в фактор применением к нему функции as.factor().

Функция tapply() (временно) разбивает x на группы, соответствующие уровням факторов (или комбинациям уровней в случае нескольких факторов), а затем применяет g() к полученным подвекторам x. Небольшой пример:

```
> ages <- c(25,26,55,37,21,42)
> affils <- c("R","D","D","R","U","D")
> tapply(ages,affils,mean)
 D  R  U
41 31 21
```

Посмотрим, что здесь произошло. Функция tapply() интерпретирует вектор ("R","D","D","R","U","D") как фактор с уровнями "D", "R" и "U". В данных указано, что"D" встречается в позициях с индексами 2, 3 и 6; "R" встречается в позициях с индексами 1 и 4; и "U" встречается в позиции с индексом 5. Для удобства будем называть три индексных вектора (2,3,6), (1,4) и (5) x, y и z соответственно. Затем tapply() вычисляет mean(u[x]), mean(u[y]) и mean(u[z]) и возвращает эти значения в виде вектора из трех элементов. А имена элементов этого вектора "D", "R" и "U" отражают уровни фактора, используемые функцией tapply().

А если факторов два и более? Каждый фактор создает набор групп, как в предыдущем примере, и объединяет группы операцией AND. Допустим, у вас имеется экономический набор данных с переменными для пола, возраста и дохода. В вызове tapply(x,f,g) x может быть доходом, а f — парой факторов: для пола и для признака возраста (старше/младше 25 лет). Требуется определить

средний доход с разбивкой по полу и возрасту. Если задать для `g()` функцию `mean()`, `tapply()` вернет средний доход в каждой из четырех подгрупп:

- мужчины младше 25 лет;
- женщины младше 25 лет;
- мужчины старше 25 лет;
- женщины старше 25 лет.

Небольшой учебный пример для этой конфигурации:

```
> d <- data.frame(list(gender=c("M","M","F","M","F","F"),
+   age=c(47,59,21,32,33,24),income=c(55000,88000,32450,76500,123000,45650)))
> d
  gender age  income
1      M  47  55000
2      M  59  88000
3      F  21  32450
4      M  32  76500
5      F  33 123000
6      F  24  45650
> d$over25 <- ifelse(d$age > 25,1,0)
> d
  gender age income over25
1      M  47  55000      1
2      M  59  88000      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0
> tapply(d$income,list(d$gender,d$over25),mean)
      0         1
F 39050 123000.00
M    NA  73166.67
```

Мы указали два фактора: пол и индикатор возраста. Так как каждый из этих факторов имеет два уровня, `tapply()` разбивает входные данные на четыре группы, по одной для каждой комбинации пола и возраста, а затем применяет функцию `mean()` к каждой группе.

6.2.2. Функция split()

В отличие от функции `tapply()`, которая разбивает вектор на группы, а затем применяет заданную функцию к каждой группе, `split()` останавливается на первой стадии и ограничивается формированием групп.

Базовая форма без каких-либо украшений и удобств имеет вид `split(x,f)`. Здесь `x` и `f` играют те же роли, что и при вызове `tapply(x,f,g)`; то есть `x` является вектором или кадром данных, а `f` — фактор или список факторов. Действием является разбиение `x` на группы, которые возвращаются в виде списка. (Заметим, что `x` может быть кадром данных с `split()`, но не с `tapply()`).

Проведем эксперимент с более ранним примером.

```
> d
  gender age income over25
1      M  47  55000      1
2      M  59  88000      1
3      F  21  32450      0
4      M  32  76500      1
5      F  33 123000      1
6      F  24  45650      0
> split(d$income,list(d$gender,d$over25))
$F.0
[1] 32450 45650

$M.0
numeric(0)

$F.1
[1] 123000

$M.1
[1] 55000 88000 76500
```

Вывод `split()` представляет собой список; вспомните, что компоненты списка обозначаются знаком `$`. Таким образом, последний вектор, например, назывался `"M.1"`; это имя является результатом объединения `"M"` из первого фактора и `1` из второго.

В качестве еще одной иллюстрации рассмотрим пример с абалонами из раздела 2.9.2. Мы хотим определить индексы элементов вектора, соответствующих мужским, женским и молодым особям. Данные в этом маленьком примере образуют один вектор с семью наблюдениями `("M","F","F","I","M","M","F")`, присвоенный `g`. Это можно моментально сделать при помощи `split()`.

```
> g <- c("M","F","F","I","M","M","F")
> split(1:7,g)
$F
[1] 2 3 7

$I
[1] 4
```

```
$M
[1] 1 5 6
```

Из результатов видно, что женские особи представлены записями 2, 3 и 7; молодая особь представлена записью 4; а мужские особи представлены записями 1, 5 и 6.

Проанализируем происходящее шаг за шагом. Вектор `g`, передаваемый в виде фактора, содержит три уровня: `"M"`, `"F"` и `"I"`. Индексы, соответствующие первому уровню, равны 1, 5 и 6, это означает, что `g[1]`, `g[5]` и `g[6]` содержат `"M"`. В результате R присваивает компоненту `M` вывода элементы 1, 5 и 6 вектора `1:7`, что дает вектор `(1,5,6)`.

Аналогичным образом можно упростить код в примере с анализом текста из раздела 4.2.4. В той задаче программа должна была ввести текстовый файл, определить, какие слова присутствуют в тексте, а затем вывести список со словами и их позициями в тексте. Использование функции `split()` позволит сократить объем кода:

```
1 findwords <- function(tf) {
2    # Прочитать слова из файла в вектор с режимом character
3    txt <- scan(tf,"")
4    words <- split(1:length(txt),txt)
5    return(words)
6 }
```

Вызов `scan()` возвращает список `txt` слов, прочитанных из файла `tf`. Таким образом, `txt[[1]]` содержит первое слово, прочитанное из файла, `txt[[2]]` — второе слово и т. д.; таким образом, значение `length(txt)` будет равно общему количеству прочитанных слов. Для конкретности предположим, что это число равно 220.

При этом сам список `txt` как второй аргумент функции `split()` будет получен как фактор. Уровнями этого фактора будут различные слова в файле. Если, например, в файле слово «world» встречается 6 раз, а слово «climate» — 10 раз, то «world» и «climate» станут первыми двумя уровнями `txt`.

Вызов `split()` определит, где эти и другие слова встречаются в тексте.

6.2.3. Функция by()

Предположим, в примере с данными абалонов требуется провести регрессионный анализ диаметра раковины относительно длины по отдельности для каждого кода пола: для мужских, женских и молодых особей. На первый взгляд

кажется, что ситуация идеально подходит для `tapply()`, но первым аргументом функции должен быть вектор, а не матрица или кадр данных. Применяемая функция может быть многомерной (например, `range()`), но на входе она должна получать вектор. С другой стороны, входными данными для регрессии должна быть матрица (или кадр данных), содержащая минимум два столбца: один для прогнозируемой переменной и один или несколько для переменных-предикторов. В нашем примере с данными абалонов матрица должна содержать столбец с данными диаметра и столбец для длины. Здесь можно воспользоваться функцией `by()`. Она работает как `tapply()` (которую она вызывает в своей внутренней реализации), но применяется к объектам, а не к векторам. Вот как она используется для нужного регрессионного анализа:

```
> aba <- read.csv("abalone.data",header=TRUE)
> by(aba,aba$Gender,function(m) lm(m[,2]~m[,3]))
aba$Gender: F

Call:
lm(formula = m[, 2] ~ m[, 3])

Coefficients:
(Intercept)       m[, 3]
    0.04288      1.17918

------------------------------------------------------------
aba$Gender: I

Call:
lm(formula = m[, 2] ~ m[, 3])

Coefficients:
(Intercept)       m[, 3]
    0.02997      1.21833

------------------------------------------------------------
aba$Gender: M

Call:
lm(formula = m[, 2] ~ m[, 3])

Coefficients:
(Intercept)       m[, 3]
    0.03653      1.19480
```

Вызовы `by()` очень похожи на вызовы `tapply()`; первый аргумент задает данные, второй — фактор группировки, а третий — функцию, применяемую к каждой группе.

Подобно тому как `tapply()` формирует группы индексов вектора в соответствии с уровнями фактора, этот вызов `by()` находит группы номеров строк кадра данных `aba`. В результате будут созданы три подкадра данных: по одному для каждого из уровней пола `M`, `F` и `I`.

Анонимная функция, которую мы определяем, выполняет регрессию второго столбца аргумента-матрицы `m` относительно третьего столбца. Эта функция будет вызвана трижды — по одному разу для каждого из трех подкадров данных, созданных ранее; таким образом будут созданы три регрессионных анализа.

6.3. Работа с таблицами

Наше знакомство с таблицами R начнется со следующего примера:

```
> u <- c(22,8,33,6,8,29,-2)
> fl <- list(c(5,12,13,12,13,5,13),c("a","bc","a","a","bc","a","a"))
> tapply(u,fl,length)
   a bc
5  2 NA
12 1  1
13 2  1
```

Здесь `tapply()` снова временно разбивает `u` на подвекторы, как было показано ранее, а затем применяет функцию `length()` к каждому подвектору. (Заметим, что происходящее не зависит от содержимого `u` — нас интересуют исключительно факторы.) Длины подвекторов определяют количества вхождений каждой из 3 × 2 = 6 комбинаций двух факторов. Например, 5 дважды встречается с `"a"` и вовсе не встречается с `"bc"`; отсюда элементы 2 и NA в первой строке вывода. В статистике такая структура называется *факторной таблицей*.

В этом примере есть одна проблема: значение `NA`. В действительности оно должно быть равно 0, это означает, что ни в одном случае первый фактор не имеет уровень 5 при втором факторе с уровнем `"bc"`. Функция `table()` правильно создает факторные таблицы.

```
> table(fl)
    fl.2
fl.1 a bc
  5  2  0
  12 1  1
  13 2  1
```

Первым аргументом вызова `table()` является либо фактор, либо список факторов. В данном случае используются два фактора — `(5,12,13,12,13,5,13)`

и (`"a","bc","a","a","bc", "a","a"`), при этом объект, который может интерпретироваться как фактор, считается таковым.

Обычно в аргументе данных `table()` передается кадр данных. Например, предположим, что файл ct.dat состоит из данных предвыборных опросов на переизбрание кандидата X. Файл ct.dat выглядит примерно так:

```
"Vote for X" "Voted For X Last Time"
"Yes" "Yes"
"Yes" "No"
"No" "No"
"Not Sure" "Yes"
"No" "No"
```

Как это часто бывает в статистике, каждая строка файла представляет одного участника опроса. В данном случае пяти людям были заданы следующие два вопроса:

- Собираетесь ли вы голосовать за кандидата X?
- Голосовали ли вы за кандидата X на прошлых выборах?

В файле данных появляются пять строк. Прочитаем данные из файла:

```
> ct <- read.table("ct.dat",header=T)
> ct
  Vote.for.X Voted.for.X.Last.Time
1        Yes                   Yes
2        Yes                    No
3         No                    No
4   Not Sure                   Yes
5         No                    No
```

Функцией `table()` можно воспользоваться для построения факторной таблицы для этих данных:

```
> cttab <- table(ct)
> cttab
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

Число 2 в левом верхнем углу таблицы показывает, например, что двое людей ответили «нет» на первый и второй вопросы. Число 1 в правой части средней строки указывает, что один человек ответил «не уверен» на первый вопрос и «да» на второй вопрос.

Также можно получить одномерные счетчики для одного фактора:

```
> table(c(5,12,13,12,8,5))
 5  8 12 13
 2  1  2  1
```

А в этом примере трехмерной таблицы учитывается пол избирателей, расовая принадлежность (белые, черные, азиаты и т. д.) и политические взгляды (либерал или консерватор):

```
> v # кадр данных
  gender race pol
1 M W L
2 M W L
3 F A C
4 M O L
5 F B L
6 F B C
> vt <- table(v)
> vt
, , pol = C
     race
gender A B O W
     F 1 1 0 0
     M 0 0 0 0

, , pol=L

     race
gender A B O W
    F  0 1 0 0
    M  0 0 1 2
```

R выводит трехмерную таблицу в виде последовательности двумерных таблиц. В данном случае генерируется таблица пола и расы для консерваторов, а затем соответствующая таблица для либералов. Например, вторая двумерная таблица сообщает, что среди участников было два белых либерала мужского пола.

6.3.1. Операции матриц/массивов с таблицами

Многие (нематематические) операции матриц/массивов могут использоваться с кадрами данных, но они также могут применяться и к таблицам. (И это неудивительно, если учесть, что часть объекта таблицы со счетчиками ячеек представляет собой массив.)

Например, можно обратиться к счетчикам ячеек при помощи матричной записи. Воспользуемся ею в примере с голосованием из предыдущего раздела:

```
> class(cttab)
[1] "table"
> cttab[1,1]
[1] 2
> cttab[1,]
 No Yes
  2   0
```

Во второй команде, несмотря на то что первая команда сообщила, что `cttab` имеет класс `cttab`, мы интерпретируем `cttab` как матрицу и выводим ее «элемент `[1,1]`». Продолжая эту идею, третья команда выводит первый столбец этой «матрицы».

Матрицу можно умножить на скаляр. Например, вот как изменить счетчики ячеек пропорциональными величинами:

```
> cttab/5
          Voted.for.X.Last.Time
Vote.for.X  No Yes
  No       0.4 0.0
  Not Sure 0.0 0.2
  Yes      0.2 0.2
```

В статистике *предельными значениями* (marginal value) переменной называются значения, полученные при суммировании других переменных, тогда как эта переменная остается постоянной. В примере с голосованием предельными значениями переменной `Vote.for.X` будут 2+0=2, 0+1=1 и 1+1=2. Конечно, эти значения можно получить при помощи матричной функции `apply()`:

```
> apply(cttab,1,sum)
      No Not Sure       Yes
       2        1         2
```

Обратите внимание: метки (такие, как «No») берутся из имен строк матрицы, построенной функцией `table()`.

Однако R предоставляет для этой цели, то есть для вычисления предельных сумм, функцию `addmargins()`. Пример:

```
> addmargins(cttab)
          Voted.for.X.Last.Time
Vote.for.X No Yes Sum
  No        2   0   2
```

```
  Not Sure  0   1   1
  Yes       1   1   2
  Sum       3   2   5
```

Здесь мы получаем предельные данные для удобства наложенные на исходную таблицу сразу по обоим измерениям.

Для получения имен измерений и уровней используется функция `dimnames()`:

```
> dimnames(cttab)
$Vote.for.X
[1] "No"       "Not Sure" "Yes"

$Voted.for.X.Last.Time
[1] "No"  "Yes"
```

6.3.2. Расширенный пример: извлечение подтаблицы

Продолжим пример с данными голосования:

```
> cttab
           Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

Предположим, вы хотите представить эти данные на встрече, ограничившись теми респондентами, которые знают, что они будут голосовать на текущих выборах. Другими словами, вы хотите исключить записи `Not Sure` и представить подтаблицу, которая выглядит так:

```
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Yes       1   1
```

Приведенная ниже функция `subtable()` выполняет извлечение подтаблиц. Она получает два аргумента:

- `tbl`: исходная таблица (класс `"table"`);
- `subnames`: список, определяющий параметры извлечения таблицы. Каждому компоненту этого списка присваивается имя по некоторому измерению `tbl`, а значением этого компонента является вектор имен нужных уровней.

Прежде чем рассматривать код, посмотрим, что происходит в этом примере. Аргумент `cttab` представляет двумерную таблицу с именами измерений `Voted.for.X` и `Voted.for.X.Last.Time`. В этих двух измерениях уровни называются `No`, `Not Sure` и `Yes` в первом измерении и `No` и `Yes` во втором. Из них мы хотим исключить случаи `Not Sure`, так что фактический аргумент, соответствующий формальному аргументу `subnames`, выглядит так:

```
list(Vote.for.X=c("No","Yes"),Voted.for.X.Last.Time=c("No","Yes"))
```

Попробуем вызвать эту функцию:

```
> subtable(cttab,list(Vote.for.X=c("No","Yes"),
+    Voted.for.X.Last.Time=c("No","Yes")))
          Voted.for.X.Last.Time
Vote.for.X No Yes
       No   2   0
       Yes  1   1
```

Итак, вы получили представление о том, что делает эта функция. Пора взглянуть на ее внутреннее устройство.

```
 1 subtable <- function(tbl,subnames) {
 2    # Получить массив счетчиков ячеек в tbl
 3    tblarray <- unclass(tbl)
 4    # Мы получим подмассив счетчиков ячеек, соответствующих подименам,
 5    # вызовом do.call() для функции "["; сначала необходимо построить
 6    # список аргументов.
 7    dcargs <- list(tblarray)
 8    ndims <- length(subnames) # Количество измерений
 9    for (i in 1:ndims) {
10       dcargs[[i+1]] <- subnames[[i]]
11    }
12    subarray <- do.call("[",dcargs)
13    # Построение новой таблицы, состоящей из подмассива, количества
14    # уровней по каждому измерению и значения dimnames(), а также
15    # атрибута класса "table".
16    dims <- lapply(subnames,length)
17    subtbl <- array(subarray,dims,dimnames=subnames)
18    class(subtbl) <- "table"
19    return(subtbl)
20 }
```

Что же здесь происходит? Чтобы подготовиться к написанию этого кода, я сначала провел небольшое исследование для определения структуры объектов класса `"table"`. Просмотрев код функции `table()`, я обнаружил, что на базовом уровне объект класса `"table"` состоит из массива, элементами которого являются

счетчики ячеек. Таким образом, стратегия заключалась в извлечении нужного подмассива, добавления имен в измерения подмассива и последующего назначения статуса класса "table" результату. Первой задачей этого кода было извлечение подмассива, соответствующего нужной пользователю подтаблице; эта задача занимает большую часть кода. В строке 3 мы сначала извлекаем полный массив счетчиков ячеек, сохраняя его в tblarray. Вопрос в том, как использовать эту информацию для нахождения нужного подмассива. Теоретически это просто; на практике иногда возникают сложности.

Чтобы получить нужный подмассив, необходимо сформировать выражение сегментации для массива tblarray — примерно такое:

```
tblarray[диапазоны индексов]
```

В нашем примере с голосованием выражение выглядит так:

```
tblarray[c("No","Yes"),c("No","Yes")]
```

На концептуальном уровне задача нетрудная, но решить ее напрямую сложно, потому что tblarray может иметь разное количество измерений (два, три или сколько угодно). Вспомните, что индексирование массивов R в действительности осуществляется функцией с именем "["(). Эта функция получает переменное число аргументов: два для двумерных массивов, три — для трехмерных и т. д.

Задача решается при помощи функции R do.call(). Базовая форма этой функции выглядит так:

```
do.call(f,argslist)
```

где f — функция, а argslist — список аргументов, для которых вызывается f().

Другими словами, код, по сути, делает следующее:

```
f(argslist[[1]],argslist[[2]],...)
```

Это упрощает вызов функции с переменным количеством аргументов.

В нашем примере необходимо сформировать список, состоящий сначала из tblarray, а затем из желательных уровней для каждого измерения. Наш список выглядит так:

```
list(tblarray,Vote.for.X=c("No","Yes"),Voted.for.X.Last.Time=c("No","Yes"))
```

Строки 7–11 строят этот список для общего случая; это наш подмассив. Затем к нему необходимо прикрепить имена и назначить класс "table". Первая опе-

рация может выполняться функцией R `array()`, которая получает следующие аргументы:

- `data`: данные, которые должны быть помещены в новый массив (`subarray` в нашем случае);
- `dim`: длины измерений (количество строк, количество столбцов, количество слоев и т. д.). В нашем случае это значение `ndims`, вычисленное в строке 16;
- `dimnames`: имена измерений и имена их уровней, уже предоставленные пользователем в аргументе `subnames`.

На концептуальном уровне функция получилась довольно сложной, но стоит разобраться во внутренней структуре класса `"table"`, и все упрощается.

6.3.3. Расширенный пример: поиск максимальных ячеек в таблице

Просматривать очень большую таблицу с большим количеством строк или измерений неудобно. В такой ситуации пользователь может обратить первоочередное внимание на ячейки с наибольшими частотами. В этом ему поможет функция `tabdom()`, которую мы разработаем ниже, — эта функция сообщает доминирующие частоты в таблице. Простой вызов этой функции выглядит так:

```
tabdom(tbl,k)
```

Функция сообщает, какие ячейки в таблице `tbl` содержат `k` наибольших частот. Пример:

```
> d <- c(5,12,13,4,3,28,12,12,9,5,5,13,5,4,12)
> dtab <- table(d)
> tabdom(dtab,3)
   d Freq
3  5    4
5 12    4
2  4    2
```

Функция сообщает, что значения 5 и 12 имели наибольшую частоту в `d` (по четыре экземпляра каждое), а следующим по частоте значением было 4 с двумя экземплярами (3, 5 и 2 слева — избыточная информация; преобразование таблицы в кадр данных рассматривается ниже).

Другой пример: возьмем таблицу cttab из примеров предыдущих разделов:

```
> tabdom(cttab,2)
  Vote.for.X Voted.For.X.Last.Time Freq
1         No                    No    2
3        Yes                    No    1
```

Комбинация No-No была самой частой (два экземпляра), тогда как второй по частоте была комбинация Yes-No с одним экземпляром[1].

Как это делается? Код выглядит сложно, но на самом деле задача упрощается одним трюком, который использует тот факт, что данные могут представляться в формате кадров данных. Снова воспользуемся таблицей cttab:

```
> as.data.frame(cttab)
  Vote.for.X Voted.For.X.Last.Time Freq
1         No                    No    2
2   Not Sure                    No    0
3        Yes                    No    1
4         No                   Yes    0
5   Not Sure                   Yes    1
6        Yes                   Yes    1
```

Обратите внимание: это *не* исходный кадр данных ct, на основе которого была построена таблица cttab, а просто другое представление самой таблицы. Для каждой комбинации факторов существует одна строка, а добавленный столбец Freq содержит количество экземпляров каждой комбинации. Благодаря ему задача решается достаточно просто.

```
 1 # Находит в таблице tbl ячейки с k наибольшими частотами;
 2 # обработка одинаковых результатов не реализована
 3 tabdom <- function(tbl,k) {
 4    # Создать представление tbl в виде кадра данных, добавить столбец Freq
 5    tbldf <- as.data.frame(tbl)
 6    # Определить позиции частот в порядке сортировки.
 7    freqord <- order(tbldf$Freq,decreasing=TRUE)
 8    # Упорядочить кадр данных в этом порядке, взять первые k строк.
 9    dom <- tbldf[freqord,][1:k,]
10 return(dom)
11 }
```

С комментариями этот код не требует особых объяснений.

[1] Но ведь комбинации Not Sure-Yes и Yes-Yes тоже встречаются в одном экземпляре, а значит, должны разделить второе место с Yes-No? Да, определенно. Мой код небрежно относится к разрешению ничейных результатов; читатель может доработать его самостоятельно.

Решение с сортировкой в строке 7, использующее функцию `order()`, демонстрирует стандартный способ сортировки кадров данных (его стоит запомнить, потому что ситуация возникает достаточно часто).

Подход, выбранный здесь, — преобразование таблицы в кадр данных — также можно было использовать в разделе 6.3.2. При этом нужно действовать осторожно и исключить уровни из факторов, чтобы избежать нулей в ячейках.

6.4. Другие функции для работы с факторами и таблицами

R также включает ряд других функций для работы с таблицами и факторами. Мы обсудим две из них — `aggregate()` и `cut()`.

ПРИМЕЧАНИЕ

Пакет reshape Хэдли Уикхэма (Hadley Wickham) «позволяет гибко изменять структуру и обобщать данные с использованием всего двух функций melt и cast». Вероятно, на освоение этого пакета вам потребуется какое-то время, но он исключительно мощен. Пакет plyr того же автора тоже весьма универсален. Оба пакета можно загрузить из репозитория R CRAN. За дополнительной информацией о загрузке и установке пакетов обращайтесь к приложению Б.

6.4.1. Функция aggregate()

Функция `aggregate()` может вызвать `tapply()` по одному разу для каждой переменной в группе. Например, в данных абалонов можно вычислить медиану по каждой переменной с разбиением по полу:

```
> aggregate(aba[,-1],list(aba$Gender),median)
  Group.1 Length Diameter Height WholeWt ShuckedWt ViscWt ShellWt Rings
  1       F  0.590    0.465  0.160 1.03850   0.44050 0.2240   0.295 10
  2       I  0.435    0.335  0.110 0.38400   0.16975 0.0805   0.113  8
  3       M  0.580    0.455  0.155 0.97575   0.42175 0.2100   0.276 10
```

Первый аргумент, `aba[,-1]`, содержит весь кадр данных, кроме первого столбца, — собственно `Gender`. Второй аргумент, который должен быть списком, содержит фактор `Gender`, как и прежде. Наконец, третий аргумент приказывает R вычислить медиану по каждому столбцу для каждого из кадров данных, сгенерированных в результате выделения подгрупп, соответствующих нашим

факторам. В данном примере используются три такие подгруппы, поэтому выходные данные `aggregate()` состоят из трех строк.

6.4.2. Функция cut()

Стандартный способ генерирования факторов, особенно для таблиц, основан на использовании функции `cut()`. Функция получает вектор данных `x` и набор категорий, определяемых вектором `b`. Функция определяет, к какой категории относится каждый из элементов `x`. В данном примере используется следующая форма вызова:

```
y <- cut(x,b,labels=FALSE)
```

где категории определяются полуоткрытыми интервалами `(b[1],b[2]]`, `(b[2], b[3]]`, ... Пример:

```
> z
[1] 0.88114802 0.28532689 0.58647376 0.42851862 0.46881514 0.24226859
    0.05289197
[8] 0.88035617
> seq(from=0.0,to=1.0,by=0.1)
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> binmarks <- seq(from=0.0,to=1.0,by=0.1)
> cut(z,binmarks,labels=F)
[1]93655319
```

Он показывает, что значение `z[1]` (0,88114802) попало в группу 9, которая соответствует интервалу (0,8,0,9], значение `z[2]` (0,28532689) — в группу 3 и т. д.

Функция возвращает вектор, как видно из результата. Однако его можно преобразовать в фактор или использовать для построения таблицы. Например, на этой основе можно написать специализированную функцию построения гистограммы (также для этого пригодится функция R `findInterval()`).

7 Программные конструкции

R является языком с блочной структурой, как и многие языки, произошедшие от ALGOL, такие как C, C++, Python, Perl и т. д. Как вы уже видели, блоки заключаются в фигурные скобки; хотя если блок состоит всего из одной команды, фигурные скобки не обязательны. Команды разделяются символами новой строки или символами ; (по желанию).

В этой главе рассматриваются основные конструкции R как языка программирования. Мы рассмотрим некоторые особенности циклов и других похожих команд, а затем перейдем прямо к теме функций, которой будет посвящена большая часть главы.

В частности, исключительно важны вопросы области видимости переменных. Как и во многих сценарных языках, в R переменные не «объявляются». Программисты с опытом работы на C на первый взгляд увидят некоторое сходство с R, но потом поймут, что структура видимости в R обладает более широкими возможностями.

7.1. Управляющие команды

Управляющие команды R очень похожи на стандартные конструкции языков семейства ALGOL, упоминавшихся выше. В этом разделе рассматриваются циклы и команды `if-else`.

7.1.1. Циклы

В разделе 1.3 мы определили функцию `oddcount()`. В этой функции следующая строка моментально покажется знакомой программистам Python:

```
for (n in x) {
```

Это означает, что для каждого компонента вектора x будет выполнена одна итерация цикла, при этом переменная n будет принимать значения этих компонентов — при первой итерации n = x[1], при второй итерации n = x[2] и т. д. Например, следующий код использует эту конструкцию для вывода квадратов всех элементов вектора:

```
> x <- c(5,12,13)
> for (n in x) print(n^2)
[1] 25
[1] 144
[1] 169
```

Кроме того, доступны конструкции while и repeat в стиле C, а также break — команда для выхода из цикла. В следующем примере используются все три конструкции:

```
> i <-1
> while (i <= 10) i <- i+4
> i
[1] 13
>
> i <- 1
> while(TRUE) { # Аналогично предыдущему циклу
+ i <- i+4
+ if (i > 10) break
+ }
> i
[1] 13
>
> i <- 1
> repeat { # Еще один аналог
+ i <- i+4
+ if (i > 10) break
+ }
> i
[1] 13
```

В первом фрагменте в процессе выполнения итераций цикла переменная i принимает значения 1, 5, 9 и 13. В последнем случае условие i<=10 не выполняется, поэтому срабатывает команда break, и программа выходит из цикла.

Пример демонстрирует три разных способа достижения одной цели, при этом команда break играет ключевую роль во втором и третьем варианте.

Следует помнить, что в конструкции repeat нет логического условия выхода. Необходимо использовать break (или некий аналог return). Конечно, команда

break может использоваться и с циклами for. Другая полезная команда, next, приказывает интерпретатору пропустить оставшуюся часть текущей итерации и перейти сразу к следующей. Это позволяет избежать сложных вложенных конструкций if-then-else, перегружающих код. Следующий пример, демонстрирующий использование next, взят из расширенного примера главы 8:

```
 1 sim <- function(nreps) {
 2    commdata <- list()
 3    commdata$countabsamecomm <- 0
 4    for (rep in 1:nreps) {
 5       commdata$whosleft <- 1:20
 6       commdata$numabchosen <- 0
 7       commdata <- choosecomm(commdata,5)
 8       if (commdata$numabchosen > 0) next
 9       commdata <- choosecomm(commdata,4)
10       if (commdata$numabchosen > 0) next
11       commdata <- choosecomm(commdata,3)
12    }
13    print(commdata$countabsamecomm/nreps)
14 }
```

Команды next присутствуют в строках 8 и 10. Посмотрим, как они работают и чем они лучше своих альтернатив. Две команды next находятся в цикле, который начинается в строке 4. Затем, когда условие if в строке 8 выполняется, строки 9–11 будут пропущены и управление будет передано в строку 4. Ситуация в строке 10 аналогична.

Без использования next нам пришлось бы воспользоваться вложенными командами if примерно такого вида:

```
 1 sim <- function(nreps) {
 2    commdata <- list()
 3    commdata$countabsamecomm <- 0
 4    for (rep in 1:nreps) {
 5       commdata$whosleft <- 1:20
 6       commdata$numabchosen <- 0
 7       commdata <- choosecomm(commdata,5)
 8       if (commdata$numabchosen == 0) {
 9          commdata <- choosecomm(commdata,4)
10          if (commdata$numabchosen == 0)
11             commdata <- choosecomm(commdata,3)
12       }
13    }
14    print(commdata$countabsamecomm/nreps)
15 }
```

Поскольку в этом простом примере всего два уровня, все не так плохо. Тем не менее при большем количестве уровней вложенные команды `if` быстро усложняются. Конструкция `for` работает с любыми векторами независимо от режима. Например, можно перебрать элементы вектора с именами файлов. Допустим, имеется файл **file1** со следующим содержимым:

```
1
2
3
4
5
6
```

Также имеется файл с именем `file2`:

```
5
12
13
```

Следующий цикл читает и выводит каждый из файлов. Функция `scan()` используется для чтения файла с числовыми данными, эти значения сохраняются в векторе. Функция `scan()` рассматривается в главе 10.

```
> for (fn in c("file1","file2")) print(scan(fn))
Read 6 items
[1] 1 2 3 4 5 6
Read 3 items
[1] 5 12 13
```

Итак, `fn` сначала присваивается `file1`, программа читает и выводит содержимое этого файла. Затем то же самое происходит с файлом `file2`.

7.1.2. Перебор невекторных множеств

R не поддерживает напрямую перебор по невекторным множествам, но есть пара косвенных, но несложных способов решения этой задачи:

- Используйте функцию `lapply()` (предполагается, что итерации цикла не зависят друг от друга, а следовательно, могут выполняться в любом порядке).
- Используйте функцию `get()`. Функция получает в аргументе символьную строку с именем некоторого объекта и возвращает объект с этим именем. Вроде бы тривиально, но `get()` — очень мощная функция.

Рассмотрим пример использования `get()`. Допустим, имеются две матрицы, `u` и `v`, содержащие статистические данные, и мы хотим применить функцию линейной регрессии R `lm()` к каждой из них.

```
> u
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    4
> v
     [,1] [,2]
[1,]    8   15
[2,]   12   10
[3,]   20    2
> for (m in c("u","v")) {
+    z <- get(m)
+    print(lm(z[,2] ~ z[,1]))
+ }

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept) z[, 1]
    -0.6667 1.5000

Call:
lm(formula = z[, 2] ~ z[, 1])

Coefficients:
(Intercept) z[, 1]
     23.286 -1.071
```

Сначала переменной `m` присваивается `u`. Затем следующие строки присваивают матрицу `u` переменной `z`, что позволяет вызвать `lm()` для `u`:

```
z <- get(m)
print(lm(z[,2] ~ z[,1]))
```

Затем то же самое происходит с `v`.

7.1.3. if-else

Синтаксис `if-else` выглядит так:

```
if (r == 4) {
   x <- 1
} else {
```

```
    x <- 3
    y <- 4
}
```

Выглядит просто, но здесь есть один важный нюанс. Секция `if` состоит из всего одной команды:

```
x <- 1
```

Напрашивается предположение, что заключать эту команду в фигурные скобки не нужно. Тем не менее фигурные скобки здесь обязательны.

Правая фигурная скобка перед `else` используется парсером R, который заключает, что выполняется конструкция `if-else` вместо простой конструкции `if`. В интерактивном режиме без фигурных скобок парсер ошибочно решит, что имеет дело со вторым вариантом, и будет действовать соответственно, а это не то, что нам нужно.

Команда `if-else` работает как вызов функции, поэтому она возвращает последнее присвоенное значение.

```
v <- if (условие) выражение1 else выражение2
```

Команда присваивает `v` результат *выражения1* или *выражения2* в зависимости от того, истинно условие или нет. Этот факт может использоваться для улучшения компактности кода. Простой пример:

```
> x <- 2
> y <- if(x == 2) x else x+1
> y
[1] 2
> x <- 3
> y <- if(x == 2) x else x+1
> y
[1] 4
```

Без этого приема код

```
y <- if(x == 2) x else x+1
```

получится более громоздким:

```
if(x == 2) y <- x else y <- x+1
```

В более сложных примерах *выражение1* и/или *выражение2* могут быть вызовами функций. С другой стороны, компактность все же не должна достигаться в ущерб ясности кода.

При работе с факторами используйте функцию `iselse()`, описанную в главе 2, — вероятно, такой код будет работать быстрее.

7.2. Арифметические и логические операторы и значения

Базовые операторы перечислены в табл. 7.1.

Таблица 7.1. Базовые операторы R

Операция	Описание
x + y	Сложение
x – y	Вычитание
x * y	Умножение
x / y	Деление
x ^ y	Возведение в степень
x %% y	Вычисление остатка
x %/% y	Целочисленное деление
x == y	Проверка равенства
x <= y	Проверка условия «меньше либо равно»
x >= y	Проверка условия «больше либо равно»
x && y	Логическая операция AND для скаляров
x || y	Логическая операция OR для скаляров
x & y	Логическая операция AND для векторов (вектор x, y, результат)
x | y	Логическая операция OR для векторов (вектор x, y, результат)
!x	Логическое отрицание

Хотя в R формально нет скалярных типов (скалярные значения рассматриваются как векторы с одним элементом), в табл. 7.1 встречается исключение: разные логические операторы для скалярных и векторных случаев. На первый взгляд это странно, но простой пример показывает, для чего нужны две разновидности одной операции.

```
> x
[1] TRUE FALSE TRUE
> y
[1] TRUE TRUE FALSE
> x&y
[1] TRUE FALSE FALSE
> x[1] && y[1]
[1] TRUE
> x && y # Проверяет только первые элементы обоих векторов
[1] TRUE
> if (x[1] && y[1]) print("both TRUE")
[1] "both TRUE"
> if (x & y) print("both TRUE")
[1] "both TRUE"
Warning message:
In if (x & y) print("both TRUE") :
  the condition has length > 1 and only the first element will be used
```

Суть в том, что при определении результата `if` необходимо только одно логическое значение, а не вектор логических значений; отсюда предупреждение в приведенном примере, а также необходимость в двух разных операторах, `&` и `&&`.

Логические значения `TRUE` и `FALSE` можно сократить до `T` и `F` (в обоих случаях верхний регистр обязателен). В арифметических выражениях эти значения заменяются 1 и 0:

```
> 1 < 2
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
> (1 < 2) * (3 < 4) * (5 < 1)
[1] 0
> (1 < 2) == TRUE
[1] TRUE
>(1 < 2) == 1
[1] TRUE
```

Например, во второй проверке сравнение `1 < 2` возвращает `TRUE`, а `3 < 4` также возвращает `TRUE`. Оба значения интерпретируются как 1, так что произведение равно 1.

На поверхности функции R кажутся похожими на функции C, Java и т. д. Тем не менее у них гораздо больше общего с конструкциями функционального программирования, что имеет прямые последствия для программистов R.

7.3. Значения по умолчанию для аргументов

В разделе 5.1.2 набор данных читается из файла с именем `exams`:

```
> testscores <- read.table("exams",header=TRUE)
```

Аргумент `header=TRUE` сообщает R о наличии строки заголовка, чтобы первая строка файла не интерпретировалась как данные.

Ниже приведен пример использования *именованных аргументов*. Несколько первых строк файла:

```
> read.table
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), encoding = "unknown")
{
    if (is.character(file)) {
        file <- file(file, "r")
        on.exit(close(file))
...
...
```

Второму формальному аргументу присвоено имя `header`. Поле `=FALSE` означает, что этот аргумент необязателен, и если он не задан, по умолчанию будет использоваться значение `FALSE`. Если значение по умолчанию вас не устраивает, аргумент должен быть указан при вызове по имени:

```
> testscores <- read.table("exams",header=TRUE)
```

Отсюда и термин *именованный аргумент*.

Учтите, что R использует *отложенное вычисление* — выражение не вычисляется до того момента, пока это не станет необходимо. Может оказаться, что именованный аргумент не будет использоваться в программе.

7.4. Возвращаемые значения

Возвращаемым значением функции может быть любой объект R. Хотя возвращаемое значение часто представляет собой список, им может быть даже другая функция.

Значение можно явно вернуть на сторону вызова явным выполнением `return()`. Без этого вызова по умолчанию будет возвращено значение последней выполненной команды. Для примера рассмотрим `oddcount()` из главы 1:

```
> oddcount
function(x) {
   k <- 0 # Присвоить k значение 0
   for (n in x) {
      if (n %% 2 == 1) k <- k+1 # %% – оператор вычисления остатка
   }
   return(k)
}
```

Функция возвращает количество нечетных чисел в аргументе. Код можно немного упростить, исключив из него вызов `return()`. Для этого возвращаемое значение `k` включается как последняя команда в коде функции:

```
oddcount <- function(x) {
   k <- 0
   pagebreak
   for (n in x) {
      if (n %% 2 == 1) k <- k+1
   }
   k
}
```

С другой стороны, взгляните на следующий код:

```
oddcount <- function(x) {
   k<-0
   for (n in x) {
      if (n %% 2 == 1) k <- k+1
   }
}
```

Код работать не будет по довольно неочевидной причине: последней выполненной командой является вызов `for()`, возвращающий значение `NULL` (и делает это *прозрачно* в терминологии R; это означает, что если значение не сохраняется присваиванием, то оно будет потеряно). Таким образом, возвращаемого значения здесь вообще нет.

7.4.1. Нужен ли явный вызов return()?

Преобладающая идиома R — нежелательность явных вызовов `return()`. Одна из причин заключается в том, что вызов функции увеличивает время выполнения.

Тем не менее, кроме очень коротких функций, сэкономленное время пренебрежимо мало, так что это может быть не самой серьезной причиной для того, чтобы избегать `return()`. Тем не менее обычно она не нужна.

Возьмем второй пример из предыдущего раздела:

```
oddcount <- function(x) {
   k <- 0
   for (n in x) {
      if (n %% 2 == 1) k <- k+1
   }
   k
}
```

Здесь последовательность команд просто завершается возвращаемым выражением — в данном случае `k`. Вызов `return()` не обязателен. Чтобы приводимый в книге код был более понятным, я обычно включаю вызов `return()`, но в реальном коде принято его опускать.

Тем не менее качественное проектирование означает, что вы должны с одного взгляда на код функции немедленно распознать различные точки, в которых управление может возвращаться на сторону вызова. Для этого проще всего включить явный вызов `return()` во всех строках в середине кода, которые могут вернуть управление. (Вызов `return()` в конце функции при желании можно опустить.)

7.4.2. Возвращение сложных объектов

Так как возвращаемым значением должен быть любой объект R, функции могут возвращать сложные объекты. Пример:

```
> g
function() {
   t <- function(x) return(x^2)
   return(t)
}
> g()
function(x) return(x^2)
<environment: 0x8aafbc0>
```

Если ваша функция возвращает несколько значений, поместите их в список или другой контейнер.

7.5. Функции как объекты

Функции R являются *полноправными объектами* (класса `"function"`, конечно); это означает, что в большинстве случаев их можно использовать так же, как любые другие объекты. Это видно по синтаксису создания функций:

```
> g <- function(x) {
+    return(x+1)
+ }
```

Здесь `function()` — встроенная функция R, предназначенная для создания функций! В правой части в действительности содержатся два аргумента `function()`: список формальных аргументов создаваемой функции (в данном случае только `x`) и тело функции (в данном случае только одна команда `return(x+1)`). Второй аргумент должен относиться к классу `"expression"`. Таким образом, здесь важно то, что правая часть создает объект функции, который затем присваивается `g`.

Кстати говоря, даже `"{"` является функцией, в чем нетрудно убедиться:

```
> ?"{"
```

Задача этой функции — создать единое целое из нескольких команд. К этим двум аргументам `function()` можно обратиться позднее при помощи функций R `formals()` и `body()`:

```
> formals(g)
$x
> body(g)
{
   return(x + 1)
}
```

Вспомните, что в интерактивном режиме при вводе имени объекта этот объект выводится на экран. Функции не являются исключением, поскольку они такие же объекты, как и все остальное.

```
> g
function(x) {
   return(x+1)
}
```

Это будет удобно, если вы написали функцию, а потом забыли подробности реализации. Вывод функции пригодится и в том случае, если вы не совсем точно представляете, что делает библиотечная функция R. Просматривая код, вы сможете лучше понять его. Например, если вы забыли какие-то тонкости

поведения графической функции abline(), код этой функции поможет лучше понять, как ею пользоваться.

```
> abline
function (a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,
    coef = NULL, untf = FALSE, ...)
{
    int_abline <- function(a, b, h, v, untf, col = par("col"),
        lty = par("lty"), lwd = par("lwd"), ...) .Internal(abline(a,
        b, h, v, untf, col, lty, lwd, ...))
    if (!is.null(reg)) {
        if (!is.null(a))
            warning("'a' is overridden by 'reg'")
        a<-reg
    }

    if (is.object(a) || is.list(a)) {
        p <- length(coefa <- as.vector(coef(a)))
...
...
```

Если вы хотите просмотреть длинную функцию этим способом, передайте ее функции page():

```
> page(abline)
```

Также можно воспользоваться функцией edit(), которая будет рассматриваться в разделе 7.11.2.

Учтите, что некоторые фундаментальные встроенные функции R написаны на C, поэтому просмотреть их таким способом не удастся. Пример:

```
> sum
function (..., na.rm = FALSE) .Primitive("sum")
```

Поскольку функции являются объектами, их также можно присваивать, передавать в аргументах другим функциям и т. д.

```
> f1 <- function(a,b) return(a+b)
> f2 <- function(a,b) return(a-b)
> f <- f1
> f(3,2)
[1] 5
> f <- f2
> f(3,2)
[1] 1
> g <- function(h,a,b) h(a,b)
```

```
> g(f1,3,2)
[1] 5
> g(f2,3,2)
[1] 1
```

А поскольку функции являются объектами, вы можете перебрать элементы списка нескольких функций. Например, такой перебор может использоваться для вывода графиков нескольких функций на одной диаграмме:

```
> g1 <- function(x) return(sin(x))
> g2 <- function(x) return(sqrt(x^2+1))
> g3 <- function(x) return(2*x-1)
> plot(c(0,1),c(-1,1.5)) # Подготовить график, задать диапазоны X и Y ranges
> for (f in c(g1,g2,g3)) plot(f,0,1,add=T) # Добавить график на диаграмму
```

Функции `formals()` и `body()` даже могут использоваться в качестве функций замены. Функции замены будут рассматриваться в разделе 7.10, а пока посмотрим, как изменить тело функции присваиванием:

```
> g <- function(h,a,b) h(a,b)
> body(g) <- quote(2*x+3)
> g
function (x)
2 * x+3
> x <- 3
> g(3)
[1] 9
```

Необходимость `quote()` объясняется тем, что с технической точки зрения тело функции имеет класс `"call"`, создаваемый вызовом `quote()`. Без вызова `quote()` R попытается вычислить величину `2*x+3`. Итак, если для `x` определено значение 3, телу `g()` будет присвоено значение 9 — конечно, это не то, что нужно. Кстати говоря, поскольку `*` и `+` являются функциями (см. раздел 2.4.1), `2*x+3` как языковой объект действительно является вызовом (call) — собственно, это один вызов функции, вложенный в другой вызов.

7.6. Окружение и проблемы видимости

Функция — формально называемая *замыканием* (closure) в документации R — состоит не только из аргументов и тела, но и из *окружения* (environment). Окружение формируется из объектов, присутствующих на момент создания функции. Чтобы писать эффективные функции R, необходимо понимать, как работает окружение в R.

7.6.1. Окружение верхнего уровня

Пример:

```
> w<-12
> f <- function(y) {
+    d <- 8
+    h <- function() {
+       return(d*(w+y))
+    }
+    return(h())
+ }
> environment(f)
<environment: R_GlobalEnv>
```

Здесь функция `f()` создается на *верхнем уровне*, то есть на уровне приглашения командного интерпретатора, а следовательно, имеет окружение верхнего уровня, которое в выводе R обозначается `R_GlobalEnv`; при этом в коде R используется обозначение `.GlobalEnv`, что создает дополнительную путаницу. Если запустить программу R в пакетном файле, это также будет считаться верхним уровнем.

Функция `ls()` выводит список объектов в окружении. Если вызвать ее на верхнем уровне, вы получите окружение верхнего уровня. Опробуем ее с кодом нашего примера:

```
> ls()
[1] "f" "w"
```

Как видите, окружение верхнего уровня здесь включает переменную `w`, которая в действительности используется внутри `f()`. Обратите внимание: `f()` тоже входит в список, так как функции действительно являются объектами, и эта функция была создана на верхнем уровне. На всех уровнях, кроме верхнего, `ls()` работает немного иначе, как будет показано в разделе 7.6.3.

Чуть более подробную информацию можно получить от `ls.str()`:

```
> ls.str()
f : function (y)
w : num 12
```

Теперь посмотрим, как `w` и другие переменные начинают действовать в `f()`.

7.6.2. Иерархия видимости

Для начала попробуем составить интуитивное представление о том, как работают области видимости в R, а затем свяжем их с окружениями.

Если вы работали с языком C (хотя, как обычно, опыт работы на C для понимания материала не обязателен), можно сказать, что переменная `w` из предыдущего раздела глобальна для `f()`, тогда как переменная `d` локальна для `f()`. В R ситуация обстоит аналогично, но структура R обладает более сложной иерархией. В C функции не могут определяться внутри функций, как в случае с `h()` внутри `f()` в нашем примере. Тем не менее, поскольку функции являются объектами, возможно — а иногда даже желательно с позиций инкапсуляции как одной из целей объектно-ориентированного программирования — определить функцию внутри функции; при этом вы просто создаете объект, что можно сделать где угодно.

Здесь функция `h()` локальна по отношению к `f()`, как и переменная `d`. В таких ситуациях иерархия областей видимости выглядит естественно. Таким образом, в R переменная `d`, локальная для `f()`, в свою очередь, является глобальной для `h()`. То же можно сказать об `y`, так как аргументы в R считаются локальными переменными.

Аналогичным образом из иерархической природы областей видимости следует, что переменная `w`, глобальная для `f()`, также является глобальной по отношению к `h()`. Собственно, в своем примере мы используем `w` внутри `h()`.

Итак, в контексте окружений окружение `h()` состоит из объектов, определенных на момент создания `h()`, то есть на момент выполнения следующего присваивания:

```
h <- function() {
   return(d*(w+y))
}
```

(Если `f()` вызывается многократно, `h()` будет создаваться многократно и уходить в небытие каждый раз, когда `f()` возвращает управление.)

Что же тогда будет в окружении `h()`? На момент создания `h()` в `f()` уже будут созданы объекты `d` и `y`, а также окружение `f()` (`w`). Другими словами, если одна функция определяется внутри другой, то окружение внутренней функции состоит из окружения внешней функции и локальных переменных, созданных на данный момент внутри внешней функции. При многоуровневом вложении функций существует последовательность разрастающихся окружений, а «корневой» уровень состоит из объектов верхнего уровня.

Опробуем следующий код:

```
> f(2)
[1] 112
```

Что произошло? В результате вызова `f(2)` локальной переменной `d` было присвоено значение 8, после чего следует вызов `h()`. Последнее вычисленное значение `d*(w+y)`, то есть `8*(12+2)`, равно 112.

Внимательно обдумайте роль `w`. Интерпретатор R обнаруживает, что локальной переменной с таким именем не существует, и поэтому поднимается на следующий уровень (в данном случае это верхний уровень), где обнаруживает переменную `w` со значением 12. Следует помнить, что функция `h()` является локальной для `f()`, и на верхнем уровне она не видна.

```
> h
Error: object 'h' not found
```

Возможно (хотя и нежелательно) намеренно разрешить конфликты имен в этой иерархии. Скажем, в нашем примере в `h()` может существовать локальная переменная `d`, конфликтующая с одноименной переменной `f()`. В такой ситуации первым используется внутреннее окружение. В таком случае ссылка на `d` внутри `h()` будет относиться к переменной `d` из функции `h()`, а не из функции `f()`.

Подобные окружения, созданные в результате наследования, обычно обозначаются своими адресами в памяти. Вот что произойдет после добавления в `f()` команды `print` (с вызовом `edit()`, здесь не показанным) и последующим выполнением кода:

```
> f
function(y) {
   d <- 8
   h <- function() {
      return(d*(w+y))
   }
   print(environment(h))
   return(h())
}
> f(2)
<environment: 0x875753c>
[1] 112
```

Сравните с ситуацией без вложения функций:

```
> f
function(y) {
   d<-8
   return(h())
}

> h
```

```
function() {
   return(d*(w+y))
}
```

Результат выглядит так:

```
> f(5)
Error in h() : object 'd' not found
```

Код не работает, так как `d` уже не находится в окружении `h()`, ведь `h()` определяется на верхнем уровне. Выдается сообщение об ошибке.

Хуже, если в окружении верхнего уровня случайно окажется переменная `d`, которая не имеет отношения к делу; сообщения об ошибке не будет, а вы получите неправильные результаты.

Почему же R не сообщает об отсутствии `y` в альтернативном определении `h()` в предыдущем примере? Как упоминалось ранее, R не вычисляет переменную до момента ее фактического использования — эта политика называется *отложенным вычислением*. В этом случае R уже успевает обнаружить ошибку с `d` и не доходит до точки, в которой попытается вычислить `y`.

Проблема решается передачей `d` и `y` в аргументах:

```
> f
function(y) {
   d <- 8
   return(h(d,y))
}
> h
function(dee,yyy) {
   return(dee*(w+yyy))
}
> f(2)
[1] 112
```

А теперь последний вариант:

```
> f
function(y,ftn) {
   d <- 8
   print(environment(ftn))
   return(ftn(d,y))
}
> h
function(dee,yyy) {
   return(dee*(w+yyy))
}
```

```
> w <- 12
> f(3,h)
<environment: R_GlobalEnv>
[1] 120
```

При выполнении `f()` формальному аргументу `ftn` ставится в соответствие фактический аргумент `h`. Так как аргументы рассматриваются как локальные переменные, можно предположить, что окружение `ftn` отличается от окружения верхнего уровня. Но, как упоминалось выше, в замыкание включается окружение, а следовательно, `ftn` имеет доступ к окружению `h`.

Обратите внимание: во всех примерах до настоящего момента нелокальные переменные используются для чтения, а не для записи. Запись — особый случай, который будет рассмотрен в разделе 7.8.1.

7.6.3. Подробнее о ls()

Без аргументов вызов `ls()` внутри функции возвращает имена текущих локальных переменных (включая аргументы). С аргументом `envir` функция выведет имена локальных переменных любого кадра в цепочке вызовов.

Пример:

```
> f
function(y) {
d <- 8
   return(h(d,y))
}
> h
function(dee,yyy) {
   print(ls())
   print(ls(envir=parent.frame(n=1)))
   return(dee*(w+yyy))
}

> f(2)
[1] "dee" "yyy"
[1] "d" "y"
[1] 112
```

С `parent.frame()` аргумент `n` указывает, на сколько кадров нужно подняться вверх по цепочке вызовов. Здесь мы находимся на середине выполнения функции `h()`, которая была вызвана из `f()`, поэтому аргумент `n=1` определяет кадр `f()`. В результате мы получаем локальные переменные именно этого кадра.

7.6.4. Функции (почти) не имеют побочных эффектов

Еще один аспект влияния философии функционального программирования заключается в том, что функции не изменяют нелокальные переменные; другими словами, в общем случае они не имеют побочных эффектов. В общих чертах, в коде функции нелокальные переменные доступны для чтения, но не для записи. На первый взгляд может показаться, что ваш код присваивает им новые значения, но действие распространяется только на копии, а не на сами переменные. Чтобы продемонстрировать этот факт, дополним предыдущий пример новым кодом.

```
> w <- 12
> f
function(y) {
   d <- 8
   w <- w + 1
   y <- y - 2
   print(w)
   h <- function() {
      return(d*(w+y))
   }
   return(h())
}
> t <- 4
> f(t)
[1] 13
[1] 120
> w
[1] 12
> t
[1] 4
```

Итак, переменная `w` на верхнем уровне не изменилась, хотя она вроде бы изменялась в `f()`. Изменилась только локальная *копия* `w` внутри `f()`. Также переменная `t` на верхнем уровне осталась неизменной, хотя изменился связанный с ней формальный аргумент `y`.

ПРИМЕЧАНИЕ

Если выражаться точнее, ссылки на локальную переменную `w` на самом деле указывают на область памяти глобальной переменной, пока значение локальной переменной не изменится. В этом случае будет использована новая область памяти.

Важное исключение из правила доступности глобальных переменных только для чтения встречается с оператором суперприсваивания, который будет описан в разделе 7.8.1.

7.6.5. Расширенный пример: функция для вывода содержимого кадра вызовов

При пошаговом выполнении кода в процессе отладки часто бывает нужно узнать значения локальных переменных в текущей функции. Также иногда бывает желательно знать значения локальных переменных в родительской функции, то есть функции, из которой была вызвана текущая функция. Мы напишем код для вывода этих значений и покажем, как получить доступ к иерархии окружений. (Используется адаптированный код моей отладочной программы edtdbg из репозитория R CRAN.)

Для примера возьмем следующий код:

```
f <- function() {
   a <- 1
   return(g(a)+a)
}

g <- function(aa) {
   b <- 2
   aab <- h(aa+b)
   return(aab)
}

h <- function(aaa) {
   c <- 3
   return(aaa+c)
}
```

Функция `f()`, в свою очередь, вызывает `g()`, а `g()` вызывает `h()`. Допустим, в процессе отладки вы собираетесь выполнить `return()` внутри `g()`. Требуется узнать значения локальных переменных текущей функции — скажем, переменных `aa`, `b` и `aab`. А заодно во время нахождения в `g()` также нужно узнать значения локальных переменных из `f()` на момент вызова `g()`, а также значения глобальных переменных. Все это сделает функция `showframe()`.

Функция `showframe()` получает один аргумент `upn` с количеством кадров, на которые следует перейти по стеку вызовов. Отрицательное значение аргумента сообщает, что вы хотите просмотреть глобальные переменные (то есть переменные верхнего уровня).

```
# Выводит значения локальных переменных (включая аргументы) кадра,
# находящегося на upn кадров выше того, из которого была вызвана функция
# showframe(); если upn < 0, выводятся глобальные переменные;
# объекты функций не выводятся.
showframe <- function(upn) {
   # Определить правильное окружение
   if (upn < 0) {
      env <- .GlobalEnv
   } else {
      env <- parent.frame(n=upn+1)
   }
   # Получить список имен переменных
   vars <- ls(envir=env)
   # Для каждого имени переменной вывести ее значение
   for (vr in vars) {
      vrg <- get(vr,envir=env)
      if (!is.function(vrg)) {
         cat(vr,":\n",sep="")
         print(vrg)
      }
   }
}
```

Опробуем функцию на практике. Включите в `g()` несколько вызовов:

```
> g
function(aa) {
   b <-2
   showframe(0)
   showframe(1)
   aab <- h(aa+b)
   return(aab)
}
```

А теперь запустите:

```
> f()
aa:
[1] 1
b:
[1] 2
a:
[1] 1
```

Чтобы понять, как работает этот код, нужно сначала рассмотреть `get()` — одну из самых полезных функций R. Ее задача проста: она получает имя объекта и возвращает сам объект. Пример:

```
> m <- rbind(1:3,20:22)
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   20   21   22
> get("m")
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   20   21   22
```

В примере с `m` задействован текущий кадр вызова, но в функции `showframe()` мы имеем дело с различными уровнями иерархии окружения. А значит, необходимо задать уровень при помощи аргумента `envir` функции `get()`:

```
vrg <- get(vr,envir=env)
```

Сам уровень определяется в основном вызовом `parent.frame()`:

```
if (upn < 0) {
   env <- .GlobalEnv
} else {
   env <- parent.frame(n=upn+1)
}
```

Обратите внимание: `ls()` также может вызываться в контексте конкретного уровня, что позволяет определить, какие переменные существуют на интересующем вас уровне, и проанализировать их. Пример:

```
vars <- ls(envir=env)
for (vr in vars) {
```

Этот код получает имена всех локальных переменных в заданном кадре и перебирает их, подготавливая обстановку для функции `get()`.

7.7. В языке R нет указателей

В языке R нет переменных, которые служили бы аналогами для указателей или ссылок в других языках, например в C. Это может усложнить программирование в некоторых ситуациях. (На момент написания книги в текущей версии R была реализована экспериментальная поддержка *ссылочных классов*, которые помогут преодолеть это затруднение.)

Например, вы не сможете написать функцию, которая напрямую изменяет свои аргументы. В Python можно выполнить следующие команды:

```
>>> x = [13,5,12]
>>> x.sort()
>>> x
[5, 12, 13]
```

В этом примере значение x (аргумента sort()) изменяется. С другой стороны, в R происходит нечто иное:

```
> x <- c(13,5,12)
> sort(x)
[1] 51213
> x
[1] 13 5 12
```

Аргумент sort() не изменяется. Если вы хотите, чтобы значение x изменилось в этом коде R, используйте повторное присваивание аргумента:

```
> x <- sort(x)
> x
[1] 5 12 13
```

А если функция выводит несколько переменных? Объедините их в список, вызовите функцию с передачей этого списка в аргументе, верните список из функции, а затем присвойте его исходному списку. Примером служит следующая функция, определяющая индексы нечетных и четных чисел в целочисленном векторе:

```
> oddsevens
function(v){
   odds <- which(v %% 2 == 1)
   evens <- which(v %% 2 == 0)
   list(o=odds,e=evens)
}
```

В общем случае функция f() изменяет переменные x и y. Вы можете сохранить их в списке lxy, который станет аргументом f(). Код (вызывающий и вызываемый) может строиться по следующей схеме:

```
f <- function(lxxyy) {
   ...
   lxxyy$x <- ...
   lxxyy$y <- ...
   return(lxxyy)
}
# Задать x и y
lxy$x <- ...
lxy$y <- ...
```

```
lxy <- f(lxy)
# Использовать новые значения x и y
... <- lxy$x
... <- lxy$y
```

Тем не менее, если ваша функция изменяет много переменных, решение быстро выходит из-под контроля. Оно получится особенно громоздким, если переменные (такие, как x и y в нашем примере) сами являются списками и возвращаемое решение состоит из списков внутри списков. Проблема разрешима, но код получается более сложным с точки зрения синтаксиса и хуже читается.

Альтернативные решения основаны на использовании глобальных переменных (раздел 7.8.4) и новых ссылочных классов R, о которых упоминалось выше.

Отсутствие указателей также создает сложности при работе с древовидными структурами данных. В коде C для реализации таких структур обычно широко применяются указатели. Одно решение для R основано на возвращении к тому, что делалось в «старые добрые времена» до прихода C, когда программисты формировали собственные «указатели» как индексы векторов. Пример приведен в разделе 7.9.2.

7.8. Восходящая запись

Как упоминалось ранее, код, существующий на определенном уровне иерархии окружений, обладает как минимум доступом для чтения к переменным более высоких уровней. С другой стороны, напрямую обратиться для записи к переменным более высоких уровней с использованием стандартного оператора `<-` не удастся.

Если вы хотите выполнить запись в глобальную переменную — или в более широком смысле в любую переменную, находящуюся в иерархии окружений выше команды записи, — можно воспользоваться оператором суперприсваивания `<<-` или функцией `assign()`. Начнем с оператора суперприсваивания.

7.8.1. Запись в нелокальные переменные с использованием оператора суперприсваивания

Рассмотрим следующий код:

```
> two <- function(u) {
+     u <<- 2*u
+     z <- 2*z
```

```
+ }
> x <- 1
> z <- 3
> u
Error: object "u" not found
> two(x)
> x
[1] 1
> z
[1] 3
> u
[1] 2
```

Посмотрим, как этот код влияет (или не влияет) на три переменные верхнего уровня — `x`, `z` и `u`:

- `x`: несмотря на то, что переменная `x` была фактическим аргументом `two()` в данном примере, после вызова она сохранила значение 1. Это происходит из-за того, что значение 1 было скопировано в формальный аргумент `u`, который интерпретируется как локальная переменная внутри функции. При последующем изменении переменной `u` переменная `x` не изменялась вместе с ней.
- `z`: два значения `z` совершенно не связаны друг с другом — одно находится на верхнем уровне, другое локально по отношению к `two()`. Изменение локальной переменной не отражается на глобальной переменной. Конечно, наличие двух одноименных переменных вряд ли можно назвать хорошим стилем программирования.
- `u`: до вызова `two()` значение `u` не существовало как переменная верхнего уровня, отсюда и сообщение об ошибке. Тем не менее она была создана как переменная верхнего уровня оператором суперприсваивания внутри `two()`, что подтверждается после вызова.

Хотя оператор `<<-` обычно используется для записи в переменные верхнего уровня, как в нашем примере, формально она делает нечто иное. Использование этого оператора для записи в переменную `w` запускает восходящий поиск по иерархии окружений с остановкой на первом уровне, на котором было обнаружено имя переменной. Если переменная не найдена, выбирается глобальный уровень. Посмотрите, что происходит в следующем маленьком примере:

```
> f
function() {
   inc <- function() {x <<- x + 1}
   x <- 3
```

```
   inc()
   return(x)
}
> f()
[1] 4
> x
Error: object 'x' not found
```

Здесь `inc()` определяется в `f()`. При выполнении `inc()` интерпретатор R обнаруживает суперприсваивание `x` и начинает восхождение по иерархии. На первом же верхнем уровне — окружении `f()` — он находит переменную `x`, и запись будет выполнена именно в эту переменную `x`, а не в `x` верхнего уровня.

7.8.2. Запись в нелокальные переменные с использованием assign()

Функция `assign()` также может использоваться для записи в переменные верхнего уровня. Перед вами модифицированная версия предыдущего примера:

```
> two
function(u) {
   assign("u",2*u,pos=.GlobalEnv)
   z <- 2*z
}
> two(x)
> x
[1] 1
> u
[1] 2
```

Здесь оператор суперприсваивания заменяется вызовом `assign()`. Этот вызов приказывает R присвоить значение `2*u` (локальная переменная `u`) переменной `u` выше по стеку вызовов, а именно из окружения верхнего уровня. В данном случае это окружение находится всего одним уровнем выше, но если бы мы использовали цепочку вызовов, то оно могло бы оказаться намного выше.

Факт обращения к переменным по символьным строкам в `assign()` может пригодиться. Вспомните пример из главы 5 с анализом закономерностей найма работников разными крупными корпорациями. Мы хотим сформировать для каждой фирмы подкадр данных, выделенный из общего кадра данных `all2006`. Возьмем следующий вызов:

```
makecorpdfs(c("MICROSOFT CORPORATION","ms","INTEL CORPORATION","intel","
SUN MICROSYSTEMS, INC.","sun","GOOGLE INC.","google"))
```

Эта команда сначала извлекает все записи Microsoft из общего кадра данных и присваивает полученному подкадру данных имя `ms2006`. После этого будет создан подкадр `intel2006` для Intel и т. д. Вот как выглядит код (для наглядности преобразованный в функциональную форму):

```
makecorpdfs <- function(corplist) {
   for (i in 1:(length(corplist)/2)) {
      corp <- corplist[2*i-1]
      newdtf <- paste(corplist[2*i],"2006",sep="")
      assign(newdtf,makecorp(corp),pos=.GlobalEnv)
   }
}
```

Во время итерации `i=1` код использует функцию `paste()` для объединения строк `"ms"` и `"2006"`; в результате будет создано нужное имя `"ms2006"`.

7.8.3. Расширенный пример: дискретно-событийное моделирование в R

Дискретно-событийное моделирование (DES) широко применяется в бизнесе, промышленности и государственном управлении. Термином «дискретное событие» обозначается тот факт, что состояние системы изменяется не непрерывно, а с дискретными приращениями.

Типичный пример — система очередей. Допустим, люди встают в очередь к банкомату. Определим состояние системы в момент `t` как количество людей в очереди на этот момент времени. Состояние изменяется только с приращениями +1, когда в очередь встает новый клиент, или –1, если клиент завершает операцию с банкоматом. В этом отношении модель отличается, скажем, от моделирования погоды, при котором температура, атмосферное давление и т. д. изменяются непрерывно.

В этом разделе рассматривается один из самых длинных и сложных примеров в книге. Тем не менее он представляет ряд важных аспектов R, особенно относящихся к глобальным переменным, и служит примером при обсуждении правильного использования глобальных переменных в следующем разделе. Будьте терпеливы, и вы не пожалеете о потраченном времени. (Пример не требует от читателя опыта использования DES.)

В работе DES центральное место занимает ведение *списка событий*, который представляет собой список запланированных событий. Это общий термин DES, поэтому список слов в этом примере не имеет отношения к конкретному типу

данных R. Собственно, для представления списка событий будет использоваться кадр данных.

В примере с банкоматом список событий в какой-то момент моделирования может выглядеть так:

```
customer 1 arrives at time 23.12
customer 2 arrives at time 25.88
customer 3 arrives at time 25.97
customer 1 finishes service at time 26.02
```

Так как следующим всегда обрабатывается самое раннее событие, в простейшей форме кодирования списка события хранятся в хронологическом порядке, как в примере. (Читатели с подготовкой в области теории данных заметят, что более эффективный подход может использовать некоторую разновидность бинарного дерева для хранения данных.) Здесь мы реализуем список в форме кадра данных: первая строка содержит самое раннее запланированное событие, вторая строка — второе и т. д.

Главный цикл моделирования выполняет постоянные итерации. Каждая итерация извлекает из списка событий самое раннее событие, обновляет моделируемое время, отражая возникновение данного события, и реагирует на событие. Реакция обычно приводит к созданию нового события. Например, если клиент приходит к пустой очереди, начинается обслуживание клиента — одно событие инициирует создание другого. Наш код должен определить время обслуживания; тогда он будет знать время завершения обслуживания — еще одного события, которое должно быть добавлено в список событий. Один из самых старых подходов к написанию кода DES основан на *событийно-ориентированной парадигме*. Здесь код обработки одного события напрямую создает другое событие, как описано выше.

Чтобы понять, в каком направлении следует мыслить, рассмотрим ситуацию с банкоматом. В момент времени 0 очередь пуста. Код моделирования генерирует случайное время первого поступления — скажем, 2,3. На этот момент список событий содержит только один элемент (`2.3, "arrival"`). Событие извлекается из списка, моделируемое время обновляется значением 2,3, а поступление обрабатывается следующим образом:

- Очередь банкомата пуста, поэтому обслуживание начинается случайным генерированием времени обслуживания — скажем, 1,2 временных единицы. Обслуживание будет завершено в моделируемое время 2,3 + 1,2 = 3,5.
- В список событий добавляется событие завершения обслуживания (`3.5,"service done"`).

- Также генерируется время следующего поступления — скажем, 0,6, то есть клиент придет во время 2,9. Теперь список событий состоит из элементов `(2.9,"arrival")` и `(3.5,"service done")`.

Код оформлен в виде библиотеки общего назначения. Также приводится код примера, который моделирует очередь M/M/1 (очередь с одним сервером и экспоненциальным распределением как времени между событиями, так и времени обслуживания).

ПРИМЕЧАНИЕ

Код примера вряд ли можно назвать оптимальным, и при желании читатели могут доработать его, особенно переписав некоторые части на C. (В главе 15 показано, как организовать взаимодействие C с R.) Тем не менее пример демонстрирует ряд важных аспектов, которые обсуждались в этой главе.

Сводка функций библиотеки:

- `schedevnt()` — вставляет созданное событие в список событий.
- `getnextevnt()` — извлекает самое раннее событие из списка событий.
- `dosim()` — включает основной цикл моделирования. Многократно вызывает `getnextevnt()` для получения самого раннего из ожидающих событий; обновляет текущее время модели `sim$currtime`, чтобы отразить возникновение события, и вызывает функцию `reactevnt()` для обработки произошедшего события.

В коде используются следующие функции, связанные с конкретным применением:

- `initglbls()` — инициализирует глобальные переменные для конкретного применения.
- `reactevnt()` — выполняет необходимые действия при возникновении события, в результате чего обычно генерируются новые события.
- `prntrslts()` — выводит результаты моделирования для конкретного применения.

Функции `initglbls()`, `reactevnt()` и `prntrslts()` пишутся прикладным программистом и передаются `dosim()` в аргументах. В приведенном примере с очередью M/M/1 этим функциям присвоены имена `mm1initglbls()`, `mm1reactevnt()` и `mm1prntrslts()` соответственно. Таким образом, в соответствии с определением `dosim()`,

```
dosim <- function(initglbls,reactevnt,prntrslts,maxsimtime,apppars=NULL,
         dbg=FALSE){
```

наш вызов выглядит так:

```
dosim(mm1initglbls,mm1reactevnt,mm1prntrslts,10000.0,
list(arrvrate=0.5,srvrate=1.0))
```

А теперь код библиотеки:

```
 1 # DES.R: функции R для дискретно-событийного моделирования (DES)
 2
 3 # Каждое событие представляется строкой кадра данных, состоящей
 4 # из следующих компонентов: evnttime, время предстоящего события;
 5 # evnttype, строка с типом события, определяемого программистом;
 6 # необязательные компоненты для конкретного применения (например,
 7 # время поступления задания в системе планирования)
 8
 9 # Глобальный список "sim" содержит кадр данных событий evnts
10 # и текущее время модели currtime; также имеется компонент dbg,
11 # признак режима отладки.
12
13 # Формирует строку для события типа evntty, происходящего во время
14 # evnttm; относительно appin см. комментарии к schedevnt()
15 evntrow <- function(evnttm,evntty,appin=NULL) {
16    rw <- c(list(evnttime=evnttm,evnttype=evntty),appin)
17    return(as.data.frame(rw))
18 }
19
20 # Вставляет в список событие с временем evnttm и типом evntty;
21 # appin – необязательный набор характеристик события для конкретного,
22 # применения, заданный в форме списка с именованными компонентами.
23 schedevnt <- function(evnttm,evntty,appin=NULL) {
24    newevnt <- evntrow(evnttm,evntty,appin)
25    # Если список событий пуст, присвоить ему evnt и вернуть управление
26    if (is.null(sim$evnts)) {
27       sim$evnts <<- newevnt
28       return()
29    }
30    # В противном случае найти позицию вставки
31    inspt <- binsearch((sim$evnts)$evnttime,evnttm)
32    # Выполнить "вставку" повторным построением кадра данных; мы находим,
33    # какие части текущей матрицы предшествуют новому событию и следуют
34    # после него, а затем объединяем все вместе.
35    before <-
36       if (inspt == 1) NULL else sim$evnts[1:(inspt-1),]
37    nr <- nrow(sim$evnts)
38    after <- if (inspt <= nr) sim$evnts[inspt:nr,] else NULL
```

```
39    sim$evnts <<- rbind(before,newevnt,after)
40 }
41
42 # Бинарный поиск позиции вставки y в отсортированном векторе x;
43 # возвращает позицию x, перед которой должно вставляться значение y,
44 # со значением length(x)+1, если y больше x[length(x)];
45 # код можно переписать на C для эффективности.
46 binsearch <- function(x,y) {
47    n <- length(x)
48    lo <- 1
49    hi <- n
50    while(lo+1 < hi) {
51       mid <- floor((lo+hi)/2)
52       if (y == x[mid]) return(mid)
53       if (y < x[mid]) hi <- mid else lo <- mid
54    }
55    if (y <= x[lo]) return(lo)
56    if (y < x[hi]) return(hi)
57    return(hi+1)
58 }
59
60 # Начинает обработку следующего события (вторая половина реализуется
61 # прикладным программистом через вызов reactevnt())
62 getnextevnt <- function() {
63    head <- sim$evnts[1,]
64    # Удалить начало
65    if (nrow(sim$evnts) == 1) {
66       sim$evnts <<- NULL
67    } else sim$evnts <<- sim$evnts[-1,]
68    return(head)
69 }
70
71 # Основной код моделирования.
72 # Аргументы:
73   # initglbls: специализированная функция инициализации; инициализирует
74   # глобальные переменные статистическими величинами и т. д.; сохраняет
75   # apppars в глобальных переменных; планирует первое событие
76   # reactevnt: функция обработки события для конкретного применения,
77   # кодирующая действия для каждого типа событий
78   # prntrslts: выводит результаты для конкретного применения (например,
79   # среднее время ожидания в очереди)
80   # apppars: список параметров для конкретного применения (например,
81   # количество серверов в приложении обслуживания очередей)
82   # maxsimtime: предельное время моделирования
83   # dbg: флаг отладки; TRUE – после каждого события выводится описание
84 dosim <- function(initglbls,reactevnt,prntrslts,maxsimtime,apppars=NULL,
85       dbg=FALSE) {
86    sim <<- list()
```

```
87    sim$currtime <<- 0.0 # Текущее время модели
88    sim$evnts <<- NULL # Кадр данных событий
89    sim$dbg <<- dbg
90    initglbls(apppars)
91    while(sim$currtime < maxsimtime) {
92       head <- getnextevnt()
93       sim$currtime <<- head$evnttime # Обновить текущее время модели
94       reactevnt(head) # Обработать событие
95       if (dbg) print(sim)
96    }
97    prntrslts()
98 }
```

А теперь рассмотрим пример практического применения кода. Напомню, что моделируется очередь M/M/1 — система очередей с одним сервером, в которой время обслуживания и время между поступлениями заданий имеют экспоненциальное распределение.

```
 1 # Применение DES: очередь M/M/1, поступление 0.5, обслуживание 1.0
 2
 3 # Вызов
 4 # dosim(mm1initglbls,mm1reactevnt,mm1prntrslts,10000.0,
 5 #    list(arrvrate=0.5,srvrate=1.0))
 6 # должен вернуть значение около 2 (может потребовать времени)
 7
 8 # Инициализирует глобальные переменные для конкретного применения
 9 mm1initglbls <- function(apppars) {
10    mm1glbls <<- list()
11    # Параметры модели
12    mm1glbls$arrvrate <<- apppars$arrvrate
13    mm1glbls$srvrate <<- apppars$srvrate
14    # Очередь сервера с временем поступления заданий в очередь
15    mm1glbls$srvq <<- vector(length=0)
16    # Статистика
17    mm1glbls$njobsdone <<- 0 # Количество выполненных заданий
18    mm1glbls$totwait <<- 0.0 # Накопленное время ожидания
19    # Создать первое событие; данные каждого события состоят
20    # из времени поступления, которое необходимо хранить, если
21    # в будущем потребуется вычислить время пребывания задания
22    # в системе.
23    arrvtime <- rexp(1,mm1glbls$arrvrate)
24    schedevnt(arrvtime,"arrv",list(arrvtime=arrvtime))
25 }
26
27 # Функция обработки событий, вызываемая dosim()
28 # в общей библиотеке DES
29 mm1reactevnt <- function(head) {
30    if (head$evnttype == "arrv") { # Поступление
```

```
      # Если сервер свободен, начать обслуживание; в противном случае
      # добавить в очередь (даже при пустой очереди для удобства)
      if (length(mm1glbls$srvq) == 0) {
         mm1glbls$srvq <<- head$arrvtime
         srvdonetime <- sim$currtime + rexp(1,mm1glbls$srvrate)
         schedevnt(srvdonetime,"srvdone",list(arrvtime=head$arrvtime))
      } else mm1glbls$srvq <<- c(mm1glbls$srvq,head$arrvtime)
      # Сгенерировать следующее поступление
      arrvtime <- sim$currtime + rexp(1,mm1glbls$arrvrate)
      schedevnt(arrvtime,"arrv",list(arrvtime=arrvtime))
   } else { # Обслуживание завершено
      # Обработать только что завершенное задание.
      # Обновить служебную информацию.
      mm1glbls$njobsdone <<- mm1glbls$njobsdone + 1
      mm1glbls$totwait <<-
      mm1glbls$totwait + sim$currtime - head$arrvtime
      # Удалить из очереди
      mm1glbls$srvq <<- mm1glbls$srvq[-1]
      # В очереди остались задания?
      if (length(mm1glbls$srvq) > 0) {
         # Запланировать новое обслуживание
         srvdonetime <- sim$currtime + rexp(1,mm1glbls$srvrate)
         schedevnt(srvdonetime,"srvdone",list(arrvtime=mm1glbls$srvq[1]))
      }
   }
}

mm1prntrslts <- function() {
   print("mean wait:")
   print(mm1glbls$totwait/mm1glbls$njobsdone)
}
```

Чтобы увидеть, как работает моделирование, рассмотрим код для модели M/M/1. Сначала создается глобальная переменная `mm1glbls`, которая содержит переменные, относящиеся к коду M/M/1, — например, `mm1glbls$totwait`, накопленное время ожидания для всех заданий, смоделированных до настоящего момента. Как видите, для записи в такие переменные используется оператор суперприсваивания, как в следующей команде:

```
mm1glbls$srvq <<- mm1glbls$srvq[-1]
```

В коде `mm1reactevnt()` мы сосредоточимся на части кода, в которой обрабатывается событие «обслуживание завершено».

```
} else { # Обслуживание завершено
   # Обработать только что завершенное задание.
   # Обновить служебную информацию
```

```
   mm1glbls$njobsdone <<- mm1glbls$njobsdone + 1
   mm1glbls$totwait <<-
      mm1glbls$totwait + sim$currtime - head$arrvtime
   # Удалить из очереди
   mm1glbls$srvq <<- mm1glbls$srvq[-1]
   # В очереди остались задания?
   if (length(mm1glbls$srvq) > 0) {
      # Запланировать новое обслуживание
      srvdonetime <- sim$currtime + rexp(1,mm1glbls$srvrate)
      schedevnt(srvdonetime,"srvdone",list(arrvtime=mm1glbls$srvq[1]))
   }
}
```

Сначала код обновляет служебную информацию — количество завершенных заданий и накопленное время ожидания. Затем только что завершенное задание удаляется из очереди сервера. Наконец, он проверяет, остались ли задания в очереди, и если остались, вызывает `schedevnt()`, чтобы организовать обслуживание задания в начале очереди.

Как насчет самого библиотечного кода DES? Для начала заметим, что состояние моделирования, состоящее из текущего моделируемого времени и списка событий, хранится в структуре списка R `sim`. Это делалось для того, чтобы инкапсулировать всю основную информацию в один пакет, что в R обычно означает использование списка. Список `sim` был оформлен в глобальную переменную.

Как упоминалось ранее, важнейшим аспектом написания библиотеки DES является список событий. В этом коде он реализуется в виде кадра данных `sim$evnts`. Каждая строка кадра данных соответствует одному запланированному событию и содержит время события, символьную строку с типом события (допустим, поступление события или завершение обслуживания), а также любые специализированные данные, которые пожелает добавить программист. Так как каждая строка состоит как из числовых, так и из символьных данных, выбор кадра данных для представления списка был естественным. Строки кадра данных следуют в порядке возрастания времени события, которое хранится в первом столбце.

Главный цикл моделирования содержится в функции `dosim()` кода библиотеки DES, начиная со строки 91:

```
while(sim$currtime < maxsimtime) {
   head <- getnextevnt()
   sim$currtime <<- head$evnttime # Обновить текущее время модели
   reactevnt(head) # Обработать событие
   if (dbg) print(sim)
}
```

Сначала вызывается функция `getnextevnt()` для удаления из списка событий начального элемента, то есть самого раннего события. (Обратите внимание на побочный эффект: список событий при этом изменяется.) Затем текущее смоделированное время обновляется в соответствии с запланированным временем в событии начала списка. Наконец, вызывается предоставленная программистом функция `reactevnt()` и обрабатывает событие (как видно из кода M/M/1, рассмотренного выше).

Основным потенциальным преимуществом использования кадра данных в качестве нашей структуры является то, что он позволяет поддерживать список событий в порядке возрастания по времени с помощью операции двоичного поиска по времени события:

```
inspt <- binsearch((sim$evnts)$evnttime,evnttm)
```

Здесь вновь созданное событие вставляется в список событий, а тот факт, что мы работаем с вектором, позволяет применить быстрый бинарный поиск. (Впрочем, как упоминается в комментариях к коду, эту функцию следовало бы реализовать на C для хорошего быстродействия.)

Последняя строка в `schedevnt()` — хороший пример использования `rbind()`:

```
sim$evnts <<- rbind(before,newevnt,after)
```

К этому моменту мы извлекли из списка события, время которых предшествует времени `evnt`, и сохранили их в `before`. Также был построен аналогичный набор `after` для событий со временем, более поздним по сравнению с `newevnt`. После этого функция `rbind()` объединяет наборы в правильном порядке.

7.8.4. Когда следует использовать глобальные переменные?

По поводу использования глобальных переменных в сообществе программистов нет единого мнения. Очевидно, на вопрос, вынесенный в название этого раздела, нет правильного ответа, поскольку это вопрос личных предпочтений и стиля. Тем не менее многие программисты считают, что полный запрет на глобальные переменные, за который выступают многие преподаватели программирования, был бы излишне жестким. В этом разделе мы исследуем возможную пользу глобальных переменных в контексте структур R. Термином *«глобальная переменная»* будет обозначаться любая переменная, находящаяся в иерархии окружений выше уровня кода, представляющего интерес.

Использование глобальных переменных в R более распространено, чем можно было бы ожидать. Как ни удивительно, R очень широко использует глобальные переменные в своей внутренней реализации (и в коде C, и в функциях R). Так, оператор суперприсваивания <<- используется во многих библиотечных функциях R (хотя обычно для записи в переменную, находящуюся всего одним уровнем выше в иерархии переменных). В многопоточном коде и коде графического процессора, используемом для написания быстрых программ (см. главу 16), обычно широко задействованы глобальные переменные, обеспечивающие основной механизм взаимодействия между параллельными исполнителями.

А теперь для конкретности вернемся к более раннему примеру из раздела 7.7:

```
f <- function(lxxyy) { # lxxyy – список, содержащий x и y
   ...
   lxxyy$x <- ...
   lxxyy$y <- ...
   return(lxxyy)
}
# Задать x и y
lxy$x <- ...
lxy$y <- ...
lxy <- f(lxy)
# Использовать новые x и y
... <- lxy$x
... <- lxy$y
```

Как упоминалось ранее, этот код может стать громоздким, особенно если x и y сами являются списками.

С другой стороны, взгляните на альтернативную схему с использованием глобальных переменных:

```
f <- function() {
   ...
   x <<- ...
   y <<- ...
}

# Задать x и y
x <-...
y <-...
f() # Здесь x и y изменяются
# Использовать новые x и y
... <- x
... <- y
```

Пожалуй, вторая версия намного чище, менее громоздка и не требует манипуляций со списками. Понятный код обычно создает меньше проблем в написании, отладке и сопровождении.

По этим причинам — для упрощения и снижения громоздкости кода — мы решили использовать глобальные переменные вместо возвращения списков в коде DES, приведенном ранее. Рассмотрим этот пример чуть подробнее.

Использовались две глобальные переменные (обе представляют собой списки, содержащие разную информацию): переменная `sim` связана с библиотечным кодом, а переменная `mm1glbls` связана с кодом конкретного применения M/M/1. Начнем с `sim`.

Даже программисты, сдержанно относящиеся к глобальным переменным, согласны с тем, что применение таких переменных может быть оправданно в том случае, если они действительно глобальны — в том смысле, что они широко используются в программе. Все это относится к переменной `sim` из примера DES: она используется как в коде библиотеки (в `schedevnt()`, `getnextevnt()` и `dosim()`) и в коде M/M/1 (в `mm1reactevnt()`). В этом конкретном примере последующие обращения к `sim` ограничиваются чтением, но в некоторых ситуациях возможна запись. Типичный пример такого рода — возможная реализация отмены событий. Например, такая ситуация может встретиться при моделировании принципа «более раннее из двух»: планируются два события, и когда одно из них происходит, другое должно быть отменено.

Таким образом, использование `sim` в качестве глобальной переменной выглядит оправданным. Тем не менее, если бы мы решительно отказались от использования глобальных переменных, `sim` можно было бы поместить в локальную переменную внутри `dosim()`. Эта функция будет передавать `sim` в аргументе всех функций, упоминаемых в предыдущем абзаце (`schedevnt()`, `getnextevnt()` и т. д.), и каждая из этих функций будет возвращать измененную переменную `sim`.

Например, строка 94:

```
reactevnt(head)
```

преобразуется к следующему виду:

```
sim <- reactevnt(head)
```

После этого в функцию `mm1reactevnt()`, связанную с конкретным приложением, необходимо добавить следующую строку:

```
return(sim)
```

Нечто похожее можно сделать и с `mm1glbls`, включив в `dosim()` локальную переменную с именем (например) `appvars`. Но если это делается с двумя переменными, то их необходимо поместить в список, чтобы обе переменные можно было вернуть из функции, как в приведенном выше примере функции `f()`. И тогда возникает громоздкая структура списков внутри списков, о которой говорилось выше, а вернее, списков внутри списков внутри списков.

С другой стороны, противники использования глобальных переменных замечают, что простота кода не дается даром. Их беспокоит то, что в процессе отладки возникают трудности с поиском мест, в которых глобальная переменная изменяет значение, поскольку изменение может произойти в любой точке программы. Казалось бы, в мире современных текстовых редакторов и интегрированных средств разработки, которые помогут найти все вхождения переменной, проблема уходит на второй план (исходная статья, призывающая отказаться от использования глобальных переменных, была опубликована в 1970 году!). И все же этот фактор необходимо учитывать.

Другая проблема, о которой упоминают критики, встречается при вызове функции из нескольких несвязанных частей программы с разными значениями. Например, представьте, что функция `f()` вызывается из разных частей программы, причем каждый вызов получает собственные значения `x` и `y` вместо одного значения для каждого. Проблему можно решить созданием векторов значений `x` и `y`, в которых каждому экземпляру `f()` в вашей программе соответствует отдельный элемент. Однако при этом будет утеряна простота от использования глобальных переменных.

Указанные проблемы встречаются не только в R, но и в более общем контексте. Впрочем, в R использование глобальных переменных на верхнем уровне создает дополнительную проблему, так как у пользователя на этом уровне обычно существует множество переменных. Появляется опасность того, что код, использующий глобальные переменные, может случайно заменить совершенно постороннюю переменную с тем же именем.

Конечно, проблема легко решается — достаточно выбирать для глобальных переменных длинные имена, привязанные к конкретному применению. Однако окружения также предоставляют разумный компромисс, как в следующей ситуации для примера DES.

Внутри функции `dosim()` строка

```
sim <<- list()
```

может быть заменена строкой

```
assign("simenv",new.env(),envir=.GlobalEnv)
```

Она создает новое окружение, на которое ссылается переменная `simenv` на верхнем уровне. Это окружение служит контейнером для инкапсуляции глобальных переменных, к которым можно обращаться вызовами `get()` и `assign()`. Например, строки

```
if (is.null(sim$evnts)) {
   sim$evnts <<- newevnt
```

в `schedevnt()` принимают вид

```
if (is.null(get("evnts",envir=simenv))) {
   assign("evnts",newevnt,envir=simenv)
```

Да, это решение тоже получается громоздким, но по крайней мере оно не такое сложное, как списки внутри списков внутри списков. И оно защищает от случайной записи в постороннюю переменную на верхнем уровне. Использование оператора суперприсваивания по-прежнему дает менее громоздкий код, но этот компромисс следует принять во внимание.

Как обычно, не существует единого стиля программирования, который бы обеспечивал лучшие результаты во всех ситуациях. Решение с глобальными переменными — еще один вариант, который стоит включить в ваш арсенал средств программирования.

7.8.5. Замыкания

Напомню, что *замыкания* (closure) R состоят из аргументов и тела функции в совокупности с окружением на момент вызова. Факт включения окружения задействован в парадигме программирования, которая использует концепцию, также называемую замыканием (здесь наблюдается некоторая перегрузка терминологии).

Замыкание представляет собой функцию, которая создает локальную переменную, а затем создает другую функцию, обращающуюся к этой переменной. Описание получается слишком абстрактным, поэтому я лучше приведу пример[1].

```
1 > counter
2 function () {
3    ctr <- 0
4    f <- function() {
5       ctr <<- ctr + 1
```

[1] Адаптированная версия кода из статьи «Top-level Task Callbacks in R» Дункана Темпла Лэнга (Duncan Temple Lang) (2001), http://developer.r-project.org/TaskHandlers.pdf.

```
6       cat("this count currently has value",ctr,"\n")
7    }
8    return(f)
9 }
```

Проверим, как работает этот код, прежде чем погружаться в подробности реализации:

```
> c1 <- counter()
> c2 <- counter()
> c1
function() {
      ctr <<- ctr + 1
      cat("this count currently has value",ctr,"\n")
   }
<environment: 0x8d445c0>
> c2
function() {
      ctr <<- ctr + 1
      cat("this count currently has value",ctr,"\n")
   }
<environment: 0x8d447d4>
> c1()
this count currently has value 1
> c1()
this count currently has value 2
> c2()
this count currently has value 1
> c2()
this count currently has value 2
> c2()
this count currently has value 3
> c1()
this count currently has value 3
```

Здесь функция `counter()` вызывается дважды, а результаты присваиваются `c1` и `c2`. Как и ожидалось, эти две переменные состоят из функций, а именно копий `f()`. Тем не менее `f()` обращается к переменной `ctr` через оператор суперприсваивания, и эта переменная будет переменной с указанным именем, локальной по отношению к `counter()`, так как она будет первой на пути по иерархии окружений. Она является частью окружения `f()`, и как таковая упаковывается в то, что возвращается на сторону вызова `counter()`. Ключевой момент заключается в том, что при разных вызовах `counter()` переменная `ctr` будет находиться в разных окружениях (в примере окружения хранились в памяти по адресам 0x8d445c0 и 0x8d447d4). Иначе говоря, разные вызовы `counter()` будут создавать физически разные экземпляры `ctr`.

В результате функции `c1()` и `c2()` работают как полностью независимые счетчики. Это видно из примера, где каждая функция вызывается по несколько раз.

7.9. Рекурсия

Однажды ко мне обратился за советом аспирант, весьма сообразительный, но не обладавший особыми знаниями в области программирования. Его интересовало, как написать некую функцию. Я быстро ответил: «Вам даже не нужно говорить, что должна делать функция. Ответ: используйте рекурсию». Он встревожился и спросил, что такое рекурсия. Я порекомендовал ему почитать о знаменитой задаче «Ханойские башни». И разумеется, на следующий день он вернулся и сказал, что ему удалось решить свою задачу в нескольких строках кода с применением рекурсии. Очевидно, рекурсия может стать очень мощным инструментом. Что же это такое?

Рекурсивная функция вызывает сама себя. Если вы еще не сталкивались с этой концепцией, может показаться довольно странным, но концепция на самом деле проста. В несколько упрощенном виде идея выглядит так:

Чтобы решить задачу типа X при помощи рекурсивной функции `f()`:

1. Разбейте исходную задачу типа X на одну или несколько меньших задач типа `X`.
2. Внутри `f()` вызовите `f()` для каждой из меньших задач.
3. Внутри `f()` объедините результаты (2) для решения исходной задачи.

7.9.1. Реализация быстрой сортировки

Классический пример рекурсии — алгоритм быстрой сортировки, используемый для сортировки вектора чисел по возрастанию. Предположим, вы хотите отсортировать вектор (5, 4, 12, 13, 3, 8, 88). Сначала все элементы сравниваются с первым (5) и формируются два подвектора: один содержит элементы меньше 5, а другой содержит элементы больше или равные 5. Мы получаем подвектор (4, 3) и (12, 13, 8, 88). Затем функция вызывается для каждого из подвекторов, получаем подвектор (3, 4) и (8, 12, 13, 88). Результаты объединяются с промежуточным элементом 5, и мы получаем (3, 4, 5, 8, 12, 13, 88), как и требовалось.

Благодаря средствам фильтрации векторов в R и функции `c()` быстрая сортировка реализуется достаточно просто.

ПРИМЕЧАНИЕ

Этот пример предназначен исключительно для демонстрации рекурсии. Собственная функция сортировки R, sort(), работает намного быстрее, поскольку она написана на C.

```
qs <- function(x) {
   if (length(x) <= 1) return(x)
   pivot <- x[1]
   therest <- x[-1]
   sv1 <- therest[therest < pivot]
   sv2 <- therest[therest >= pivot]
   sv1 <- qs(sv1)
   sv2 <- qs(sv2)
   return(c(sv1,pivot,sv2))
}
```

Обратите внимание на *условие завершения*:

```
if (length(x) <= 1) return(x)
```

Без этого функция будет бесконечно вызывать сама себя для пустых векторов. (На самом деле интерпретатор R рано или поздно откажется продолжать выполнение, но вы меня поняли.)

Волшебство? Безусловно, рекурсия является элегантным способом решения многих задач. Тем не менее у нее есть два потенциальных недостатка:

- Она относительно абстрактна. Я знал, что аспирант, как хороший математик, будет чувствовать себя с рекурсией как рыба в воде, потому что рекурсия в действительности является обратной по отношению к методу математической индукции. Тем не менее у многих программистов могут возникнуть трудности.
- Рекурсия крайне расточительно расходует память, что может создать проблемы для R в очень больших задачах.

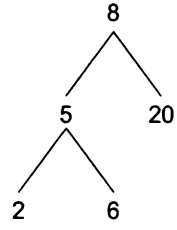
7.9.2. Расширенный пример: бинарное дерево поиска

Древовидные структуры данных широко распространены в информатике и статистике. Например, в R существует очень популярная библиотека `rpart` для рекурсивного разбиения при регрессии и классификации. Очевидно, деревья находят широкое применение в генеалогии, а в более широком смысле графы формируют основу для анализа социальных сетей.

Тем не менее при работе с деревьями в R возникают серьезные проблемы. Многие из них связаны с отсутствием ссылок-указателей в R (см. раздел 7.7). По этой причине и по соображениям быстродействия лучше писать базовый код на C с оберткой на R, как будет показано в главе 15. Тем не менее деревья можно реализовать напрямую на R, и если быстродействие не критично, этот подход может оказаться более удобным.

Для простоты в примере будет использоваться бинарное дерево поиска — структура данных из классической теории обработки данных, обладающая следующим свойством: в каждом узле дерева значение левого дочернего узла (если он есть) меньше либо равно значению родителя, а значение правого дочернего узла (если он есть) больше значения родителя.

Пример:



Значение 8 хранится в *корне* дерева. Два дочерних узла содержат 5 и 20, а первый из них сам имеет два дочерних узла со значениями 2 и 6.

Из определения бинарных деревьев поиска следует, что в любом узле все элементы левого поддерева меньше либо равны значению элемента, хранящегося в этом узле, а правое поддерево содержит элементы, превышающие значение в этом узле. В дереве из примера, где корневой узел содержит 8, все значения из левого поддерева — 5, 2 и 6 — меньше 8, а 20 больше 8.

При реализации на C узел дерева будет представлен структурой C, похожей на список R. Структура содержит хранимое значение, указатель на левый дочерний узел и указатель на правый дочерний узел. Но если в R нет указателей, как это сделать?

Решение этой проблемы возвращается к основам. В древнюю эпоху FORTRAN, когда не было указателей, связанные структуры данных реализовывались в виде длинных массивов. Указателями, которые в C представляют собой адрес памяти, были индексы массива. А конкретно каждый узел представлялся строкой матрицы из трех столбцов. Хранимое значение узла находится в третьем элементе строки, а первый и второй элементы определяют левый и правый дочерние узлы.

Например, если первый элемент строки равен 29, это означает, что левым дочерним узлом текущего узла является узел, хранящийся в 29-й строке матрицы.

Следует помнить, что выделение памяти для матриц в R является относительно долгой операцией. Для экономии времени на выделение памяти мы выделяем новую память не по одной строке, а сразу для нескольких строк матрицы дерева. Количество строк, выделяемых каждый раз, будет содержаться в переменной `inc`. Как это часто бывает при обходе деревьев, алгоритм будет реализован на базе рекурсии.

ПРИМЕЧАНИЕ

Если вы предвидите, что матрица станет достаточно большой, ее размер можно удваивать при каждом выделении памяти, вместо того чтобы наращивать его линейно, как в нашем примере. Это позволит дополнительно сократить число повторных выделений памяти.

Прежде чем обсуждать код, вкратце разберем процедуру построения дерева с использованием его функций.

```
> x <- newtree(8,3)
> x
$mat
     [,1] [,2] [,3]
[1,]   NA   NA    8
[2,]   NA   NA   NA
[3,]   NA   NA   NA

$nxt
[1] 2

$inc
[1] 3

> x <- ins(1,x,5)
> x
$mat
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]   NA   NA    5
[3,]   NA   NA   NA

$nxt
[1] 3

$inc
```

```
[1] 3

> x <- ins(1,x,6)
> x
$mat
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]   NA    3    5
[3,]   NA   NA    6

$nxt
[1] 4

$inc
[1] 3

> x <- ins(1,x,2)
> x
$mat
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]    4    3    5
[3,]   NA   NA    6
[4,]   NA   NA    2
[5,]   NA   NA   NA
[6,]   NA   NA   NA

$nxt
[1] 5

$inc
[1] 3

> x <- ins(1,x,20)
> x
$mat
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    4    3    5
[3,]   NA   NA    6
[4,]   NA   NA    2
[5,]   NA   NA   20
[6,]   NA   NA   NA

$nxt
[1] 6

$inc
[1] 3
```

Что здесь происходит? Сначала команда с вызовом `newtree(8,3)` создает новое дерево, присвоенное `x`; в нем хранится число 9. Аргумент 3 указывает, что память будет выделяться для трех строк. В результате матричный компонент списка `x` принимает следующий вид:

```
     [,1] [,2] [,3]
[1,]   NA   NA    8
[2,]   NA   NA   NA
[3,]   NA   NA   NA
```

Действительно была выделена память для трех строк, а данные в настоящий момент состоят из единственного числа 8. Два значения `NA` в первой строке показывают, что узел дерева в настоящее время не имеет дочерних узлов.

Затем вызов `ins(1,x,5)` вставляет в дерево `x` второе значение 5. Аргумент 1 задает корневой узел. Другими словами, вызов означает: «Вставить 5 в поддерево `x` с корнем в строке 1». Обратите внимание: возвращаемое значение вызова должно быть присвоено `x`. Как упоминалось выше, это связано с отсутствием полноценных указателей в R. Матрица принимает следующий вид:

```
     [,1] [,2] [,3]
[1,]    2   NA    8
[2,]   NA   NA    5
[3,]   NA   NA   NA
```

Элемент 2 означает, что левая связь узла со значением 8 указывает на строку 2, в которой хранится новый элемент 5.

Так продолжается сеанс. Обратите внимание: когда три исходные выделенные строки будут заполнены, `ins()` выделит еще три строки, итого шесть. В результате матрица будет выглядеть так:

```
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    4    3    5
[3,]   NA   NA    6
[4,]   NA   NA    2
[5,]   NA   NA   20
[6,]   NA   NA   NA
```

Матрица представляет дерево, изображенное на диаграмме для этого примера.

Ниже приведен код. Учтите, что он содержит только функции для вставки новых элементов и обхода дерева. Код удаления узлов более сложный, но построен по похожему принципу.

```
# Ниже приведены функции создания деревьев и вставки элементов;
# функция удаления предоставляется читателю для самостоятельной работы.
# Данные хранятся в матрице (допустим, m), по одной строке на узел дерева;
# если строка i состоит из элементов (u,v,w), то узел i содержит w,
# а его левым и правым дочерними узлами являются u и v;
# пустые связи представлены значением NA.
# Дерево представляется в виде списка (mat,nxt,inc), где mat – матрица,
# nxt – следующая пустая строка, а inc – число строк расширения,
# выделяемых при заполнении матрицы.

# Вывод отсортированного дерева в порядке симметричного обхода.
printtree <- function(hdidx,tr) {
   left <- tr$mat[hdidx,1]
   if (!is.na(left)) printtree(left,tr)
   print(tr$mat[hdidx,3]) # print root
   right <- tr$mat[hdidx,2]
   if (!is.na(right)) printtree(right,tr)
}

# Инициализирует матрицу и сохраняет исходное значение firstval
newtree <- function(firstval,inc) {
   m <- matrix(rep(NA,inc*3),nrow=inc,ncol=3)
   m[1,3] <- firstval
   return(list(mat=m,nxt=2,inc=inc))
}

# Вставляет newval в поддерево tr с корнем поддерева в позиции
# с индексом hdidx; учтите, что возвращаемое значение должно быть
# присвоено tr при вызове (включая саму функцию ins() из-за рекурсии)
ins <- function(hdidx,tr,newval) {
   # Направление нового узла: левое или правое?
   dir <- if (newval <= tr$mat[hdidx,3]) 1 else 2
   # Если в этом направлении находится пустая связь, поместить
   # новый узел; иначе выполнить рекурсивный вызов.
   if (is.na(tr$mat[hdidx,dir])) {
      newidx <- tr$nxt # where new node goes
      # Проверить место для добавления нового элемента.
      if (tr$nxt == nrow(tr$mat) + 1) {
         tr$mat <-
            rbind(tr$mat, matrix(rep(NA,tr$inc*3),nrow=tr$inc,ncol=3))
      }
      # Вставить новый узел дерева
      tr$mat[newidx,3] <- newval
      # Создать связь с новым узлом
      tr$mat[hdidx,dir] <- newidx
      tr$nxt <- tr$nxt + 1 # Приготовиться к следующей вставке
      return(tr)
   } else tr <- ins(tr$mat[hdidx,dir],tr,newval)
}
```

Рекурсия присутствует как в `printtree()`, так и в `ins()`. Первая функция определенно проще, поэтому начнем с нее: она выводит дерево в порядке сортировки.

Вспомните наше описание рекурсивной функции `f()` для решения задачи типа X: функция `f()` разбивает исходную задачу X на одну или несколько меньших задач X, вызывает для них `f()` и объединяет результаты. В данном случае задачей типа X является вывод дерева, которое может быть поддеревом другого, большего дерева. Функция в строке 13 предназначена для вывода указанного дерева, что она и делает, вызывая сама себя в строках 15 и 18. Здесь она сначала выводит левое поддерево, а затем правое поддерево; между ними функция выводит корень.

Этот подход — вывести левое поддерево, затем корень, затем правое поддерево — образует интуитивное представление для написания кода, но мы снова должны позаботиться о необходимом механизме завершения. Этот механизм содержится в командах `if()` в строках 15 и 18. При обнаружении пустой связи рекурсия должна быть остановлена.

Рекурсия в `ins()` следует тем же принципам, но требует существенно большей осторожности. Здесь «задачей типа X» становится вставка значения в поддерево. Мы начинаем с корня дерева и определяем, какому поддереву должно принадлежать новое значение — левому или правому (строка 33), после чего снова вызываем функцию для этого поддерева. В принципе ничего сложного нет, но нужно проследить за большим количеством подробностей, включая расширение матрицы при нехватке свободного места (строки 40–41).

Одно из различий между рекурсивным кодом `printtree()` и `ins()` заключается в том, что первая функция включает два вызова самой себя, тогда как вторая ограничивается только одним. Отсюда следует, что, возможно, вторую функцию можно без особого труда переписать в нерекурсивной форме.

7.10. Функции замены

Вспомните пример из главы 2:

```
> x <- c(1,2,4)
> names(x)
NULL
> names(x) <- c("a","b","ab")
> names(x)
[1] "a" "b" "ab"
> x
 a  b ab
 1  2  4
```

Обратите внимание на следующую строку:

```
> names(x) <- c("a","b","ab")
```

Выглядит безобидно, не правда ли? Вообще-то нет. Более того, это совершенно возмутительно! Как можно присвоить значение результату вызова функции? Это странное состояние дел объясняется концепцией *функций замены* в R. Приведенная строка кода R в действительности является результатом выполнения следующей команды:

```
x <- "names<-"(x,value=c("a","b","ab"))
```

Нет, это не опечатка. Здесь действительно вызывается функция с именем `names<-()`. (Кавычки необходимы из-за присутствия специальных символов.)

7.10.1. Что считается функцией замены?

Любой оператор присваивания, у которого левая часть содержит не только идентификатор (то есть имя переменной), считается функцией замены. Встретив следующую команду:

```
g(u) <- v
```

R попытается выполнить такую конструкцию:

```
u <- "g<-"(u,value=v)
```

Обратите внимание на слово «попытается»: попытка завершится неудачей, если вы не определили ранее `g<-()`. Учтите, что функция замены имеет один дополнительный аргумент по сравнению с исходной функцией `g()` (именованный аргумент `value`) по причинам, рассмотренным в этом разделе.

В предыдущих главах вы неоднократно видели эту команду, которая выглядела вполне безобидно:

```
x[3] <- 8
```

В левой части стоит не имя переменной; следовательно, это должна быть функция замены — и это действительно так.

Операции индексирования являются функциями. Функция `"["()` предназначена для чтения элементов вектора, а функция `"[<-"()` предназначена для записи. Пример:

```
> x <- c(8,88,5,12,13)
> x
[1] 8 88 5 12 13
> x[3]
[1] 5

> "["(x,3)
[1] 5
> x <- "[<-"(x,2:3,value=99:100)
> x
[1]   8  99 100  12  13
```

А сложный вызов в следующей строке:

```
> x <- "[<-"(x,2:3,value=99:100)
```

просто делает то, что происходит незаметно для вас при выполнении следующей команды:

```
x[2:3] <- 99:100
```

Вы можете легко проверить результат:

```
> x <- c(8,88,5,12,13)
> x[2:3] <- 99:100
> x
[1]   8  99 100  12  13
```

7.10.2. Расширенный пример: класс вектора с хранением служебной информации

Предположим, вам потребовалось отслеживать операции записи в вектор. Иначе говоря, при выполнении команды:

```
x[2] <- 8
```

нужно не только заменить значение в `x[2]` на 8, но и увеличить счетчик операций записи в `x[2]`. Для этого можно написать специализированные версии обобщенных функций замены для индексирования вектора.

ПРИМЕЧАНИЕ

В этом коде используются классы, которые будут подробно рассмотрены в главе 9. А пока вам достаточно знать, что для того, чтобы создать класс S3, нужно создать список, а затем пометить его как класс вызовом функции class().

```
1 # Класс "bookvec" для векторов с подсчетом записи в элементы.
2
3 # Каждый экземпляр класса состоит из списка, в котором хранятся
4 # значения элементов вектора, и вектора счетчиков.
5
6 # Построение нового объекта класса bookvec
7 newbookvec <- function(x) {
8    tmp <- list()
9    tmp$vec <- x # the vector itself
10    tmp$wrts <- rep(0,length(x)) # Счетчики операций записи
11    class(tmp) <- "bookvec"
12    return(tmp)
13 }
14
15 # Функция чтения
16 "[.bookvec" <- function(bv,subs) {
17    return(bv$vec[subs])
18 }
19
20 # Функция записи
21 "[<-.bookvec" <- function(bv,subs,value) {
22    bv$wrts[subs] <- bv$wrts[subs] + 1 # note the recycling
23    bv$vec[subs] <- value
24    return(bv)
25 }
```

Проверим, как работает этот класс.

```
> b <- newbookvec(c(3,4,5,5,12,13))
> b
$vec
[1]  3  4  5  5 12 13

$wrts
[1] 0 0 0 0 0 0

attr(,"class")
[1] "bookvec"
> b[2]
[1] 4
> b[2] <- 88 # Проверить запись
> b[2] # Сработало?
[1] 88
> b$wrts # Счетчик увеличился?
[1] 0 1 0 0 0 0
```

Классу было присвоено имя `"bookvec"`. Таким образом, операции индексирования будут иметь вид `[.bookvec()` и `[<-.bookvec()`.

Функция `newbookvec()` (строка 7) строит класс. В ней видна структура класса: объект состоит из самого вектора, `vec` (строка 9) и вектора счетчиков (строка 10).

Обратите внимание: в строке 11 сама функция `class()` становится функцией замены!

Функции `[.bookvec()` и `[<-.bookvec()` достаточно просты. Главное — не забудьте вернуть весь объект из последней.

7.11. Средства организации кода функций

Если вы пишете короткую функцию, которая нужна только на время, есть простой и быстрый прием: написать ее «на месте», прямо в интерактивном терминальном сеансе. Пример:

```
> g <- function(x) {
+    return(x+1)
+ }
```

Очевидно, этот подход неприемлем для длинных и более сложных функций. А теперь рассмотрим более эффективные средства организации кода R.

7.11.1. Текстовые редакторы и интегрированные среды разработки

Воспользуйтесь текстовым редактором (Vim, Emacs или даже Блокнот) или редактором интегрированной среды разработки (IDE), сохраните свой код в файле и загрузите его в R из файла. Загрузка осуществляется функцией R `source()`.

Предположим, в файле `xyz.R` содержатся функции `f()` и `g()`. Введите в R следующую команду:

```
> source("xyz.R")
```

Команда загружает `f()` и `g()` в R, как если бы они были введены простым способом, описанным в начале раздела.

Если объем кода невелик, его можно скопировать и вставить из окна редактора в окно R.

Некоторые редакторы общего назначения содержат специальные плагины для R (например, ESS для Emacs и Vim-R для Vim). Также существуют интегрированные среды (IDE) для R — например, коммерческая версия от Revolution Analytics и такие продукты с открытым кодом, как StatET, JGR, Rcmdr и RStudio.

7.11.2. Функция edit()

Функции являются объектами, и у этого факта есть одно положительное следствие: возможность редактирования функций в интерактивном режиме R. Многие программисты R пишут код в текстовом редакторе, открытом в отдельном окне, но для мелких, быстрых изменений может пригодиться функция `edit()`.

Например, функцию `f1()` можно отредактировать следующей командой:

```
> f1 <- edit(f1)
```

Команда открывает для `f1` редактор кода по умолчанию. В нем можно отредактировать код и снова присвоить его `f1`.

А если вы хотите создать функцию `f2()`, очень похожую на `f1()`, можно выполнить следующую команду:

```
> f2 <- edit(f1)
```

Команда создает копию `f1()` как отправную точку для редактирования. Вы можете слегка отредактировать код и сохранить его в `f2()`, как видно в предыдущей команде.

Используемый редактор зависит от переменной `editor` из внутренних настроек R. В системах семейства UNIX R задает значение этой переменной из переменных среды `EDITOR` и `VISUAL` оболочки вашей системы или же вы задаете ее значение самостоятельно:

```
> options(editor="/usr/bin/vim")
```

За дополнительной информацией о внутренних настройках обращайтесь к электронной документации. Для этого введите следующую команду:

```
> ?options
```

Функция `edit()` также может использоваться для редактирования структур данных.

7.12. Написание собственных бинарных операций

Вы не ограничены готовым набором операций — вы можете определять собственные операции! Напишите функцию, имя которой начинается и заканчивается символом %; функция должна получать два аргумента определенного типа и возвращать значение этого типа.

Например, следующая бинарная операция прибавляет удвоенный второй аргумент к первому:

```
> "%a2b%" <- function(a,b) return(a+2*b)
> 3 %a2b% 5
[1] 13
```

Менее тривиальный пример приводится в описании операций с множествами в разделе 8.5.

7.13. Анонимные функции

Как неоднократно отмечалось в книге, функция R `function()` предназначена для создания функций. Например, возьмем следующий код:

```
inc <- function(x) return(x+1)
```

Он приказывает R создать функцию, которая увеличивает аргумент на 1, и присваивает эту функцию `inc`. Тем не менее последний шаг — присваивание — выполняется не всегда. Можно просто использовать объект функции, созданный вызовом `function()`, без указания его имени. Функции в таком контексте называются *анонимными*, потому что им не присваивается имя. (Термин не совсем точен, потому что даже у неанонимных функций имя существует только в том смысле, что на них указывает переменная.)

Анонимные функции могут быть удобными, особенно если они состоят из одной строки и вызываются другой функцией. Вернемся к нашему примеру с использованием `apply` в разделе 3.3:

```
> z
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
> f <- function(x) x/c(2,8)
> y <- apply(z,1,f)
> y
  [,1] [,2] [,3]
[1,] 0.5 1.000 1.50
[2,] 0.5 0.625 0.75
```

Обойдем лишнего посредника, то есть присваивание `f`, используя анонимную функцию с вызовом `apply()`:

```
> y <- apply(z,1,function(x) x/c(2,8))
> y
     [,1]  [,2] [,3]
[1,]  0.5 1.000 1.50
[2,]  0.5 0.625 0.75
```

Что же здесь происходит? Третий формальный аргумент `apply()` должен быть функцией, — а именно она передается в данном примере, поскольку возвращаемое значение `function()` является функцией!

Такое определение часто получается более понятным, чем внешнее определение функций. Конечно, для более сложных функций ни о каком упрощении говорить не приходится.

8 Математические вычисления и моделирование в R

R содержит встроенные функции для основных математических операций и, конечно, для работы со статистическими распределениями. В этой главе приведен краткий обзор использования этих функций. С учетом математического содержания этой главы примеры требуют знаний чуть более высокого уровня, чем примеры в других главах. Чтобы извлечь максимум пользы из этих примеров, вы должны быть знакомы с математическим анализом и линейной алгеброй.

8.1. Математические функции

В R входит обширный набор встроенных математических функций. Ниже приведен неполный список:

- `exp()` — возведение в степень с основанием *e*;
- `log()` — натуральный логарифм;
- `log10()` — логарифм по основанию 10;
- `sqrt()` — квадратный корень;
- `abs()` — абсолютное значение;
- `sin()`, `cos()` и т. д. — тригонометрические функции;
- `min()` и `max()` — минимальное и максимальное значение в векторе;
- `which.min()` и `which.max()` — индекс минимального и максимального элемента вектора;
- `pmin()` и `pmax()` — поэлементные минимумы и максимумы нескольких векторов;
- `sum()` и `prod()` — сумма и произведение элементов вектора;

- `cumsum()` и `cumprod()` — накапливаемая сумма и произведение элементов вектора;
- `round()`, `floor()` и `ceiling()` — округление до ближайшего целого, до ближайшего меньшего целого и до ближайшего большего целого;
- `factorial()` — вычисление факториала.

8.1.1. Расширенный пример: вычисление вероятности

В первом примере мы займемся вычислением вероятности функцией `prod()`. Предположим, имеются n независимых событий, при этом i-е событие происходит с вероятностью p_i. Какова вероятность того, что произойдет ровно одно из этих событий?

Допустим, $n = 3$, а событиям присвоены имена A, B и C. Тогда вычисления разбиваются следующим образом:

```
P(ровно одно событие) =
  P(A и не B и не C) +
  P(не A и B и не C) +
  P(не A и не B и C)
```

Вероятность P(A и не B и не C) равна $p_A(1 - p_B)(1-p_C)$ и т. д. Для обобщенного n вероятность вычисляется по следующей формуле:

$$\sum_{i=1}^{n} p_i(1-p_i)\ldots(1-p_{i-1})(1-p_{i+1})\ldots(1-p_n)$$

(здесь i-е слагаемое в сумме — вероятность того, что событие i произошло, а все остальные *не* происходят).

Код вычисления этой величины выглядит так (вероятности p_i хранятся в векторе `p`):

```
exactlyone <- function(p) {
   notp <-1-p
   tot <- 0.0
   for (i in 1:length(p))
      tot <- tot + p[i] * prod(notp[-i])
   return(tot)
}
```

Как работает этот код? Присваивание

```
notp <-1 - p
```

создает вектор всех вероятностей «невозникновения» события $1-p_j$ с использованием переработки. Выражение `prod(notp[-i])` вычисляет произведение всех элементов `notp`, кроме i-го, — в точности то, что нам нужно.

8.1.2. Накапливаемые суммы и произведения

Как упоминалось ранее, функции `cumsum()` и `cumprod()` возвращают накапливаемые суммы и произведения.

```
> x <- c(12,5,13)
> cumsum(x)
[1] 12 17 30
> cumprod(x)
[1] 12 60 780
```

В `x` сумма первого элемента равна 12, сумма первых двух элементов равна 17, а сумма первых трех — 30.

Функция `cumprod()` работает аналогично `cumsum()`, но вычисляет произведение вместо суммы.

8.1.3. Минимумы и максимумы

Между функциями `min()` и `pmin()` существуют серьезные различия. Первая просто объединяет все аргументы в один длинный вектор и возвращает минимальное значение в этом векторе. С другой стороны, если функция `pmin()` применяется к двум и более векторам, она возвращает вектор попарных минимумов.

Пример:

```
> z
     [,1] [,2]
[1,]    1    2
[2,]    5    3
[3,]    6    2
> min(z[,1],z[,2])
[1] 1
> pmin(z[,1],z[,2])
[1] 1 3 2
```

В первом случае `min()` вычисляет наименьшее значение в (1, 5, 6, 2, 3, 2). Вызов `pmin()` вычисляет меньшее из значений 1 и 2 (1), затем меньшее из 5 и 3 (3), наконец, меньшее из 6 и 2 (2). В итоге вызов возвращает вектор (1, 3, 2).

Также pmin() можно передать более двух аргументов:

```
> pmin(z[1,],z[2,],z[3,])
[1] 1 2
```

1 в выводе является минимумом среди 1, 5 и 6; аналогичные вычисления дают второй элемент 2.

Функции max() и pmax() аналогичны min() и pmin(). Определение минимумов/максимумов функций может осуществляться вызовами nlm() и optim(). Например, найдем наименьшее значение $f(x)= x^2 - \sin(x)$:

```
> nlm(function(x) return(x^2-sin(x)),8)
$minimum
[1] -0.2324656

$estimate
[1] 0.4501831

$gradient
[1] 4.024558e-09

$code
[1] 1

$iterations
[1] 5
```

Функция находит минимум приблизительно –0,23 в точке $x = 0{,}45$. Используется метод Ньютона—Рафсона (итерационный численный метод приближенного нахождения корней) с пятью итерациями (в данном случае). Второй аргумент определяет начальное значение, которое мы задаем равным 8. (Второй аргумент был выбран достаточно произвольно, но в некоторых задачах приходится поэкспериментировать для поиска значения, обеспечивающего сходимость.)

8.1.4. Численные методы

В R также реализованы некоторые средства математического анализа, включая символическое дифференцирование и численное интегрирование, как показывает следующий пример:

```
> D(expression(exp(x^2)),"x") # Производная
exp(x^2) * (2 * x)
> integrate(function(x) x^2,0,1)
0.3333333 with absolute error < 3.7e-15
```

Здесь R выдает

$$\frac{d}{dx}e^{x^2} = 2xe^{x^2}$$

и

$$\int_0^1 x^2 dx \approx 0{,}3333333.$$

Существуют пакеты R для дифференциальных уравнений (`odesolve`), для взаимодействия R с системой символической математики Yacas (`ryacas`) и для других операций математического анализа. Эти пакеты (а также тысячи других) доступны в архиве CRAN (Comprehensive R Archive Network); см. приложение Б.

8.2. Функции статистических распределений

В R существуют функции для большинства известных статистических распределений. Название функции начинается с префикса:

- `d` для плотности или для функции распределения масс (`pmf`);
- `p` для накопительной функции распределения (`cdf`);
- `q` для квантилей;
- `r` для генерирования случайных чисел.

Оставшаяся часть имени обозначает распределение. В табл. 8.1 перечислены некоторые распространенные функции статистических распределений.

Таблица 8.1. Стандартные функции статистических распределений R

Распределение	Плотность/pmf	cdf	Квантили	Случайные числа
Нормальное	`dnorm()`	`pnorm()`	`qnorm()`	`rnorm()`
Хи-квадрат	`dchisq()`	`pchisq()`	`qchisq()`	`rchisq()`
Биномиальное	`dbinom()`	`pbinom()`	`qbinom()`	`rbinom()`

Для примера смоделируем 1000 величин с распределением хи-квадрат с двумя степенями свободы и вычислим их среднее значение:

```
> mean(rchisq(1000,df=2))
[1] 1.938179
```

Префикс r в rchisq указывает, что мы хотим генерировать случайные числа — в данном случае из распределения хи-квадрат. Как видно из примера, первый аргумент функций серии r задает количество генерируемых случайных величин.

Функции также получают аргументы, специфические для конкретных семейств распределений. В нашем примере для семейства хи-квадрат используется аргумент df, обозначающий количество степеней свободы.

ПРИМЕЧАНИЕ

За информацией об аргументах функций статистических распределений обращайтесь к электронной документации R. Например, чтобы узнать больше о функции хи-квадрат для квантилей, введите команду ?qchisq() в командной строке.

Также вычислим 95-процентиль распределения хи-квадрат с двумя степенями свободы:

```
> qchisq(0.95,2)
[1] 5.991465
```

Здесь префикс q обозначает квантиль — в данном случае 0,95, или 95-процентиль.

Первый аргумент в сериях d, p и q в действительности представляет собой вектор, чтобы мы могли вычислить плотность, функция распределения масс (pmf), функцию распределения (cdf) или функцию квантилей в нескольких точках. Вычислим 50- и 95-процентиль распределения хи-квадрат с двумя степенями свободы.

```
qchisq(c(0.5,0.95),df=2)
[1] 1.386294 5.991465
```

8.3. Сортировка

Обычная числовая сортировка вектора может быть выполнена функцией sort(), как в следующем примере:

```
> x <- c(13,5,12,5)
> sort(x)
[1] 5 5 12 13
> x
[1] 13 5 12 5
```

Обратите внимание: сам вектор x не изменился, что соответствует философии функциональных языков, принятой в R.

Если вас интересуют индексы отсортированных значений в исходном векторе, используйте функцию `order()`. Пример:

```
> order(x)
[1] 2 4 3 1
```

Это означает, что `x[2]` является наименьшим значением в `x`, `x[4]` — предпоследним по величине, `x[3]` — третьим с конца и т. д.

Для сортировки кадров данных можно воспользоваться функцией `order()` в сочетании с индексированием:

```
> y
    V1 V2
1  def  2
2   ab  5
3 zzzz  1
> r <- order(y$V2)
> r
[1] 3 1 2
> z <- y[r,]
> z
    V1 V2
3 zzzz  1
1  def  2
2   ab  5
```

Что здесь произошло? Мы вызвали `order()` для второго столбца `y` и получили вектор `r`, который сообщает, куда попадут числа при сортировке. Число 3 в этом векторе означает, что `x[3,2]` является наименьшим числом в `x[,2]`; 1 — что `x[1,2]` является предпоследним по величине; 2 — что `x[2,2]` является третьим с конца. Затем мы используем индексирование для получения кадра, отсортированного по столбцу 2, и сохраняем его в `z`.

Функция `order()` может использоваться для сортировки как символьных, так и числовых данных:

```
> d
   kids ages
1 Jack  12
2 Jill  10
3 Billy 13
> d[order(d$kids),]
   kids ages
3 Billy 13
1 Jack  12
```

```
2 Jill  10
> d[order(d$ages),]
   kids ages
2 Jill  10
1 Jack  12
3 Billy 13
```

Сопутствующая функция `rank()` сообщает ранг каждого элемента в векторе.

```
> x <- c(13,5,12,5)
> rank(x)
[1] 4.0 1.5 3.0 1.5
```

Из выходных данных видно, что 13 имеет ранг 4 в `x` (то есть является четвертым с конца). Значение 5 встречается в `x` дважды, являясь наименьшим и предпоследним, поэтому обоим вхождениям присваивается ранг 1,5. При желании можно задать другие способы обработки ситуаций с равенством.

8.4. Операции линейной алгебры с векторами и матрицами

Как вы уже видели ранее, вектор можно умножить на скаляр напрямую. Другой пример:

```
> y
[1]  1  3  4 10
> 2*y
[1]  2  6  8 20
```

Чтобы вычислить скалярное произведение двух векторов, используйте функцию `crossprod()`:

```
> crossprod(1:3,c(5,12,13))
[,1]
[1,] 68
```

Функция вычисляет $1 \cdot 5 + 2 \cdot 12 + 3 \cdot 13 = 68$.

Имя `crossprod()` выбрано неудачно, потому что функция не вычисляет векторное произведение (cross product). Функция для вычисления полноценных векторных произведений будет разработана в разделе 8.4.1.

Для выполнения умножения матриц в математическом смысле используется оператор `%*%`, а не `*`. Пример вычисления матричного произведения:

$$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}\begin{pmatrix}1 & -1\\0 & 1\end{pmatrix}=\begin{pmatrix}1 & 1\\3 & 1\end{pmatrix}.$$

Код выглядит так:

```
> a
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> b
     [,1] [,2]
[1,]    1   -1
[2,]    0    1
> a %*% b
     [,1] [,2]
[1,]    1    1
[2,]    3    1
```

Функция `solve()` решает системы линейных уравнений и даже находит обратные матрицы. Например, решим следующую систему:

$$x_1 + x_2 = 2$$
$$-x_1 + x_2 = 4$$

Матричная форма выглядит так:

$$\begin{pmatrix}1 & 1\\-1 & 1\end{pmatrix}\begin{pmatrix}x_1\\x_2\end{pmatrix}=\begin{pmatrix}2\\4\end{pmatrix}.$$

А теперь код:

```
> a <- matrix(c(1,-1,1,1),nrow=2)
> b <- c(2,4)
> solve(a,b)
[1] -1 3
> solve(a)
     [,1] [,2]
[1,]  0.5 -0.5
[2,]  0.5  0.5
```

При втором вызове `solve()` отсутствие второго аргумента означает, что мы просто хотим вычислить обратную матрицу.

В R также есть несколько других функций линейной алгебры:

- `t()` — транспонирование матрицы;
- `qr()` — QR-разложение;

- `chol()` — разложение Холецкого;
- `det()` — детерминант, или определитель;
- `eigen()` — собственные значения/собственные векторы;
- `diag()` — извлечение диагонали квадратной матрицы (полезно для получения дисперсии из ковариационной матрицы и для построения диагональной матрицы);
- `sweep()` — sweep-операции из численного анализа, или метод прогонки.

Обратите внимание на гибкость `diag()`: если аргумент является матрицей, она возвращает вектор, и наоборот. Кроме того, если аргумент является скаляром, функция возвращает матрицу тождественного преобразования заданного размера.

```
> m
     [,1] [,2]
[1,]    1    2
[2,]    7    8
> dm <- diag(m)
> dm
[1] 1 8
> diag(dm)
     [,1] [,2]
[1,]    1    0
[2,]    0    8
> diag(3)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Функция `sweep()` способна выполнять довольно сложные операции. Например, возьмем матрицу 3 × 3 и прибавим 1 к строке 1, 4 к строке 2 и 7 к строке 3.

```
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> sweep(m,1,c(1,4,7),"+")
     [,1] [,2] [,3]
[1,]    2    3    4
[2,]    8    9   10
[3,]   14   15   16
```

Первые два аргумента `sweep()` аналогичны аргументам `apply()`: массив и измерение (1 для строк в данном случае). Четвертый аргумент задает применяемую функцию, а третий — аргумент этой функции (для функции `"+"`).

8.4.1. Расширенный пример: векторное произведение

Перейдем к задаче вычисления векторных произведений. Определение выглядит очень просто: векторное произведение векторов (x_1, x_2, x_3) и (y_1, y_2, y_3) в трехмерном пространстве представляет собой новый трехмерный вектор, представленный в уравнении 8.1.

$$(x_2y_3 - x_3y_2,\ -x_1y_3 + x_3y_1,\ x_1y_2 - x_2y_1). \tag{8.1}$$

В компактном виде эта запись может быть выражена расширением верхней строки детерминанта, как показывает уравнение 8.2.

$$\begin{pmatrix} - & - & - \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{pmatrix}. \tag{8.2}$$

Здесь элементы верхней строки всего лишь резервируют место.

Не беспокойтесь об этой псевдоматематике. Суть в том, что векторное произведение может вычисляться как сумма субдетерминантов. Например, как нетрудно увидеть, первый компонент в уравнении 8.1 $x_2y_3 - x_3y_2$ является детерминантом подматрицы, полученной удалением первой строки и первого столбца в уравнении 8.2, как видно из уравнения 8.3.

$$\begin{pmatrix} x_2 & x_3 \\ y_2 & y_3 \end{pmatrix}. \tag{8.3}$$

Необходимость вычисления поддетерминантов (то есть детерминантов подматриц) идеально подходит для R и его превосходных средств определения подматриц. Напрашивается мысль вызвать `det()` для соответствующих подматриц:

```
xprod <- function(x,y) {
   m <- rbind(rep(NA,3),x,y)
   xp <- vector(length=3)
   for (i in 1:3)
      xp[i] <- -(-1)^i * det(m[2:3,-i])
   return(xp)
}
```

Обратите внимание: здесь задействована даже возможность определения значений NA в R — они нужны для представления временных «заполнителей», упоминавшихся выше.

Все это может показаться избыточным. В конце концов, можно без особого труда закодировать уравнение 8.1 напрямую, не прибегая к подматрицам и детерминантам. Возможно, в трехмерном случае это действительно так, но представленный метод очень полезен для n-арного случая в n-мерном пространстве. Векторное произведение определяется как детерминант $n \times n$ формы, представленной в уравнении 8.1, — таким образом, приведенный выше код идеально обобщается.

8.4.2. Расширенный пример: нахождение стационарных распределений для цепей Маркова

Цепью Маркова называется случайный процесс, который перемещается между разными состояниями без «запоминания» (формальное определение которого нас сейчас не интересует). Состоянием может быть количество заданий в очереди, количество товара на складе и т. д. Будем считать, что количество состояний конечно.

В качестве простого примера рассмотрим игру с многократным бросанием монетки. Вы выигрываете доллар при выпадении трех «орлов» подряд. Состоянием в любой момент `i` будет количество выпавших подряд «орлов» на текущий момент, так что состояние может быть равно 0, 1 или 2 (при выпадении трех «орлов» подряд состояние возвращается к 0).

При моделировании цепей Маркова основной интерес обычно представляет долгосрочное распределение состояний (то есть часть времени, проводимая в каждом состоянии, в достаточно продолжительной серии). В игре с броском монетки разработанный здесь код может использоваться для вычисления этого распределения. Как выясняется, состояния 0, 1 и 2 имеют пропорции 57,1 %, 28,6 % и 14,3 % времени. Следует заметить, что доллар выигрывается при выпадении «орла» в состоянии 2, так что лишь $0{,}143 \times 0{,}5 = 0{,}071$ броска приводит к выигрышу.

Поскольку индексы векторов и матриц R начинаются с 1, а не с 0, будет удобно переобозначить состояния 1, 2 и 3 вместо 0, 1 и 2. Например, состояние 3 означает, что в настоящий момент выпали два «орла» подряд.

Пусть p_{ij} обозначает *вероятность перехода* из состояния i в состояние j за квант времени. Например, в примере с игрой $p_{23} = 0{,}5$ это означает, что с вероятностью

1/2 выпадет «орел» и система перейдет из состояния с одним «орлом» подряд в состояние с двумя «орлами». С другой стороны, если в состоянии 2 выпадет «решка», произойдет переход в состояние 1 (0 «орлов» подряд), следовательно, $p_{21} = 0{,}5$.

Нас интересует вычисление вектора $\pi = (\pi_1, ..., \pi_s)$, где π_i — долгосрочная пропорция времени, проводимого в состоянии i для всех состояний i. Пусть P — матрица вероятностей переходов, а p_{ij} — элемент в строке i и столбце j. Можно показать, что вектор π должен удовлетворять уравнению 8.4:

$$\pi = \pi P \tag{8.4}$$

что эквивалентно уравнению 8.5:

$$(I - P^T)\pi = 0. \tag{8.5}$$

Здесь I — тождественная матрица, а P^T — результат транспонирования P.

Любое отдельное уравнение в системе 8.5 избыточно. Соответственно, мы удаляем одно из них, исключая последнюю строку $I - P$ в уравнении 8.5. Это также означает удаление последнего 0 из 0-вектора в правой части уравнения 8.5.

Но обратите внимание на ограничение, показанное в уравнении 8.6:

$$\sum_i \pi_i = 1. \tag{8.6}$$

В матричном контексте это выглядит так:

$$1_n^T \pi = 1,$$

где 1_n — вектор из n единиц.

Итак, в измененном варианте уравнения 8.5 удаленная строка заменяется строкой из единиц, а в правой части удаленный 0 заменяется 1. После этого можно переходить к решению системы.

Все эти вычисления могут быть выполнены функцией R `solve()`:

```
1 findpi1 <- function(p) {
2    n <- nrow(p)
3    imp <- diag(n) - t(p)
4    imp[n,] <- rep(1,n)
5    rhs <- c(rep(0,n-1),1)
6    pivec <- solve(imp,rhs)
7    return(pivec)
8 }
```

Основные шаги:

1. Вычислить $I-P^T$ в строке 3. Напомню, что `diag()` при вызове со скалярным аргументом возвращает тождественную матрицу с размером, заданным аргументом.
2. Заменить последнюю строку P значениями 1 (строка программы 4).
3. Создать в строке 5 вектор из правой части.
4. Найти решение для π в строке 6.

Другое решение, использующее более сложные задания, основано на собственных значениях. Из уравнения 8.4 следует, что π является левым собственным вектором P с собственным значением 1. Это наводит на мысль об использовании функции R `eigen()` с выбором собственного вектора, соответствующего этому собственному значению. (Для формального обоснования может использоваться результат из математической теории, называемый *теоремой Фробениуса—Перрона*.)

Поскольку π является левым собственным вектором, аргументом вызова `eigen()` должен быть результат транспонирования P, а не сама матрица P. Кроме того, поскольку собственный вектор уникален только до скалярного умножения, необходимо разобраться с двумя проблемами, касающимися собственного вектора, возвращаемого `eigen()`:

- Он может содержать отрицательные компоненты. В таком случае выполняется умножение на -1.
- Он может не удовлетворять уравнению 8.6. Проблема решается делением на длину возвращаемого вектора.

Код выглядит так:

```
1 findpi2 <- function(p) {
2    n <- nrow(p)
3    # Найти первый собственный вектор для транспонирования P
4    pivec <- eigen(t(p))$vectors[,1]
5    # Гарантированно вещественное, но может быть отрицательным
6    if (pivec[1] < 0) pivec <- -pivec
7    # Нормализовать по сумме
8    pivec <- pivec / sum(pivec)
9    return(pivec)
10 }
```

Возвращаемое значение `eigen()` представляет собой список. Одним из компонентов списка является матрица с именем `vectors`. i-й столбец является

собственным вектором, соответствующим i-му собственному значению. Соответственно, мы берем столбец 1.

8.5. Операции с множествами

Язык R включает ряд удобных операций с множествами, среди них следующие:

- `union(x,y)` — объединение множеств `x` и `y`;
- `intersect(x,y)` — пересечение множеств `x` и `y`;
- `setdiff(x,y)` — разность множеств `x` и `y` (все элементы `x`, отсутствующие в `y`);
- `setequal(x,y)` — проверка равенства `x` и `y`;
- `c %in% y` — проверка принадлежности (проверка того, является ли `c` элементом множества `y`);
- `choose(n,k)` — количество возможных подмножеств размером `k`, выбранных из множества размером `n`.

Несколько примеров использования этих функций:

```
> x <- c(1,2,5)
> y <- c(5,1,8,9)
> union(x,y)
[1] 1 2 5 8 9
> intersect(x,y)
[1] 1 5
> setdiff(x,y)
[1] 2
> setdiff(y,x)
[1] 8 9
> setequal(x,y)
[1] FALSE
> setequal(x,c(1,2,5))
[1] TRUE
> 2 %in% x
[1] TRUE
> 2 %in% y
[1] FALSE
> choose(5,2)
[1] 10
```

Как говорилось в разделе 7.12, вы можете писать собственные бинарные операции. Например, попробуем запрограммировать симметричную разность двух множеств, то есть все элементы, принадлежащие ровно одному из двух

множеств-операндов. Так как симметричная разность между множествами x и y состоит из элементов, входящих в x, но не входящих в y, и наоборот, код состоит из простых вызовов setdiff() и union():

```
> symdiff
function(a,b) {
   sdfxy <- setdiff(x,y)
   sdfyx <- setdiff(y,x)
   return(union(sdfxy,sdfyx))
}
```

Проверим, как работает новая операция:

```
> x
[1] 1 2 5
> y
[1] 5 1 8 9
> symdiff(x,y)
[1] 2 8 9
```

Другой пример: бинарный операнд для определения того, является ли одно множество u подмножеством другого множества v. Если немного подумать, становится ясно, что это свойство эквивалентно тому, что пересечение u и v равно u. А значит, мы получаем еще одну легко программируемую функцию:

```
> "%subsetof%" <- function(u,v) {
+    return(setequal(intersect(u,v),u))
+ }
> c(3,8) %subsetof% 1:10
[1] TRUE
> c(3,8) %subsetof% 5:10
[1] FALSE
```

Функция combn() генерирует сочетания. Найдем подмножества размером 2 для множества {1,2,3}.

```
> c32 <- combn(1:3,2)
> c32
     [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    3    3
> class(c32)
[1] "matrix"
```

Результаты содержатся в столбцах вывода. Как видите, множество {1,2,3} имеет три подмножества размером 2: (1,2), (1,3) и (2,3).

Функция `combn()` также позволяет задать функцию, которая должна вызываться для каждой комбинации. Например, можно найти сумму чисел каждого подмножества:

```
> combn(1:3,2,sum)
[1] 3 4 5
```

Первое подмножество {1,2} имеет сумму 3 и т. д.

8.6. Имитационное моделирование в R

Одно из самых распространенных применений R — имитационное моделирование. Посмотрим, какие средства существуют в R для этой области применения.

8.6.1. Встроенные генераторы случайных величин

Как упоминалось ранее, в R существуют функции генерирования величин из разных распределений. Например, `rbinom()` генерирует случайные величины с биномиальным распределением (или распределением Бернулли[1]).

Предположим, вы хотите вычислить вероятность выпадения как минимум четырех «орлов» из пяти бросков монеты (задача легко решается аналитически, но из нее получится хороший пример). Вот как это делается:

```
> x <- rbinom(100000,5,0.5)
> mean(x >= 4)
[1] 0.18829
```

Сначала мы сгенерируем 100 000 величин из биномиального распределения с пятью испытаниями и вероятностью успеха 0,5. Затем мы определим, какие из них имеют значение 4 или 5, в результате чего будет получен логический вектор той же длины, что и `x`. Функция `mean()` интерпретирует значения `TRUE` и `FALSE` в этом векторе как 1 и 0, в результате чего мы получаем оценку вероятности (поскольку среднее значение множества 1 и 0 равно пропорции 1).

Среди других функций стоит упомянуть `rnorm()` для нормального распределения, `rexp()` для экспоненциального распределения, `runif()` для равномерного распределения, `rgamma()` для гамма-распределения, `rpois()` для распределения Пуассона и т. д.

[1] Серия независимых случайных переменных со значениями 0 и 1, обладающими одинаковой вероятностью 1 для каждого случая, называется последовательностью Бернулли.

В другом простом примере вычисляется E[max(X, Y)], ожидаемое значение максимума независимых случайных величин X и Y с распределением N(0,1):

```
sum <- 0
nreps <- 100000
for (i in 1:nreps) {
   xy <- rnorm(2) # Сгенерировать 2 величины N(0,1)
   sum <- sum + max(xy)
}
print(sum/nreps)
```

Мы сгенерировали 100 000 пар, нашли максимум для каждой пары и усреднили эти максимумы для того, чтобы получить оценку ожидаемого значения.

Возможно, предыдущая версия кода с явным циклом выглядит более понятно, но как и прежде, если вы не против дополнительных затрат памяти, то же самое можно сделать более компактно.

```
> emax
function(nreps) {
   x <- rnorm(2*nreps)
   maxxy <- pmax(x[1:nreps],x[(nreps+1):(2*nreps)])
   return(mean(maxxy))
}
```

В этом фрагменте генерируются два значения `nreps`: первое моделирует X, а второе Y. Затем вызов `pmax()` вычисляет попарные максимумы, которые нам нужны. И снова обратите внимание на различия между `max()` и `pmax()` — последняя определяет максимумы для пар.

8.6.2. Получение одной случайной серии при повторных запусках

Согласно документации R, все генераторы случайных чисел используют 32-разрядные целые числа для инициализации. Таким образом, если не считать ошибки округления, одно значение должно генерировать один и тот же поток чисел.

По умолчанию R генерирует разные серии случайных чисел для разных запусков программы. Если вы хотите, чтобы каждый раз воспроизводился один числовой поток — например, это может быть важно при отладке, — вызовите `set.seed()`:

```
> set.seed(8888) # или ваше любимое число
```

8.6.3. Расширенный пример: комбинаторное моделирование

Рассмотрим следующую задачу на вычисление вероятности.

Из 20 людей выбираются три комитета, состоящие из 3, 4 и 5 человек. Какова вероятность того, что люди A и B будут выбраны в один комитет?

Эта задача легко решается аналитически, но возможно, вы предпочтете проверить наше решение посредством моделирования; и в любом случае написание кода продемонстрирует, как операции с множествами R могут использоваться в сочетании с комбинаторными задачами.

Код выглядит так:

```
sim <- function(nreps) {
   commdata <- list() # Для хранения всей информации о трех комитетах
   commdata$countabsamecomm <- 0
   for (rep in 1:nreps) {
      commdata$whosleft <- 1:20 # Остается для выбора
      commdata$numabchosen <- 0 # Количество выбранных из A и B
      # Выбрать комитет 1 и проверить одновременное вхождение A,B
      commdata <- choosecomm(commdata,5)
      # Если A или B уже выбраны, проверять другие комитеты не нужно.
      if (commdata$numabchosen > 0) next
      # Проверить комитет 2 и проверить
      commdata <- choosecomm(commdata,4)
      if (commdata$numabchosen > 0) next
      # Выбрать комитет 3 и проверить
      commdata <- choosecomm(commdata,3)
   }
   print(commdata$countabsamecomm/nreps)
}

choosecomm <- function(comdat,comsize) {
   # Выбрать комитет
   committee <- sample(comdat$whosleft,comsize)
   # Подсчитать, сколько из кандидатов A и B были выбраны
   comdat$numabchosen <- length(intersect(1:2,committee))
   if (comdat$numabchosen == 2)
      comdat$countabsamecomm <- comdat$countabsamecomm + 1
   # Удалить выбранный комитет из набора доступных кандидатов
   comdat$whosleft <- setdiff(comdat$whosleft,committee)
   return(comdat)
}
```

Потенциальным участникам комитетов присваиваются номера с 1 до 20; кандидатам A и B соответствуют идентификаторы 1 и 2. Вспомните, что списки R

часто используются для хранения нескольких связанных переменных в одном контейнере; мы создаем список `comdat` со следующими компонентами:

- `comdat$whosleft` — случайная выборка комитетов моделируется случайным выбором элементов из этого вектора. Каждый раз при выборе комитета удаляются идентификаторы участников комитета. Вектор инициализируется интервалом `1:20`; это означает, что никто еще не был выбран.
- `comdat$numabchosen` — количество кандидатов, выбранных из A и B на настоящий момент. Если вы выбрали комитет и значение оказалось положительным, остальные комитеты можно пропустить по следующей причине: если число равно 2, мы точно знаем, что A и B входят в один комитет; если оно равно 1, мы точно знаем, что A и B не входят в один комитет.
- `comdat$countabsamecomm` — здесь хранится количество случаев, в которых A и B оказывались в одном комитете.

Так как в выборе участников комитета задействованы подмножества, неудивительно, что в реализации задействована пара операций с множествами R — `intersect()` и `setdiff()`. Также обратите внимание на команду `next`, которая приказывает R пропустить оставшуюся часть текущей итерации цикла.

9 Объектно-ориентированное программирование

Многие программисты согласны с тем, что объектно-ориентированное программирование (ООП) способствует написанию более понятного кода, пригодного для повторного использования. И хотя R очень сильно отличается от знакомых ООП-языков, таких как C++, Java и Python, в R присутствуют очень многие отличительные признаки ООП.

Следующие факты играют ключевую роль в R:

- Все, с чем вы имеете дело в R — от чисел и символьных строк до матриц, — является объектом.
- R способствует применению *инкапсуляции* — хранению различных, но логически связанных элементов данных в одном экземпляре класса. Инкапсуляция помогает отслеживать взаимосвязанные переменные и делает код более понятным.
- Классы R являются *полиморфными*; это означает, что один и тот же вызов функции может приводить к выполнению различных действий для объектов разных классов. Например, вызов `print()` для объекта конкретного класса инициирует вызов функции `print`, предназначенной именно для этого класса. Полиморфизм способствует повторному использованию кода.
- В R поддерживается *наследование*, позволяющее расширить класс общего назначения в более специализированную версию.

В этой главе охватывается применение ООП в R. Мы рассмотрим программирование для двух разновидностей классов, S3 и S4, а затем представим несколько полезных средств R, относящихся к ООП.

9.1. Классы S3

Исходная структура классов R, называемая *S3*, все еще остается доминирующей парадигмой классов в современном R. Кстати говоря, большинство встроенных классов R относится к типу S3.

Класс S3 состоит из списка, атрибута с именем класса и функциональности *диспетчеризации*. Последняя позволяет использовать обобщенные функции вроде тех, которые были представлены в главе 1. Классы S4 были созданы позднее для повышения *безопасности*; они предотвращают случайное добавление компонентов класса, которые еще не существуют.

9.1.1. Обобщенные функции S3

Как упоминалось ранее, R является полиморфным языком в том смысле, что одна и та же функция может инициировать разные операции для разных классов. Например, вы можете вызвать `plot()` для разных типов объектов, получая разные типы графиков для разных объектов. То же относится к `print()`, `summary()` и многим другим функциям.

Таким образом можно создать унифицированный интерфейс к разным классам. Например, если вы пишете код с построением графиков, то полиморфизм позволит вам программировать, не беспокоясь о разных типах объектов, для которых могут строиться графики.

Кроме того, с полиморфизмом пользователю будет проще запомнить, как пользоваться вашим кодом, а исследование новых библиотечных функций и связанных с ними классов станет более увлекательным и удобным. Столкнувшись с новой функцией, попробуйте вызвать `plot()` для вывода этой функции; скорее всего, попытка сработает. С точки зрения программиста полиморфизм позволяет писать достаточно общий код, не беспокоясь о том, с какими типами объектов он работает, потому что базовые механизмы классов решат эту проблему за него.

Функции, работающие с полиморфизмом (такие, как `plot()` и `print()`), называются *обобщенными* (generic) функциями. При вызове обобщенной функции R осуществляет диспетчеризацию вызова (то есть передает его) правильному методу класса; это означает, что вызов будет перенаправлен функции, определенной для класса объекта.

9.1.2. Пример: ООП в функции линейной модели lm()

Для примера рассмотрим простой регрессионный анализ с использованием функции R `lm()`. Сначала посмотрим, что делает `lm()`:

```
> ?lm
```

В выводе этого запроса среди прочего сообщается, что эта функция возвращает объект класса "lm".

Попробуем создать экземпляр этого класса, а затем вывести его:

```
> x <- c(1,2,3)
> y <- c(1,3,8)
> lmout <- lm(y ~ x)
> class(lmout)
[1] "lm"
> lmout

Call:
lm(formula=y~x)

Coefficients:
(Intercept)          x
       -3.0        3.5
```

Здесь выводится объект lmout. (Вспомните, что если просто ввести в интерактивном режиме имя объекта, будет выведен сам объект.) Интерпретатор R видит, что lmout является объектом класса "lm", и поэтому вызывает print.lm() — специальный метод вывода для класса "lm". В терминологии R вызов обобщенной функции print() был передан методу print.lm(), связанному с классом "lm".

А теперь рассмотрим обобщенную функцию и метод класса для данного примера:

```
> print
function(x, ...) UseMethod("print")
<environment: namespace:base>
> print.lm
function (x, digits = max(3, getOption("digits") - 3), ...)
{
   cat("\nCall:\n", deparse(x$call), "\n\n", sep = "")
   if (length(coef(x))) {
      cat("Coefficients:\n")
      print.default(format(coef(x), digits = digits), print.gap = 2,
         quote = FALSE)
   }
   else cat("No coefficients\n")
   cat("\n")
   invisible(x)
}
<environment: namespace:stats>
```

Возможно, вас удивит, что `print()` состоит исключительно из вызова `UseMethod()`. Но в действительности это диспетчерская функция, и ввиду того, что `print()` является обобщенной функцией, ничего странного в этом нет.

Не отвлекайтесь на подробности `print.lm()`. Здесь главное то, что вывод зависит от контекста, а для класса `"lm"` вызывается специальная функция вывода. Теперь посмотрим, что происходит при выводе этого объекта с удаленным атрибутом класса:

```
> unclass(lmout)
$coefficients
(Intercept)           x
       -3.0         3.5

$residuals
  1    2    3
0.5 -1.0  0.5

$effects
(Intercept)           x
  -6.928203   -4.949747   1.224745

$rank
[1] 2
...
```

Я привожу только несколько начальных строк — на самом деле их намного больше. (Попробуйте повторить этот пример самостоятельно!) Но вы увидите, что автор `lm()` решил сделать `print.lm()` более компактным, ограничив его несколькими ключевыми характеристиками.

9.1.3. Поиск реализаций обобщенных методов

Чтобы получить все реализации заданного обобщенного метода, вызовите функцию `methods()`:

```
> methods(print)
 [1] print.acf*
 [2] print.anova
 [3] print.aov*
 [4] print.aovlist*
 [5] print.ar*
 [6] print.Arima*
```

```
 [7] print.arima0*
 [8] print.AsIs
 [9] print.aspell*
[10] print.Bibtex*
[11] print.browseVignettes*
[12] print.by
[13] print.check_code_usage_in_package*
[14] print.check_demo_index*
[15] print.checkDocFiles*
[16] print.checkDocStyle*
[17] print.check_dotInternal*
[18] print.checkFF*
[19] print.check_make_vars*
[20] print.check_package_code_syntax*
...
```

Звездочками помечены *невидимые* функции, то есть функции, не входящие в пространства имен по умолчанию. Вы можете найти эти функции вызовом `getAnywhere()`, а затем обратиться к ним с указанием квалификатора пространства имен (например, `print.aspell()`). Функция `aspell()` проверяет орфографию в файле, заданном аргументом. Например, представьте, что файл `wrds` состоит из следующей строки:

```
Which word is mispelled?
```

В этом случае функция обнаружит слово, содержащее опечатку:

```
aspell("wrds")
mispelled
wrds:1:15
```

Из вывода следует, что в символе 15 строки 1 входного файла обнаружена орфографическая ошибка. Но нас здесь интересует прежде всего механизм вывода этого результата.

Функция `aspell()` возвращает объект класса `"aspell"`, который имеет собственную обобщенную функцию вывода `print.aspell()`. Эта функция была вызвана в нашем примере после вызова `aspell()`, и возвращаемое значение было выведено на экране. В этот момент R вызывает `UseMethod()` для объекта класса `"aspell"`. Но если вызвать этот метод напрямую, R его не опознает:

```
> aspout <- aspell("wrds")
> print.aspell(aspout)
Error: could not find function "print.aspell"
```

Впрочем, метод можно найти вызовом `getAnywhere()`:

```
> getAnywhere(print.aspell)
A single object matching 'print.aspell' was found
It was found in the following places
   registered S3 method for print from namespace utils
   namespace:utils
with value

function (x, sort = TRUE, verbose = FALSE, indent = 2L, ...)
{
   if (!(nr <- nrow(x)))
...
```

Итак, функция находится в пространстве имен `utils`, и ее можно выполнить с добавлением квалификатора:

```
> utils:::print.aspell(aspout)
mispelled
wrds:1:15
```

Таким образом можно просмотреть все обобщенные методы:

```
> methods(class="default")
...
```

9.1.4. Написание классов S3

Классы S3 имеют довольно примитивную структуру. Чтобы создать экземпляр класса, следует создать список, компоненты которого являются переменными класса. (Для читателей, знакомых с Perl: эта узкоспециализированная природа реализации также присуща ООП-системе языка Perl.) Атрибут `"class"` устанавливается вручную при помощи функции `attr()` или `class()`, после чего определяются различные реализации обобщенных функций. Чтобы убедиться в этом для функции `lm()`, проверьте код функции:

```
>lm
...
z <- list(coefficients = if (is.matrix(y))
                    matrix(,0,3) else numeric(0L), residuals = y,
                  fitted.values = 0 * y, weights = w, rank = 0L,
                  df.residual = if (is.matrix(y)) nrow(y) else length(y))
}
...
class(z) <- c(if(is.matrix(y)) "mlm", "lm")
...
```

И снова не обращайте внимания на подробности, важна суть происходящего. Список был создан и присвоен переменной z, которая послужит основой для экземпляра класса "lm" (и в конечном итоге станет значением, возвращаемым функцией). Некоторые компоненты этого списка, например residuals, уже были заданы в момент создания списка. Кроме того, атрибуту class было присвоено значение "lm" (а возможно, "mlm"; об этом в следующем разделе).

Чтобы продемонстрировать написание классов S3, мы переключимся на более простую задачу. Продолжая пример с работником из раздела 4.1, можно написать следующий код:

```
> j <- list(name="Joe", salary=55000, union=T)
> class(j) <- "employee"
> attributes(j) # Проверка
$names
[1] "name" "salary" "union"

$class
[1] "employee"
```

Прежде чем писать метод print для этого класса, посмотрим, что произойдет при вызове версии print() по умолчанию:

```
>j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

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

Фактически в отношении вывода j интерпретируется как список. А теперь напишем собственный метод print:

```
print.employee <- function(wrkr) {
   cat(wrkr$name,"\n")
   cat("salary",wrkr$salary,"\n")
   cat("union member",wrkr$union,"\n")
}
```

Итак, любой вызов `print()` для объекта класса `"employee"` теперь должен обозначаться `print.employee()`. В этом можно убедиться формально:

```
> methods(,"employee")
[1] print.employee
```

Или, конечно, можно просто опробовать его на практике:

```
>j
Joe
salary 55000
union member TRUE
```

9.1.5. Наследование

Идея наследования заключается в формировании новых классов как специализированных версий старых классов. Например, в примере с работниками новый класс для почасовых работников `"hrlyemployee"` как подкласс `"employee"` может определяться следующим образом:

```
k <- list(name="Kate", salary= 68000, union=F, hrsthismonth= 2)
class(k) <- c("hrlyemployee","employee")
```

Новый класс содержит одну дополнительную переменную: `hrsthismonth`. Имя нового класса состоит из двух символьных строк, представляющих новый и старый класс. Новый класс наследует методы от старого класса. Например, `print.employee()` работает и для нового класса:

```
> k
Kate
salary 68000
union member FALSE
```

С учетом целей наследования это вполне логично. Тем не менее важно точно понимать, что здесь происходит.

Если вы просто введете `k`, это приведет к вызову `print(k)`. В свою очередь, это заставляет `UseMethod()` искать метод вывода для первого из двух имен классов `k`, `"hrlyemployee"`. Поиск завершился неудачей, поэтому `UseMethod()` проверяет другое имя класса `"employee"` и находит `print.employee()`. Найденный метод выполняется.

Вспомните, что при просмотре кода `"lm"` вы видели следующую строку:

```
class(z) <- c(if(is.matrix(y)) "mlm", "lm")
```

Она показывает, что "mlm" является подклассом "lm" для переменных отклика с векторными значениями.

9.1.6. Расширенный пример: класс для хранения верхних треугольных матриц

Пора рассмотреть более сложный пример, в котором мы напишем класс R "ut" для верхних треугольных матриц. Так называются квадратные матрицы, элементы которых под диагональю заполнены нулями, как показано в уравнении 9.1.

$$\begin{pmatrix} 1 & 5 & 12 \\ 0 & 6 & 9 \\ 0 & 0 & 2 \end{pmatrix}. \tag{9.1}$$

Здесь мы стремимся к экономии памяти (за счет незначительного увеличения времени доступа), сохраняя только ненулевую часть матрицы.

ПРИМЕЧАНИЕ

Класс R «dist» тоже использует такой механизм хранения, хотя и в более узком контексте и без функций класса, имеющихся в нашем случае.

Компонент mat этого класса используется для хранения матрицы. Как упоминалось ранее, для экономии памяти храниться будут только элементы диагонали и находящиеся выше диагонали. Например, матрица 9.1 состоит из вектора (1, 5, 6, 12, 9, 2), и компонент mat содержит это значение.

В класс будет включен компонент ix, показывающий, где в матрице начинаются различные столбцы. Для приведенного примера ix содержит c(1,2,4); это означает, что столбец 1 начинается с mat[1], столбец 2 начинается с mat[2], а столбец 3 начинается с mat[4]. Таким образом обеспечивается удобный доступ к отдельным элементам или столбцам матрицы.

```
# Класс "ut", компактное хранение верхних треугольных матриц

# Вспомогательная функция, возвращает 1+...+i
sum1toi <- function(i) return(i*(i+1)/2)

# Создать объект класса "ut" по полной матрице inmat (с нулями)
ut <- function(inmat) {
   n <- nrow(inmat)
   rtrn <- list() # Начать построение объекта
```

```
   class(rtrn) <- "ut"
   rtrn$mat <- vector(length=sum1toi(n))
   rtrn$ix <- sum1toi(0:(n-1)) + 1
   for (i in 1:n) {
      # Сохранить столбец i
      ixi <- rtrn$ix[i]
      rtrn$mat[ixi:(ixi+i-1)] <- inmat[1:i,i]
   }
   return(rtrn)
}

# Распаковать utmat в полную матрицу
expandut <- function(utmat) {
   n <- length(utmat$ix) # Количество строк и столбцов матрицы
   fullmat <- matrix(nrow=n,ncol=n)
   for (j in 1:n) {
      # Заполнить j-й столбец
      start <- utmat$ix[j]
      fin <- start+j-1
      abovediagj <- utmat$mat[start:fin] # Часть столбца j над диагональю
      fullmat[,j] <- c(abovediagj,rep(0,n-j))
   }
   return(fullmat)
}

# Вывести матрицу
print.ut <- function(utmat)
   print(expandut(utmat))

# Умножение двух матриц ut с возвращением другого экземпляра ut;
# реализуется как бинарная операция
"%mut%" <- function(utmat1,utmat2) {
   n <- length(utmat1$ix) # Количество строк и столбцов матрицы
   utprod <- ut(matrix(0,nrow=n,ncol=n))
   for (i in 1:n) { # Вычислить столбец i произведения
      # Пусть a[j] и bj — столбцы j для utmat1 и utmat2 соответственно;
      # например, b2[1] обозначает элемент 1 столбца 2 матрицы utmat2.
      # Тогда столбец i произведения равен
      # bi[1]*a[1] + ... + bi[i]*a[i]
      # Найти индекс начала столбца i в utmat2.
      startbi <- utmat2$ix[i]
      # Инициализировать вектор для суммы bi[1]*a[1] + ... + bi[i]*a[i]
      prodcoli <- rep(0,i)
      for (j in 1:i) { # Найти bi[j]*a[j], добавить в prodcoli
         startaj <- utmat1$ix[j]
         bielement <- utmat2$mat[startbi+j-1]
         prodcoli[1:j] <- prodcoli[1:j] +
            bielement * utmat1$mat[startaj:(startaj+j-1)]
```

```
58        }
59        # Теперь присоединить нижние нули
60        startprodcoli <- sum1toi(i-1)+1
61        utprod$mat[startbi:(startbi+i-1)] <- prodcoli
62     }
63     return(utprod)
64 }
```

А теперь протестируем его.

```
> test
function() {
   utm1 <- ut(rbind(1:2,c(0,2)))
   utm2 <- ut(rbind(3:2,c(0,1)))
   utp <- utm1 %mut% utm2
   print(utm1)
   print(utm2)
   print(utp)
   utm1 <- ut(rbind(1:3,0:2,c(0,0,5)))
   utm2 <- ut(rbind(4:2,0:2,c(0,0,1)))
   utp <- utm1 %mut% utm2
   print(utm1)
   print(utm2)
   print(utp)
}
> test()
     [,1] [,2]
[1,] 1 2
[2,] 0 2
     [,1] [,2]
[1,] 3 2
[2,] 0 1
     [,1] [,2]
[1,] 3 4
[2,] 0 2
     [,1] [,2] [,3]
[1,]123
[2,]012
[3,]005
     [,1] [,2] [,3]
[1,]432
[2,]012
[3,]001
     [,1] [,2] [,3]
[1,]459
[2,]014
[3,]005
```

В этом коде учитывается тот факт, что матрицы содержат большое количество нулей. Например, мы избегаем умножения на нуль, просто не добавляя к суммам слагаемые с нулевыми множителями.

Функция `ut()` достаточно тривиальна. Это функция-*конструктор*, то есть функция, которая создает экземпляр заданного класса и в итоге возвращает этот экземпляр. Таким образом, в строке 9 создается список, который послужит «телом» для объекта класса. Присваиваем ему имя `rtrn` как напоминание о том, что это экземпляр класса, который нужно сконструировать и вернуть.

Как упоминалось ранее, главными переменными нашего класса будут переменные `mat` и `ix`, реализованные в виде компонентов списка. Память для этих двух компонентов выделяется в строках 11 и 12.

Цикл, который следует после этого, заполняет `rtrn$mat` по столбцам, а затем заполняет `rtrn$idx` элемент за элементом. Этот цикл `for` можно записать и более элегантно — с использованием малоизвестных функций `row()` и `col()`. Функция `row()` получает матрицу на входе и возвращает новую матрицу того же размера, в которой каждый элемент заменяется своим номером строки. Пример:

```
> m
     [,1] [,2]
[1,] 1 4
[2,] 2 5
[3,] 3 6
> row(m)
     [,1] [,2]
[1,] 1 1
[2,] 2 2
[3,] 3 3
```

Функция `col()` работает аналогично.

Эти функции позволяют заменить цикл `for` в `ut()` однострочной командой:

```
rtrn$mat <- inmat[row(inmat) <= col(inmat)]
```

Там, где это возможно, следует применять векторизацию. Например, взгляните на строку 12:

```
rtrn$ix <- sum1toi(0:(n-1)) + 1
```

Так как функция `sum1toi()` (определяемая в строке 4) основана только на векторизированных функциях `"*"()` и `"+"()`, функция `sum1toi()` тоже векторизирована. Это позволяет применить `sum1toi()` к вектору в приведенной команде. Также обратите внимание на применение переработки.

Мы хотим, чтобы класс "ut" включал не только переменные, но и методы. Для этого были включены три метода:

- Функция expandut() преобразует сжатую матрицу в обычную. В expandut() особенно важны строки 27 и 28, в которых rtrn$ix используется для определения места хранения j-го столбца нашей матрицы в utmat$mat. Затем эти данные копируются в j-й столбец fullmat в строке 30. Обратите внимание на использование rep() для генерирования нулей в нижней части столбца.
- Функция print.ut() — простая и быстрая функция вывода, использующая expandut(). Напомню, что любой вызов print() для объекта типа "ut" будет передан print.ut(), как в приведенных ранее примерах.
- Функция "%mut%"() предназначена для умножения двух сжатых матриц (без их распаковки). Код начинается в строке 39. Поскольку операция является бинарной, мы используем тот факт, что R поддерживает определение пользовательских бинарных операций (раздел 7.12), и реализуем функцию умножения матриц как %mut%.

Рассмотрим функцию "%mut%"() более подробно. Сначала в строке 43 выделяется память для матрицы произведения. Обратите внимание на использование переработки в необычном контексте. Первый аргумент matrix() должен быть вектором длины, совместимой с количеством заданных строк и столбцов, поэтому переданный 0 перерабатывается в вектор длиной n^2. Конечно, вместо этого можно воспользоваться функцией rep(), но применение переработки делает код чуть более коротким и элегантным.

Для ясности и хорошей скорости выполнения написанный код использует тот факт, что R хранит матрицы по столбцам. Как упоминается в комментариях, в коде используется тот факт, что столбец i произведения может быть выражен как линейная комбинация столбцов первого множителя. Будет полезно рассмотреть конкретный пример этого свойства, представленный в уравнении 9.2.

$$\begin{pmatrix}1&2&3\\0&1&2\\0&0&5\end{pmatrix}\begin{pmatrix}4&3&2\\0&1&2\\0&0&1\end{pmatrix}=\begin{pmatrix}4&5&9\\0&1&4\\0&0&5\end{pmatrix}. \tag{9.2}$$

Например, столбец 3 произведения вычисляется следующим образом:

$$2\begin{pmatrix}1\\0\\0\end{pmatrix}+2\begin{pmatrix}2\\1\\0\end{pmatrix}+1\begin{pmatrix}3\\2\\5\end{pmatrix}.$$

Анализ уравнения 9.2 подтверждает эту связь.

Представление задачи умножения в контексте столбцов двух входных матриц позволяет сократить код и, скорее всего, повысить скорость его выполнения. Последний факт опять-таки следует из векторизации — свойства, подробно рассматриваемого в главе 14. Он используется в цикле `for`, начиная со строки 53. (Хотя надо признать, что в данном случае за повышение скорости приходится расплачиваться удобочитаемостью кода.)

9.1.7. Расширенный пример: полиномиальная регрессия

В очередном примере мы рассмотрим статистическую регрессию с одной переменной-предиктором. Поскольку любая статистическая модель является всего лишь аппроксимацией, в принципе, можно строить все более и более качественные модели, осуществляя подгонку полиномиальных моделей все более высоких степеней. Тем не менее в какой-то момент это приведет к чрезмерной подгонке и, начиная с какой-то степени, качество прогнозирования будущих данных начнет снижаться.

Класс `"polyreg"` пытается решить эту проблему. Он осуществляет подгонку полиномиальных моделей различных степеней, но в то же время производит перекрестную проверку для снижения риска чрезмерной подгонки. В этой разновидности перекрестной проверки, называемой *перекрестной проверкой с исключением* (leaving-one-out), для каждой точки осуществляется подгонка регрессии по всем данным, кроме этого наблюдения, после чего это наблюдение прогнозируется по результатам подгонки. Объект этого класса содержит результаты различных регрессионных моделей, а также исходные данные. Ниже приведен код класса `"polyreg"`.

```
 1 # "polyreg," класс S3 для полиномиальной регрессии с одним предиктором
 2
 3 # polyfit(y,x,maxdeg) подгоняет все полиномиальные модели до степени
 4 # maxdeg; y — вектор переменной реакции, x — предиктор; создает
 5 # объект класса "polyreg"
 6 polyfit <- function(y,x,maxdeg) {
 7    # Вычислить степени предиктора, i-я степень в i-м столбце
 8    pwrs <- powers(x,maxdeg)
 9    lmout <- list() # Начать построение класса
10    class(lmout) <- "polyreg" # Создать новый класс
11    for (i in 1:maxdeg) {
12       lmo <- lm(y ~ pwrs[,1:i])
13       # Расширить класс lm с перекрестной проверкой прогнозов
14       lmo$fitted.cvvalues <- lvoneout(y,pwrs[,1:i,drop=F])
```

```
15       lmout[[i]] <- lmo
16    }
17    lmout$x <- x
18    lmout$y <- y
19    return(lmout)
20 }
21
22 # print() для объектов класса "polyreg": вывести среднеквадратические
23 # ошибки прогнозирования с перекрестной проверкой
24 print.polyreg <- function(fits) {
25    maxdeg <- length(fits) - 2
26    n <- length(fits$y)
27    tbl <- matrix(nrow=maxdeg,ncol=1)
28    colnames(tbl) <- "MSPE"
29    for (i in 1:maxdeg) {
30       fi <- fits[[i]]
31       errs <- fits$y - fi$fitted.cvvalues
32       spe <- crossprod(errs,errs) # Сумма квадратов ошибок прогнозов
33       tbl[i,1] <- spe/n
34    }
35    cat("mean squared prediction errors, by degree\n")
36    print(tbl)
37 }
38
39 # Формирует матрицу степеней вектора x до степени dg
40 powers <- function(x,dg) {
41    pw <- matrix(x,nrow=length(x))
42    prod <- x
43    for (i in 2:dg) {
44       prod <- prod * x
45       pw <- cbind(pw,prod)
46    }
47    return(pw)
48 }
49
50 # Находит прогнозируемые значения с перекрестной проверкой; может
51 # быть существенно ускорен методом обновления обратных матриц
52 lvoneout <- function(y,xmat) {
53    n <- length(y)
54    predy <- vector(length=n)
55    for (i in 1:n) {
56       # Регрессия с исключением i-го наблюдения
57       lmo <- lm(y[-i] ~ xmat[-i,])
58       betahat <- as.vector(lmo$coef)
59       # 1 для постоянной составляющей
60    predy[i] <- betahat %*% c(1,xmat[i,])
61    }
62    return(predy)
```

```
63 }
64
65 # Полиномиальная функция x, коэффициенты cfs
66 poly <- function(x,cfs) {
67    val <- cfs[1]
68    prod <- 1
69    dg <- length(cfs) - 1
70    for (i in 1:dg) {
71       prod <- prod * x
72       val <- val + cfs[i+1] * prod
73    }
74 }
```

Как видно из листинга, класс "polyreg" состоит из polyfit() (функция-конструктор) и print.polyreg() (функция вывода, адаптированная для этого класса). Он также содержит ряд вспомогательных функций для вычисления степеней и полиномиальных функций и для выполнения перекрестной проверки. (Учтите, что в некоторых случаях эффективность была принесена в жертву ясности кода.)

В качестве примера использования класса мы сгенерируем некоторый объем искусственных данных, создадим объект класса "polyreg" и выведем результаты.

```
> n <-60
> x <- (1:n)/n
> y <- vector(length=n)
> for (i in 1:n) y[i] <- sin((3*pi/2)*x[i]) + x[i]^2 + rnorm(1,mean=0,sd=0.5)
> dg <-15
> (lmo <- polyfit(y,x,dg))
mean squared prediction errors, by degree
           MSPE
 [1,] 0.4200127
 [2,] 0.3212241
 [3,] 0.2977433
 [4,] 0.2998716
 [5,] 0.3102032
 [6,] 0.3247325
 [7,] 0.3120066
 [8,] 0.3246087
 [9,] 0.3463628
[10,] 0.4502341
[11,] 0.6089814
[12,] 0.4499055
[13,]        NA
[14,]        NA
[15,]        NA
```

Обратите внимание: сначала в этой команде используется стандартный прием R:

```
> (lmo <- polyfit(y,x,dg))
```

Заключая всю команду присваивания в круглые скобки, мы получаем вывод и формируем `lmo` одновременно на тот случай, если последнее значение понадобится для других целей.

Функция `polyfit()` осуществляет подгонку полиномиальных моделей до заданной степени (в данном случае 15), вычисляя среднеквадратическую ошибку прогноза для каждой модели. Несколько последних значений в выводе содержали `NA`, потому что R из-за ошибок округления отказывается от подгонки полиномиальных моделей такой степени.

Как все это делается? Основная работа выполняется функцией `polyfit()`, которая создает объект класса `"polyreg"`. Этот объект состоит в основном из объектов, возвращаемых регрессионной системой подгонки R `lm()` для каждой степени.

При построении этих объектов обратите внимание на строку 14:

```
lmo$fitted.cvvalues <- lvoneout(y,pwrs[,1:i,drop=F])
```

Здесь `lmo` — объект, возвращаемый `lm()`, но в него добавляется дополнительный компонент: `fitted.cvvalues`. Новый компонент можно добавить в список в любой момент, а классы S3 являются списками, поэтому это возможно сделать.

Также в строке 24 определяется метод для обобщенной функции `print()`, `print.polyreg()`. В разделе 12.1.5 мы добавим метод для обобщенной функции `plot()`, `plot.polyreg()`.

При вычислении ошибок прогнозирования используется перекрестная проверка с исключением в форме, прогнозирующей каждое наблюдение по всем остальным. Для реализации этого механизма используются возможности отрицательных индексов в R в строке 57:

```
lmo <- lm(y[-i] ~ xmat[-i,])
```

Таким образом, подгонка модели осуществляется с исключением `i`-го наблюдения из набора данных.

ПРИМЕЧАНИЕ

Как упоминалось в комментариях к коду, реализацию можно значительно ускорить при помощи метода обновления обратных матриц — так называемой формулы Шермана—Моррисона—Вудбери. За дополнительной информацией обращайтесь к J. H. Venter and J. L. J. Snyman, «A Note on the Generalised Cross-Validation Criterion in Linear Model Selection», Biometrika, Vol. 82, no. 1, pp. 215–219.

9.2. Классы S4

Некоторые программисты считают, что S3 не обеспечивает безопасности, обычно ассоциируемой с ООП. Например, в приводившемся ранее примере с базой данных класс "employee" содержал три поля: name, salary и union.

В такой ситуации есть три возможные проблемы:

- Мы забыли задать признак в поле union.
- Вместо union мы ошибочного написали onion.
- Мы создали объект класса, отличного от "employee", но при этом случайно задали атрибуту класс значение "employee".

В каждом из этих случаев R ни на что не пожалуется. Цель S4 — обеспечить выдачу предупреждений и предотвратить подобные случайности.

Структуры S4 существенно превосходят по функциональности структуры S3, но здесь будут представлены только основы. В табл. 9.1 приведена сводка различий между этими классами.

Таблица 9.1. Основные операторы R

Операция	S3	S4
Определение класса	Неявно в коде конструктора	setClass()
Создание объекта	Построение списка, назначение атрибута class	new()
Обращение к переменной класса	$	@
Реализация обобщенной функции f()	Определение f.className()	setMethod()
Объявление обобщенной функции	UseMethod()	setGeneric()

9.2.1. Написание классов S4

Классы S4 определяются вызовом setClass(). Продолжая пример с базой данных работников, можно написать следующий код:

```
> setClass("employee",
+    representation(
+       name="character",
+       salary="numeric",
+       union="logical")
+ )
[1] "employee"
```

В этом фрагменте определяется новый класс "employee" с тремя переменными заданных типов.

Теперь создадим экземпляр этого класса для работника Joe при помощи new(), встроенной функции-конструктора классов S4:

```
> joe <- new("employee",name="Joe",salary=55000,union=T)
> joe
An object of class "employee"
Slot "name":
[1] "Joe"
Slot "salary":
[1] 55000

Slot "union":
[1] TRUE
```

Обратите внимание: переменные класса называются *слотами* (slots) и обозначаются символом @. Пример:

```
> joe@salary
[1] 55000
```

Также можно воспользоваться функцией slot() — допустим, для получения зарплаты работника:

```
> slot(joe,"salary")
[1] 55000
```

Аналогичным образом выполняется присваивание. Дадим Джо прибавку к зарплате:

```
> joe@salary <- 65000
> joe
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 65000

Slot "union":
[1] TRUE
```

А впрочем, он заслуживает большего:

```
> slot(joe,"salary") <- 88000
> joe
An object of class "employee"
```

```
Slot "name":
[1] "Joe"

Slot "salary":
[1] 88000

Slot "union":
[1] TRUE
```

Как упоминалось ранее, одним из преимуществ S4 является безопасность. Чтобы продемонстрировать этот факт, предположим, что в записи имени `salary` была допущена опечатка и оно было записано в виде `salry`:

```
> joe@salry <- 48000
Error in checkSlotAssignment(object, name, value) :
  "salry" is not a slot in class "employee"
```

С другой стороны, в S3 сообщений об ошибке не будет. Классы S3 представляют собой обычные списки и в них можно добавлять новые компоненты в любой момент (намеренно или случайно).

9.2.2. Реализация обобщенной функции в классе S4

Для определения реализации обобщенной функции в классе S4 используется метод `setMethod()`. Давайте проделаем это для класса `"employee"`. Мы реализуем функцию `show()` — S4-аналог обобщенной функции S3 `"print"`.

Как вы знаете, в R при вводе имени переменной в интерактивном режиме будет выведено значение этой переменной:

```
> joe
An object of class "employee"
Slot "name":
[1] "Joe"

Slot "salary":
[1] 88000

Slot "union":
[1] TRUE
```

Так как `joe` является объектом S4, в результате будет вызвана функция `show()`. Тот же результат можно получить следующей командой:

```
> show(joe)
```

Переопределим ее в следующем коде:

```
setMethod("show", "employee",
   function(object) {
      inorout <- ifelse(object@union,"is","is not")
      cat(object@name,"has a salary of",object@salary,
         "and",inorout, "in the union", "\n")
   }
)
```

Первый аргумент задает имя обобщенной функции, для которой будет определен метод, специфический для конкретного класса, а второй аргумент задает имя класса. Затем определяется новая функция.

Проверим, как она работает:

```
> joe
Joe has a salary of 55000 and is in the union
```

9.3. S3 и S4

По поводу выбора классов у программистов R мнения расходятся. По сути, ваша точка зрения, скорее всего, будет зависеть от ваших личных предпочтений относительно того, что для вас важнее — удобство S3 или безопасность S4.

Джон Чемберс (John Chambers), создатель языка S и один из главных разработчиков R, в своей книге «Software for Data Analysis» (Springer, 2008) рекомендует использовать классы S4 вместо S3. Он считает, что классы S4 необходимы для написания «понятных и надежных программ». С другой стороны, он замечает, что классы S3 остаются достаточно популярными.

Особый интерес в этом отношении представляет руководство по стилю R от компании Google (http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html). Google однозначно встает на сторону S3, рекомендуя «по возможности избегать объектов и методов S4». (Конечно, интересно уже то, что у Google есть свое руководство по стилю R!)

ПРИМЕЧАНИЕ

Удобное и конкретное сравнение двух методов приведено у Томаса Ламли (Thomas Lumley) «Programmer's Niche: A Simple Class, in S3 and S4», R News, апрель 2004 г., с. 33–36.

9.4. Управление объектами

В ходе типичного сеанса R обычно накапливается большое количество объектов. Существуют разные средства для управления ими. Здесь мы рассмотрим следующие из них:

- функция `ls()`;
- функция `rm()`;
- функция `save()`;
- несколько функций для получения информации о структуре объекта (такие, как `class()` и `mode()`);
- функция `exists()`.

9.4.1. Вывод списка объектов функцией ls()

Команда `ls()` выводит список всех текущих объектов. Полезный именованный аргумент этой функции `pattern` позволяет использовать *шаблоны*. В следующем примере мы приказываем `ls()` вывести только объекты, имена которых соответствуют заданному шаблону:

```
> ls()
 [1] "acc"        "acc05"     "binomci" "cmeans"  "divorg"  "dv"
 [7] "fit"        "g"         "genxc"   "genxnt"  "j"       "lo"
[13] "out1"       "out1.100" "out1.25" "out1.50" "out1.75" "out2"
[19] "out2.100"  "out2.25"  "out2.50" "out2.75" "par.set" "prpdf"
[25] "ratbootci" "simonn"   "vecprod" "x"       "zout"    "zout.100"
[31] "zout.125"  "zout3"    "zout5"   "zout.50" "zout.75"
> ls(pattern="ut")
 [1] "out1"     "out1.100" "out1.25" "out1.50" "out1.75" "out2"
 [7] "out2.100" "out2.25"  "out2.50" "out2.75" "zout"    "zout.100"
[13] "zout.125" "zout3"    "zout5"   "zout.50" "zout.75"
```

Во втором примере запрашиваются имена всех объектов, имена которых включают строку `"ut"`.

9.4.2. Удаление конкретных объектов функцией rm()

Для удаления ненужных объектов используется функция `rm()`. Пример:

```
> rm(a,b,x,y,z,uuu)
```

Этот код удаляет шесть заданных объектов (a, b и т. д.).

Один из именованных аргументов rm() — list — упрощает удаление групп объектов. Следующий код присваивает list все объекты, что приводит к удалению всех объектов:

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

С аргументом pattern функции ls() этот инструмент становится еще более мощным. Пример:

```
> ls()
[1] "doexpt"           "notebookline"   "nreps"          "numcorrectcis"
[5] "numnotebooklines" "numrules"       "observationpt"  "prop"
[9] "r"                "rad"            "radius"         "rep"
[13] "s"               "s2"             "sim"            "waits"
[17] "wbar"            "x"              "y"              "z"
> ls(pattern="notebook")
[1] "notebookline"    "numnotebooklines"
> rm(list=ls(pattern="notebook"))
> ls()
 [1] "doexpt"        "nreps"        "numcorrectcis" "numrules"
 [5] "observationpt" "prop"         "r"             "rad"
 [9] "radius"        "rep"          "s"             "s2"
[13] "sim"           "waits"        "wbar"          "x"
[17] "y"             "z"
```

Здесь мы находим два объекта, имена которых включают строку "notebook", а затем приказываем удалить их. Результат подтверждается вторым вызовом ls().

ПРИМЕЧАНИЕ

Возможно, вам также пригодится функция browseEnv(). Она выводит в веб-браузере список глобальных переменных (или объектов в другом заданном окружении) с дополнительной информации о каждой переменной.

9.4.3. Сохранение коллекции объектов функцией save()

Вызов функции save() для коллекции объектов сохраняет эти объекты на диске для последующей загрузки load(). Пример:

```
> z <- rnorm(100000)
> hz <- hist(z)
> save(hz,file="hzfile")
> ls()
```

```
[1] "hz" "z"
> rm(hz)
> ls()
[1] "z"
> load("hzfile")
> ls()
[1] "hz" "z"
> plot(hz) # Открывается окно с диаграммой
```

Здесь мы генерируем данные, а затем строим гистограмму. При этом вывод `hist()` также сохраняет вывод `hist()` в переменной `hz`. Эта переменная содержит объект (класса `"histogram"`, конечно). Ожидая, что объект будет повторно использоваться позднее в сеансе R, мы вызываем функцию `save()` для сохранения объекта в файле hzfile. После этого объект может быть загружен в будущих сеансах вызовом `load()`. Чтобы продемонстрировать эту возможность, мы намеренно удаляем объект `hz`, после чего снова загружаем его вызовом `load()` и вызываем `ls()`, чтобы убедиться в том, что объект действительно был загружен.

Однажды мне потребовалось загрузить очень большой файл данных, каждая запись которого требовала обработки. Затем я использовал функцию `save()` для сохранения версии объекта обработанного файла для будущих сеансов R.

9.4.4. «Что это такое?»

Разработчикам часто требуется узнать точную структуру объекта, возвращаемого библиотечной функцией. Что делать, если в документации нет необходимых подробностей?

В этом вам могут помочь следующие функции R:

- `class()`, `mode()`;
- `names()`, `attributes()`;
- `unclass()`, `str()`;
- `edit()`.

Рассмотрим пример: в R имеются средства построения факторных таблиц (раздел 6.4). В этом разделе был представлен пример с опросом избирателей, в котором пятерым респондентам предлагалось ответить, собираются ли они голосовать за кандидата X и голосовали ли они за кандидата X на последних выборах. Полученная таблица выглядит так:

```
> cttab <- table(ct)
> cttab
          Voted.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
```

Например, два респондента ответили «нет» на оба вопроса. Объект `cttab` был возвращен функцией `table`, а следовательно, с большой вероятностью относится к классу `"table"`. В этом нетрудно убедиться по документации (`?table`). Но что содержит этот класс?

Исследуем структуру объекта `cttab` класса `"table"`.

```
> ctu <- unclass(cttab)
> ctu
          Votes.for.X.Last.Time
Vote.for.X No Yes
  No        2   0
  Not Sure  0   1
  Yes       1   1
> class(ctu)
[1] "matrix"
```

Итак, часть объекта `counts` представляет собой матрицу. (Если бы данные включали три и более вопроса вместо двух, то использовалась бы матрица большей размерности.) Обратите внимание на имена измерений, а также отдельных строк и столбцов — они связаны с матрицей.

Функция `unclass()` станет хорошей отправной точкой. Просто выводя объект, вы отдаете себя на милость версии `print()`, связанной с классом, которая может ради компактности скрывать или искажать часть ценной информации. Вывод результата вызова `unclass()` позволяет обойти проблему, хотя в данном примере никаких различий нет. (Пример, в котором это было важно, был приведен при описании обобщенных функций S3 в разделе 9.1.1.) Функция `str()` служит той же цели, но в более компактной форме.

Однако следует заметить, что применение `unclass()` к объекту все равно дает объект с некоторым базовым классом. Здесь `cttab` имеет класс `"table"`, но `unclass(cttab)` все равно дает класс `"matrix"`.

Рассмотрим код `table()` — библиотечной функции, которая создает `cttab`. Можно просто ввести `table`, но так как функция довольно длинная, большая часть

кода промелькнет на экране слишком быстро и вы не сможете ее рассмотреть. Проблему можно решить при помощи функции `page()`, но я предпочитаю `edit()`:

```
> edit(table)
```

Это позволит вам просмотреть код в текстовом редакторе. При этом в конце обнаруживается следующий фрагмент:

```
y <- array(tabulate(bin, pd), dims, dimnames = dn)
class(y) <- "table"
y
```

Ага, интересно. Это показывает, что `table()` до определенной степени является оберткой для другой функции `tabulate()`. Но еще важнее то, что структура объекта `"table"` на самом деле проста: объект состоит из массива, созданного на базе счетчиков, к которому прикреплен атрибут `class`. Таким образом, фактически это массив.

Функция `names()` выводит компоненты объекта, а `attributes()` выдает все это и еще больше — прежде всего имя класса.

9.4.5. Функция exists()

Функция `exists()` возвращает `TRUE` или `FALSE` в зависимости от того, существует ли аргумент. Не забудьте заключить аргумент в кавычки.

Например, следующий код проверяет, существует ли объект `acc`:

```
> exists("acc")
[1] TRUE
```

Какую пользу может принести эта функция? Разве мы не всегда знаем, создали мы объект или нет и продолжает ли он существовать? Нет, не всегда. Если вы пишете код общего назначения (например, чтобы опубликовать его через репозиторий кода R CRAN), возможно, ваш код должен будет проверить, существует ли некоторый объект, и если не существует, ваш код должен создать его. Например, как вы узнали в разделе 9.4.3, вы можете сохранить объекты в файле на диске вызовом `save()`, а потом восстановить их в пространстве памяти R вызовом `load()`. Можно написать код общего назначения, который вызывает `load()`, если объект не существует, и использовать для проверки условия вызов `exists()`.

10 Ввод/вывод

Ввод/вывод — одна из тем, которым уделяется явно недостаточно внимания во многих университетских курсах программирования. Ввод/вывод играет центральную роль во многих практических применениях компьютеров. Возьмите хотя бы банкомат, использующий несколько операций ввода/вывода как для ввода (чтение карты и чтение введенного запроса суммы), так и для вывода (вывод инструкций на экран, вывод чека и, самое важное, управление машиной для выдачи денег!).

R — не тот инструмент, который стоило бы применять для управления банкоматом, но он обладает чрезвычайно гибким набором средств ввода/вывода, как вы узнаете в этой главе. Мы начнем с основ работы с клавиатурой и монитором, а затем довольно подробно обсудим чтение и запись файлов, включая перемещение по каталогам. В завершение будут рассмотрены средства R для доступа к интернету.

10.1. Работа с клавиатурой и монитором

R предоставляет несколько функций для работы с клавиатурой и монитором. В этом разделе будут рассмотрены функции `scan()`, `readline()`, `print()` и `cat()`.

10.1.1. Использование функции scan()

Функция `scan()` может использоваться для чтения вектора (числового или символьного) из файла или с клавиатуры. При некоторых дополнительных усилиях данные даже можно прочитать с построением списка.

Допустим, имеются файлы z1.txt, z2.txt, z3.txt и z4.txt. Файл z1.txt содержит следующие данные:

```
123
4 5
6
```

Содержимое файла z2.txt выглядит так:

```
123
4.2 5
6
```

Файл z3.txt содержит следующие данные:

```
abc
de f
g
```

И наконец, файл z4.txt:

```
abc
123 6
y
```

Посмотрим, что можно сделать с этими файлами при помощи функции `scan()`:

```
> scan("z1.txt")
Read 4 items
[1]123 4 5 6
> scan("z2.txt")
Read 4 items
[1] 123.0 4.2 5.0 6.0
> scan("z3.txt")
Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :
  scan() expected 'a real', got 'abc'
> scan("z3.txt",what="")
Read 4 items
[1] "abc" "de" "f" "g"
> scan("z4.txt",what="")
Read 4 items
[1] "abc" "123" "6" "y"
```

При первом вызове будет получен вектор из четырех целых чисел (хотя при этом используется режим `numeric`). Второй раз, поскольку одно число не было целым, другие числа тоже выводились в формате с плавающей точкой.

В третьем случае происходит ошибка. У функции `scan()` есть необязательный аргумент с именем `what`, который определяет режим данных (по умолчанию `double`). По этой причине для нечислового содержимого файла `z3` была выдана ошибка. Но затем была сделана повторная попытка с аргументом `what=""`. Присваивание символьной строки аргументу `what` означает, что для операции запрашивается символьный режим. (Также можно использовать любую символьную строку.)

Последний вызов работает аналогично. Первым элементом является символьная строка, поэтому все последующие элементы также интерпретируются как строки.

Конечно, при типичном применении возвращаемое значение `scan()` будет присвоено переменной. Пример:

```
> v <- scan("z1.txt")
```

По умолчанию `scan()` считает, что элементы вектора разделяются *пропусками*: пробелами, символами возврата курсора / перевода строки и горизонтальными табуляциями. Для других ситуаций можно использовать необязательный аргумент `sep`. Например, можно присвоить `sep` символ новой строки, чтобы каждая строка файла читалась как отдельная символьная строка:

```
> x1 <- scan("z3.txt",what="")
Read 4 items
> x2 <- scan("z3.txt",what="",sep="\n")
Read 3 items
> x1
[1] "abc" "de" "f" "g"
> x2
[1] "abc" "de f" "g"
> x1[2]
[1] "de"
> x2[2]
[1] "de f"
```

В первом случае значения `"de"` и `"f"` были присвоены разным элементам `x1`. Но во втором случае мы указали, что элементы `x2` должны разделяться символами новой строки, а не пробелами. Так как `"de"` и `"f"` находятся в одной строке файла, они вместе присваиваются `x[2]`.

Далее в этой главе будут представлены более сложные методы чтения файлов — например, методы построчного чтения файла. Но если потребуется прочитать весь файл сразу, `scan()` предоставляет простое решение.

Функция `scan()` может использоваться для чтения с клавиатуры; для этого передайте при вызове пустую строку вместо имени файла:

```
> v <- scan("")
1: 12 5 13
4: 3 4 5
7: 8
8:
Read 7 items
> v
[1] 12 5 13 3 4 5 8
```

Теперь функция выдает подсказку с индексом следующего вводимого элемента, а признаком конца ввода является пустая строка.

Если вы не хотите, чтобы функция `scan()` сообщала количество прочитанных элементов, включите аргумент `quiet=TRUE`.

10.1.2. Функция readline()

Для чтения одной строки с клавиатуры можно воспользоваться очень удобной функцией `readline()`:

```
> w <- readline()
abc de f
> w
[1] "abc de f"
```

Как правило, при вызове `readline()` указывается необязательная подсказка:

```
> inits <- readline("type your initials: ")
type your initials: NM
> inits
[1] "NM"
```

10.1.3. Вывод на экран

На верхнем уровне интерактивного режима для вывода значения переменной или выражения достаточно ввести имя этой переменной или выражение. При выводе из тела функции этот способ не сработает. В этом случае можно использовать функцию `print()`:

```
> x<-1:3
> print(x^2)
[1] 1 4 9
```

Напомню, что функция `print()` является обобщенной, поэтому реально вызванная функция будет зависеть от класса выводимого объекта. Если, например, аргумент относится к классу `"table"`, то будет вызвана функция `print.table()`.

Вместо `print()` лучше использовать функцию `cat()`, так как функция `print()` может вывести только одно выражение, а ее вывод нумеруется, что может создать проблемы. Сравните результаты функций:

```
> print("abc")
[1] "abc"
> cat("abc\n")
abc
```

Обратите внимание: при вызове `cat()` необходимо включить символ конца строки `"\n"`. Без него следующий вызов продолжит вывод на той же строке.

Аргументы `cat()` будут выведены с промежуточными пробелами:

```
> x
[1]123
> cat(x,"abc","de\n")
1 2 3 abc de
```

Если пробелы нежелательны, присвойте `sep` пустую строку `""`:

```
> cat(x,"abc","de\n",sep="")
123abcde
```

В аргументе `sep` может использоваться любая строка. В следующем примере передается символ новой строки:

```
> cat(x,"abc","de\n",sep="\n")
1
2
3
abc
de
```

Аргументу `sep` даже можно присвоить вектор строк, как в следующем примере:

```
> x <- c(5,12,13,8,88)
> cat(x,sep=c(".",".",".","\n","\n"))
5.12.13.8
88
```

10.2. Чтение и запись файлов

Разобравшись с азами ввода/вывода, перейдем к более реалистичным примерам чтения и записи файлов. В следующих разделах рассматривается чтение кадров данных или матриц из файлов, работа с текстовыми файлами, обращение к файлам на удаленных компьютерах, а также получение информации о файлах и каталогах.

10.2.1. Чтение кадров данных или матриц из файлов

В разделе 5.1.2 рассматривалось использование функции `read.table()` для чтения кадров данных. На всякий случай напомню: допустим, имеется файл `z` следующего вида:

```
name age
John 25
Mary 28
Jim 19
```

Первая строка содержит необязательный заголовок с именами столбцов. Файл можно прочитать следующим образом:

```
> z <- read.table("z",header=TRUE)
> z
name age
1 John 25
2 Mary 28
3  Jim 19
```

Функция `scan()` в данном случае не подойдет, потому что файл содержит комбинацию числовых и символьных данных (и заголовок).

Казалось бы, прочитать матрицу из файла напрямую не удастся, но задача легко решается при помощи других средств. Простой и быстрый способ основан на использовании `scan()` для чтения матрицы по строкам. Аргумент `byrow` функции `matrix()` сообщает, что элементы матрицы определяются по строкам, а не по столбцам.

Например, допустим, файл `x` содержит матрицу 5 × 3, которая хранится по строкам:

```
1 0 1
1 1 1
1 1 0
1 1 0
0 0 1
```

Данные можно прочитать в матрицу следующим образом:

```
> x <- matrix(scan("x"),nrow=5,byrow=TRUE)
```

Такое решение хорошо подходит для быстрых одноразовых операций, но для универсальности кода можно воспользоваться функцией `read.table()`, которая возвращает кадр данных, а затем преобразовать его `as.matrix()`. Обобщенный метод выглядит так:

```
read.matrix <- function(filename) {
   as.matrix(read.table(filename))
}
```

10.2.2. Чтение текстовых файлов

В компьютерной литературе часто проводится различие между *текстовыми* и *двоичными* файлами. Это различие отчасти условно — каждый файл можно считать двоичным в том смысле, что он состоит из 0 и 1. Будем называть *текстовым файлом* файл, который состоит в основном из символов ASCII или кодировки для другого языка (например, GB для китайского) и использует символы новой строки для того, чтобы пользователь мог понимать, где кончаются строки файла. Последний аспект здесь особенно важен. Нетекстовые файлы, например изображения в формате JPEG или файлы исполняемых программ, обычно называются *двоичными* файлами.

Функция `readLines()` может использоваться для чтения из текстового файла, по одной строке или одним блоком. Предположим, имеется файл `z1` со следующим содержимым:

```
John 25
Mary 28
Jim 19
```

Весь файл можно прочитать за один раз:

```
> z1 <- readLines("z1")
> z1
[1] "John 25" "Mary 28" "Jim 19"
```

Поскольку каждая строка файла интерпретируется как строка данных, возвращаемое значение будет вектором строк, то есть вектором символьного режима. Каждой прочитанной строке файла соответствует один элемент вектора, отсюда три элемента.

Также данные можно прочитать по строкам. Но для этого сначала необходимо создать соединение, как описано ниже.

10.2.3. Соединения

Соединением (connection) в R называется фундаментальный механизм, используемый в различных операциях ввода/вывода. Здесь он будет использоваться для работы с файлами.

Соединение создается вызовом `file()`, `url()` или другой функции R. Чтобы просмотреть список таких функций, введите команду:

```
> ?connection
```

После этого файл z1 (из предыдущего раздела) можно будет прочитать строка за строкой:

```
> c <- file("z1","r")
> readLines(c,n=1)
[1] "John 25"
> readLines(c,n=1)
[1] "Mary 28"
> readLines(c,n=1)
[1] "Jim 19"
> readLines(c,n=1)
character(0)
```

Мы открыли соединение, присвоили результат c, а затем прочитали файл по одной строке, как указывает аргумент n=1. Обнаруживая конец файла (EOF), R возвращает пустой результат. Соединение должно быть настроено так, чтобы при чтении отслеживалась текущая позиция в файле.

Признак конца файла EOF можно проверить в программе:

```
> c <- file("z","r")
> while(TRUE) {
+    rl <- readLines(c,n=1)
+    if (length(rl) == 0) {
+       print("reached the end")
+       break
+    } else print(rl)
+ }
[1] "John 25"
[1] "Mary 28"
[1] "Jim 19"
[1] "reached the end"
```

Если вы хотите вернуться к началу файла, используйте функцию seek():

```
> c <- file("z1","r")
> readLines(c,n=2)
[1] "John 25" "Mary 28"
> seek(con=c,where=0)
[1] 16
> readLines(c,n=1)
[1] "John 25"
```

Аргумент where=0 в вызове seek() означает, что указатель текущей позиции должен быть установлен в 0 символов от начала файла — иначе говоря, в самом начале.

Вызов возвращает 16, это означает, что указатель текущей позиции находился в позиции 16 перед вызовом. Результат вполне логичный: первая строка состоит из символов `"John 25"` и символа конца строки (всего 8 символов); то же самое со второй строкой. Итак, после чтения первых двух строк указатель находится в позиции 16.

Соединение закрывается вызовом `close()`. Закрывая соединение, вы сообщаете системе, что запись в файл завершена и теперь данные должны быть «официально» записаны на диск. Другой пример: в отношениях «клиент/сервер» в интернете (см. раздел 10.3.1) клиент использует `close()` для того, чтобы сообщить серверу о разрыве связи.

10.2.4. Расширенный пример: чтение файлов данных PUMS

Федеральная служба государственной статистики США публикует данные переписи населения в форме PUMS (Public Use Microdata Samples). Термин «микроданные» (microdata) в названии означает, что в файлах содержатся необработанные данные, а каждая запись относится к конкретному человеку (в отличие от статистических сводок). В файл включены данные по огромному количеству переменных.

Данные упорядочены по домашним хозяйствам. Для каждого блока сначала идет запись `Household`, описывающая различные характеристики домашнего хозяйства, после чего следуют записи `Person` для каждого человека. В позициях 106 и 107 (нумерация начинается с 1) записи `Household` задают количество записей `Person` для этого домашнего хозяйства. (Число может быть очень большим, так как некоторые учреждения считаются домашними хозяйствами.)

Для улучшения контроля целостности данных позиция 1 содержит `H` или `P` для подтверждения того, что это запись `Household` или `Person`. Таким образом, если при чтении записи `H` следует, что домашнее хозяйство состоит из трех человек, значит, следующие три записи должны быть записями `P`, а за ними должна следовать новая запись `H`; в противном случае данные содержат ошибку.

Для тестового файла мы возьмем первые 1000 записей 1 % выборки за 2000 год. Первые несколько записей выглядят так:

```
H000019510649 06010 99979997 70 631973
15758 5996765843665000012000000 000000000000 0000
0 0 0 0 0 0 0 0000 0 0 0 0 0 00000000000000000000000000000
00000000000000000000000000
```

```
P000019501000109230004201900101100000101470506002060110999999904200000 0040010000
00300280 28600 70 9997 999720202020202200000400000000000000006000000
00000 00 0000 000000000000000001322410579O4MS 476041-20311010310
0700004901000000000090010000010000010000010000001000000100013901000049000O
H000040710649 06010 99979997 70 631973
15758 599676584365300800200000300106060503010101010102010 0120000600000100001
00600020 0 0 0 0 0000 0 0 0 0 0 020001020101022000000000010750
0232112510000400000040000
P000040701000053010000103800101100000101470304001000090051999012000000 0006010000
00100000 00000 00 0000 000020202020202022000004000000000000000001000060
06010 70 9997 999701010049001000000001018703221 770051-10111010500
400040000000000000000000000000000000000000000000000000000000400000004000O349
P000040702000053030110101400101100000101470500002040040051999012000000 0006010000
00100000 00000 00 0000 000020202020 0 020000000000000000000000000050000
00000 00 0000 000000000000000000000000000000000000000000-00000000000
000 0 0 0 0 0 0 0 0 00000000349
H000061010649 06010 99979997 70 631973
15758 599676584360801190100000200204030502010101010102010 0077000480006400000l
1 0 030 0 0 0 0340 00660000000170 0 0601000000000441003960l000000
000211000000049400000000000
```

Записи очень длинные, поэтому они переносятся по строкам. Каждая запись занимает четыре строки на печатной странице.

Мы напишем функцию `extractpums()`, которая читает файл PUMS и создает кадр данных по записям `Person`. Пользователь задает имя файла, перечисляет извлекаемые поля и имена, которые им должны быть присвоены.

Также нужно сохранить серийный номер домашнего хозяйства. Это полезно, поскольку данные людей из одного домашнего хозяйства могут быть взаимосвязанными; этот аспект необходимо включить в статистическую модель. Кроме того, данные домашних хозяйств могут предоставить важную ковариацию. (И тогда данные ковариации тоже нужно сохранить.)

Прежде чем рассматривать код функции, посмотрим, как работает функция. В этом наборе данных пол хранится в столбце 23, а возраст — в столбцах 25 и 26. В приведенном примере файл называется `pumsa`. Следующий вызов создает кадр данных, состоящий из двух переменных:

```
pumsdf <- extractpums("pumsa",list(Gender=c(23,23),Age=c(25,26)))
```

Также обратите внимание на указание имен столбцов, которые должны присутствовать в итоговом кадре данных. Вы можете использовать любые имена по своему выбору — допустим, `Sex` и `Ancientness` вместо `Gender` и `Age`.

Первая часть кадра данных:

```
> head(pumsdf)
  serno Gender Age
2   195      2  19
3   407      1  38
4   407      1  14
5   610      2  65
6  1609      1  50
7  1609      2  49
```

Ниже приведен код функции `extractpums()`.

```
# Читает файл PUMS pf, извлекает записи Person и возвращает кадр
# данных; каждая строка вывода состоит из серийного номера Household
# и полей в списке flds; имена столбцов кадра данных соответствуют
# индексам flds.

extractpums <- function(pf,flds) {
   dtf <- data.frame() # Создаваемый кадр данных
   con <- file(pf,"r") # Соединение
   # Обработать входной файл
   repeat {
      hrec <- readLines(con,1) # Прочитать запись Household
      if (length(hrec) == 0) break # конец файла, выйти из цикла
      # Получить серийный номер домашнего хозяйства
      serno <- intextract(hrec,c(2,8))
      # Сколько записей Person?
      npr <- intextract(hrec,c(106,107))
      if (npr > 0)
         for (i in 1:npr) {
            prec <- readLines(con,1) # Получить запись Person
            # Создать строку для этого человека
            person <- makerow(serno,prec,flds)
            # Добавить ее в кадр данных
            dtf <- rbind(dtf,person)
         }
   }
   return(dtf)
}

# Создает строку человека для кадра данных
makerow <- function(srn,pr,fl) {
   l <- list()
   l[["serno"]] <- srn
   for (nm in names(fl)) {
      l[[nm]] <- intextract(pr,fl[[nm]])
   }
```

```
36    return(l)
37 }
38
39 # Извлекает из строки s целочисленное поле в позициях
40 # с rng[1] по rng[2]
41 intextract <- function(s,rng) {
42    fld <- substr(s,rng[1],rng[2])
43    return(as.integer(fld))
44 }
```

Давайте разберемся, как работает этот код. В начале `extractpums()` создается пустой кадр данных и соединение для чтения из файла PUMS.

```
dtf <- data.frame() # Создаваемый кадр данных
con <- file(pf,"r") # Соединение
```

Основное тело кода состоит из цикла `repeat`.

```
repeat {
   hrec <- readLines(con,1) # Прочитать запись Household
   if (length(hrec) == 0) break # конец файла, выйти из цикла
   # Получить серийный номер домашнего хозяйства
   serno <- intextract(hrec,c(2,8))
   # Сколько записей Person?
   npr <- intextract(hrec,c(106,107))
   if (npr > 0)
      for (i in 1:npr) {
         ...
      }
}
```

Цикл выполняется до достижения конца входного файла. Это условие проверяется после того, как обнаружена запись `Household` нулевой длины, как видно из приведенного кода.

В цикле `repeat` программа поочередно читает запись `Household` и связанные с ней записи `Person`. Количество записей `Person` для текущей записи `Household` извлекается из столбцов 106 и 107 этой записи; полученное число сохраняется в `npr`. Для извлечения данных используется вызов написанной нами функции `intextract()`.

Затем цикл `for` читает записи `Person` одну за другой, для каждой записи создает строку для выходного кадра данных, а затем присоединяет ее к последнему вызовом `rbind()`:

```
for (i in 1:npr) {
   prec <- readLines(con,1) # Получить запись Person
   # Создать строку для этого человека
```

```
   person <- makerow(serno,prec,flds)
      # Добавить ее в кадр данных
      dtf <- rbind(dtf,person)
   }
```

Обратите внимание на то, как функция `makerow()` создает строку, добавляемую в кадр данных для человека. В формальных аргументах функция получает серийный номер домашнего хозяйства (`srn`), запись `Person` (`pr`) и список имен переменных и полей столбцов (`fl`):

```
makerow <- function(srn,pr,fl) {
   l <- list()
   l[["serno"]] <- srn
   for (nm in names(fl)) {
      l[[nm]] <- intextract(pr,fl[[nm]])
   }
   return(l)
}
```

Для примера рассмотрим следующий вызов:

```
pumsdf <- extractpums("pumsa",list(Gender=c(23,23),Age=c(25,26)))
```

При выполнении `makerow()` `fl` будет представлять собой список из двух элементов с именами `Gender` и `Age`. У строки `pr` (текущая запись `Person`) значение `Gender` содержится в столбце 23, а значение `Age` — в столбцах 25 и 26. Для извлечения нужных чисел вызывается функция `intextract()`.

Функция `intextract()` выполняет тривиальное преобразование символов в числа — например, строка `"12"` преобразуется в число 12. Обратите внимание: если бы не присутствие записей `Household`, то же самое можно было бы гораздо проще сделать при помощи удобной встроенной функции R `read.fwf()`. Имя функции является сокращением от «чтение фиксированной ширины с форматированием» — оно говорит о том, что каждая переменная хранится в конкретных позициях символов записи. По сути, функция избавляет вас от необходимости самостоятельно писать функции, подобные `intextract()`.

10.2.5. Обращение к файлам на удаленных машинах по URL-адресам

Некоторые функции ввода/вывода, такие как `read.table()` и `scan()`, получают в аргументах URL-адреса. (Если вы захотите узнать, поддерживает ли ваша любимая функция такую возможность, обращайтесь к электронной документации R.)

В качестве примера прочитаем данные из архива Калифорнийского университета в Ирвайне (http://archive.ics.uci.edu/ml/datasets.html) из набора данных эхокардиограмм. После перехода по ссылкам мы находим этот файл и читаем его из R:

```
> uci <- "http://archive.ics.uci.edu/ml/machine-learning-databases/"
> uci <- paste(uci,"echocardiogram/echocardiogram.data",sep="")
> ecc <- read.csv(uci)
```

(URL-адрес строится поэтапно, чтобы код помещался на странице.)

А теперь посмотрим, что было загружено:

```
> head(ecc)
  X11 X0 X71 X0.1 X0.260     X9 X4.600  X14    X1  X1.1 name X1.2 X0.2
1  19  0  72    0  0.380      6  4.100   14 1.700 0.588 name    1    0
2  16  0  55    0  0.260      4  3.420   14     1     1 name    1    0
3  57  0  60    0  0.253 12.062  4.603   16 1.450 0.788 name    1    0
4  19  1  57    0  0.160     22  5.750   18 2.250 0.571 name    1    0
5  26  0  68    0  0.260      5  4.310   12     1 0.857 name    1    0
6  13  0  62    0  0.230     31  5.430 22.5 1.875 0.857 name    1    0
```

Теперь можно перейти к анализу. Например, для возраста в третьем столбце можно вычислить среднее значение или произвести другие вычисления. За описаниями всех переменных обращайтесь на страницу набора данных по адресу http://archive.ics.uci.edu/ml/machine-learning-databases/echocardiogram/echocardiogram.names.

10.2.6. Запись в файл

R прежде всего применяется для статистического анализа, поэтому операции чтения выполняются намного чаще операций записи. Однако время от времени надобность в записи тоже возникает, и в этом разделе будут представлены методы записи файлов.

Функция `write.table()` очень похожа на `read.table()`, если не считать того, что она выполняет запись кадра данных вместо чтения. Например, возьмем маленький пример из начала главы 5:

```
> kids <- c("Jack","Jill")
> ages <- c(12,10)
> d <- data.frame(kids,ages,stringsAsFactors=FALSE)
> d
  kids ages
1 Jack   12
2 Jill   10
> write.table(d,"kds")
```

Теперь файл kds будет содержать следующие данные:

```
"kids" "ages"
"1" "Jack" 12
"2" "Jill" 10
```

В случае записи матрицы в файл просто укажите, что имена строк и столбцов не нужны:

```
> write.table(xc,"xcnew",row.names=FALSE,col.names=FALSE)
```

Функция `cat()` также может использоваться для пошаговой записи в файл. Пример:

```
> cat("abc\n",file="u")
> cat("de\n",file="u",append=TRUE)
```

Первый вызов `cat()` создает файл u, состоящий из одной строки с содержимым `"abc"`. Второй вызов присоединяет вторую строку. В отличие от примера с функцией `writeLines()` (которые будут рассмотрены ниже), файл автоматически сохраняется после каждой операции. Например, после предыдущих вызовов файл будет выглядеть так:

```
abc
de
```

Также можно записать сразу несколько полей. Следовательно, команда

```
> cat(file="v",1,2,"xyz\n")
```

создаст файл v, содержащий всего одну строку:

```
1 2 xyz
```

Также можно использовать `writeLines()`, парную для `readLines()`. При использовании соединения необходимо передать аргумент `"w"`, чтобы показать, что выполняется запись в файл, а не чтение:

```
> c <- file("www","w")
> writeLines(c("abc","de","f"),c)
> close(c)
```

Создается файл www со следующим содержимым:

```
abc
de
f
```

Обратите внимание: файл должен быть явно закрыт вызовом `close()`.

10.2.7. Получение информации о файлах и каталогах

В R существует несколько функций для получения информации о каталогах и файлах, назначения разрешений доступа к файлам и т. д. Несколько примеров:

- `file.info()` — выдает размер файла, время создания, флаг каталога / обычного файла и т. д. для каждого файла, имя которого содержится в аргументе (символьный вектор).
- `dir()` — возвращает символьный вектор с именами всех файлов в каталоге, заданном первым аргументом. Если задан необязательный аргумент `recursive=TRUE`, в результат будет включено все дерево каталогов с корнем в первом аргументе.
- `file.exists()` — возвращает логический вектор, который сообщает, существует ли файл для каждого имени в первом аргументе (символьном векторе).
- `getwd()` и `setwd()` — используется для определения или изменения текущего рабочего каталога.

Чтобы просмотреть список всех функций, относящихся к работе с файлами и каталогами, введите команду:

```
> ?files
```

Некоторые функции из этого списка будут продемонстрированы в следующем примере.

10.2.8. Расширенный пример: суммирование содержимого многих файлов

В этом разделе мы разработаем функцию суммирования содержимого (числового, как предполагается) для всех файлов в дереве каталогов. В нашем примере каталог dir1 содержит файлы filea и fileb, а также подкаталог dir2 с файлом filec. Файлы содержат следующие данные:

- filea: 5, 12, 13;
- fileb: 3, 4, 5;
- filec: 24, 25, 7.

Если dir1 является текущим каталогом, то вызов `sumtree("dir1")` выдаст сумму этих девяти чисел (98). В противном случае необходимо указать полный путь dir1, такой как `sumtree("/home/nm/dir1")`. Код функции:

```
 1 sumtree <- function(drtr) {
 2    tot <- 0
 3    # Получить имена всех файлов в дереве
 4    fls <- dir(drtr,recursive=TRUE)
 5    for (f in fls) {
 6       # f является каталогом?
 7       f <- file.path(drtr,f)
 8       if (!file.info(f)$isdir) {
 9          tot <- tot + sum(scan(f,quiet=TRUE))
10       }
11    }
12    return(tot)
13 }
```

Эта задача естественно решается посредством рекурсии (см. раздел 7.9). Однако здесь R выполняет рекурсию за вас, позволяя включить соответствующий режим при помощи аргумента `dir()`. Таким образом, в строке 4 мы присваиваем `recursive=TRUE`, чтобы найти все файлы на разных уровнях дерева каталогов.

Чтобы вызвать `file.info()`, необходимо принять во внимание тот факт, что текущее имя файла `f` задается относительно `drtr`, так что файл filea будет определяться в виде dir1/filea. Чтобы сформировать это имя файла, необходимо объединить несколько компонентов — `drtr`, символ `/` и `filea`. Для этого можно воспользоваться функцией конкатенации R `paste()`, но для Windows понадобится отдельное условие, в котором используется символ \ вместо /. Однако `file.path()` делает все это за нас.

Стоит сказать несколько слов о строке 8. Функция `file.info()` возвращает информацию о `f` в виде кадра данных, одним из столбцов которого является `isdir`, каждому файлу соответствует строка, а имена строк соответствуют именам файлов. Этот столбец состоит из логических значений, указывающих, является ли каждый файл каталогом. Таким образом, в строке 8 мы определяем, является ли текущий файл `f` каталогом. Если `f` является обычным файлом, то его содержимое просто добавляется в накапливаемую сумму.

10.3. Доступ в интернет

Средства для работы с сокетами R открывают программисту доступ к протоколу интернета TCP/IP. Для читателей, не знакомых с этим протоколом, мы начнем с обзора TCP/IP.

10.3.1. Обзор TCP/IP

Протокол TCP/IP достаточно сложен, поэтому этот краткий обзор будет весьма упрощенным. Тем не менее приведенного материала будет достаточно для того, чтобы вы поняли, что делают функции сокетов R.

В контексте нашего обсуждения термином *сеть* обозначается совокупность компьютеров, объединенных локальными каналами связи без выхода в интернет. Обычно в сеть связываются все компьютеры в доме, все компьютеры малого предприятия и т. д. Физической средой передачи информации между ними обычно является канал связи Ethernet того или иного типа.

Интернет связывает между собой разные сети. Сеть в интернете соединяется с одной или несколькими другими сетями при помощи *маршрутизаторов* — специальных компьютеров, соединяющих две и более сети. Каждому компьютеру в интернете назначается IP-адрес (IP = Internet Protocol). Адрес имеет числовую форму, но он также может быть представлен в символьном виде — например, www.google.com; эта форма преобразуется в числовой адрес серверами DNS (Domain Name Service).

Тем не менее одного IP-адреса недостаточно. Когда компьютер A отправляет сообщение B, на компьютере B могут существовать различные приложения, получающие сообщения из интернета, — браузеры, электронная почта и т. д. Как операционная система B узнает, кому из них передать сообщения от A? Для этого компьютер A кроме IP-адреса указывает номер *порта*. Номер порта указывает, какая программа, работающая на компьютере B, должна быть получателем сообщения. У компьютера A тоже есть номер порта, чтобы ответ от B был получен правильным приложением на A.

Когда A желает отправить какие-либо данные B, он записывает информацию в программную абстракцию, которая называется *сокетом* (socket), с использованием системной функции, сходной с функцией записи файла. В вызове компьютер A указывает IP-адрес и номер порта компьютера B, на который A желает отправить сообщение. У компьютера B тоже имеется сокет, и он записывает свои ответы для A в этот сокет. Мы говорим, что A и B *соединены* через эти сокеты, но за этой формулировкой нет никакого физического смысла — это всего лишь соглашение между A и B по обмену данными.

Приложения работают по модели *клиент/сервер*. Допустим, веб-сервер работает на компьютере B на стандартном для WWW порте 80. Сервер на компьютере B ведет *прослушивание* на порте 80. И снова термин не следует воспринимать буквально; он всего лишь означает, что серверная программа вызвала функцию, которая оповещает операционную систему о том, что программа-сервер желает

обмениваться информацией через порт 80. Когда сетевой узел A запрашивает такое соединение, вызов функции на сервере возвращает управление и между компьютерами устанавливается соединение.

Если вы в качестве непривилегированного пользователя пишете некую серверную программу (допустим, на R!), ей должен быть назначен номер порта больше 1024.

ПРИМЕЧАНИЕ

Если серверная программа выходит из строя или аварийно завершается, может пройти несколько секунд, чтобы тот же порт снова стал доступен для использования.

10.3.2. Сокеты в R

Очень важно учитывать, что все байты, передаваемые между компьютерами A и B во время существования соединения между ними, рассматриваются как одно большое сообщение. Допустим, A отправляет одну строку текста из 8 символов, а затем еще одну строку из 20 символов. С точки зрения A есть две строки, но для TCP/IP это всего лишь 28 символов все еще не завершенного сообщения. Разбиение длинного сообщения на строки может потребовать некоторых усилий. R предоставляет для этой цели различные функции, включая следующие:

- `readLines()` и `writeLines()` — позволяют программировать так, словно TCP/IP отправляет сообщения по строкам, несмотря на то что в действительности это не так. Если ваше приложение естественно рассматривать в контексте передачи строк, эти две функции будут весьма удобными.
- `serialize()` и `unserialize()` — эти функции могут использоваться для отправки объектов R — например, матриц или сложных результатов вызова статистических функций. Объект преобразуется в форму символьной строки отправителем, а затем снова преобразуется к форме исходного объекта получателем.
- `readBin()` и `writeBin()` — функции для отправки данных в двоичной форме. (См. комментарий по поводу терминологии в начале раздела 10.2.2.)

Все эти функции работают с соединениями R, как будет показано в следующем примере.

Очень важно выбрать правильную функцию для каждой ситуации. Например, с длинными векторами функции `serialize()` и `unserialize()` могут быть более

удобными, но они занимают гораздо больше времени. Дело не только в том, что числа приходится преобразовывать в символьные представления и обратно; символьное представление занимает намного больше памяти, что означает увеличение времени передачи.

В R есть еще две функции сокетов:

- `socketConnection()` — создает соединение R через сокеты. Номер порта задается в аргументе `port`, а чтобы выбрать между созданием клиента или сервера, присвойте аргументу `server` значение `TRUE` или `FALSE` соответственно. Для клиента также необходимо задать IP-адрес сервера в аргументе `host`.
- `socketSelect()` — используется при соединении сервера с несколькими клиентами. Главный аргумент `socklist` содержит список соединений, а возвращаемое значение — подсписок соединений, у которых имеются готовые данные для чтения сервером.

10.3.3. Расширенный пример: параллелизм в R

Некоторые задачи статистического анализа выполняются очень долго, поэтому возник вполне естественный интерес к *параллелизму* в R — совместному выполнению задачи несколькими процессами R. Другая возможная причина для параллелизма — ограничения памяти. Если одна машина не располагает достаточным объемом памяти для текущей задачи, возможно, удастся каким-то образом объединить память нескольких машин. В главе 16 приведен вводный курс по этой важной теме.

Сокеты играют ключевую роль во многих параллельных пакетах R. Взаимодействующие процессы R могут работать на одной или на разных машинах. В последнем случае (и даже в первом) естественный подход к реализации параллелизма основан на использовании сокетов R. Это одно из решений, принятых в пакете `snow` и в моем пакете `Rdsm` (оба пакета доступны в CRAN, репозитории кода R; за подробностями обращайтесь к приложению):

- В `snow` сервер отправляет клиентам задачи, то есть единицы работы. Клиенты выполняют свои задачи и возвращают результаты серверу, который объединяет их в итоговый результат. Взаимодействия осуществляются функциями `serialize()` и `unserialize()`, а сервер использует `socketSelect()` для определения клиентов, у которых имеются готовые результаты.
- `Rdsm` реализует парадигму виртуальной общей памяти, а сервер используется для хранения общих переменных. Клиент связывается с сервером каждый раз, когда ему потребуется прочитать или записать общую пере-

менную. Для оптимизации скорости взаимодействие между сервером и клиентами осуществляется функциями `readBin()` и `writebin()` вместо `serialize()` и `unserialize()`.

Рассмотрим некоторые подробности Rdsm, относящиеся к работе с сокетами. Ниже приведен код сервера, который создает соединения с клиентами и сохраняет их в списке `cons` (для `ncon` клиентов):

```
# Готовит соединения с клиентами через сокеты
#
cons <<- vector(mode="list",length=ncon) # Список соединений
# Чтобы предотвратить потерю соединения при отладке или долгом ожидании
options("timeout"=10000)
for (i in 1:ncon) {
   cons[[i]] <<-
      socketConnection(port=port,server=TRUE,blocking=TRUE,open="a+b")
   # Ожидать ответа от клиента i
   checkin <- unserialize(cons[[i]])
}
# Отправить подтверждения
for (i in 1:ncon) {
   # Отправить клиенту его идентификатор и размер группы
   serialize(c(i,ncon),cons[[i]])
}
```

Так как сообщения клиента и подтверждения сервера имеют небольшую длину, `serialize()` и `unserialize()` достаточно хорошо подходят для этой цели.

Первая часть основного цикла сервера находит готового клиента и читает у него данные.

```
repeat {
   # Клиенты еще остались?
   if (remainingclients == 0) break
   # Ожидать запрос на обслуживание и прочитать его
   # Найти все незавершенные запросы от клиентов
   rdy <- which(socketSelect(cons))
   # Выбрать один из них
   j <- sample(1:length(rdy),1)
   con <- cons[[rdy[j]]]
   # Прочитать запрос клиента
   req <- unserialize(con)
```

И снова функции `serialize()` и `unserialize()` достаточно хорошо подходят для чтения короткого сообщения от клиента, которое обозначает тип запрашиваемой операции — обычно чтение или запись общей переменной. Однако

для чтения и записи самих общих переменных используются более быстрые операции `readBin()` и `writeBin()`. Часть, относящаяся к записи:

```
# Записывает данные dt режима md (integer или double) в соединение cn
binwrite <- function(dt,md,cn) {
writeBin(dt,con=cn)
```

А вот часть для чтения:

```
# Читает sz элементов режима md (integer или double) из соединения cn
binread <- function(cn,md,sz) {
return(readBin(con=cn,what=md,n=sz))
```

На стороне клиента код создания соединения выглядит так:

```
1 options("timeout"=10000)
2 # Создать соединение с сервером
3 con <- socketConnection(host=host,port=port,blocking=TRUE,open="a+b")
4 serialize(list(req="checking in"),con)
5 # Получить идентификатор клиента и общее число клиентов от сервера
6 myidandnclnt <- unserialize(con)
7 myinfo <<-
8    list(con=con,myid=myidandnclnt[1],nclnt=myidandnclnt[2])
```

Код чтения и записи данных на сервер аналогичен приведенным выше примерам для сервера.

11 Работа со строками

Хотя R является статистическим языком и числовые векторы с матрицами играют в нем центральную роль, символьные строки также, на удивление, важны. Символьные данные очень часто встречаются в программах R, от дат рождения, хранимых в файлах данных медицинских исследований, до глубокого анализа текстов. По этой причине в R реализованы многочисленные средства для работы со строками; некоторые из них будут представлены в этой главе.

11.1. Обзор функций для работы со строками

Здесь будут кратко рассмотрены некоторые строковые функции, существующие в R. Учтите, что в этом введении будут представлены очень простые формы вызовов, из которых обычно исключаются многие необязательные аргументы. Некоторые из них будут использованы в расширенных примерах этой главы, а за более подробной информацией обращайтесь к электронной документации R.

11.1.1. grep()

Вызов `grep(pattern,x)` ищет заданную подстроку `pattern` в векторе `x` со строковыми элементами. Если `x` содержит `n` элементов, то есть `n` строк, то `grep(pattern,x)` вернет вектор длиной `n`. Каждый элемент этого вектора содержит индекс позиции вектора `x`, в которой было найдено совпадение `pattern` как подстроки `x[i]`.

Пример использования `grep`:

```
> grep("Pole",c("Equator","North Pole","South Pole"))
[1]23
> grep("pole",c("Equator","North Pole","South Pole"))
integer(0)
```

В первом случае строка "Pole" была обнаружена в элементах 2 и 3 второго аргумента, отсюда вывод (2, 3). Во втором случае строка "pole" не была найдена, поэтому возвращается пустой вектор.

11.1.2. nchar()

Вызов nchar(x) находит длину строки x. Пример:

```
> nchar("South Pole")
[1] 10
```

Строка "South Pole" содержит 10 символов. Вниманию программистов C: строки R не завершаются NULL-символом. Также следует учитывать, что результаты nchar() будут непредсказуемыми, если режим x отличен от символьного. Например, значение nchar(NA) оказывается равным 2, а результат nchar(factor("abc")) равен 1. Чтобы получать более последовательные результаты для нестроковых объектов, используйте пакет stringr Хэдли Уикхэма (Hadley Wickham) из репозитория CRAN.

11.1.3. paste()

Вызов paste(...) осуществляет конкатенацию нескольких строк и возвращает результат — одну длинную строку. Несколько примеров:

```
> paste("North","Pole")
[1] "North Pole"
> paste("North","Pole",sep="")
[1] "NorthPole"
> paste("North","Pole",sep=".")
[1] "North.Pole"
> paste("North","and","South","Poles")
[1] "North and South Poles"
```

Как вы видите, необязательный аргумент sep позволяет использовать для разделения фрагментов другие символы, кроме пробела. Если присвоить sep пустую строку, между фрагментами не будет никаких дополнительных символов.

11.1.4. sprintf()

Функция sprintf(...) собирает из фрагментов отформатированную строку.

Простой пример:

```
> i <- 8
> s <- sprintf("the square of %d is %d",i,i^2)
> s
[1] "the square of 8 is 64"
```

Имя функции происходит от «string print», то есть ассоциируется с выводом в строку, а не на экран. Здесь результат выводится в строку `s`.

Что именно выводится? Функция сначала выводит строку `"the square of"`, а затем десятичное значение `i` (то есть значение, записанное в системе счисления с основанием 10). В итоге будет выведена строка `"the square of 8 is 64"`.

11.1.5. substr()

Вызов `substr(x,start,stop)` возвращает подстроку из заданного диапазона позиций символов `start:stop` строки `x`. Пример:

```
> substring("Equator",3,5)
[1] "uat"
```

11.1.6. strsplit()

Вызов `strsplit(x,split)` разбивает строку `x` на список подстрок, разделенных другой строкой `split`. Пример:

```
> strsplit("6-16-2011",split="-")
[[1]]
[1] "6" "16" "2011"
```

11.1.7. regexpr()

Вызов `regexpr(pattern,text)` находит позицию первого вхождения `pattern` в тексте, как в следующем примере:

```
> regexpr("uat","Equator")
[1] 3
```

Функция сообщает, что `"uat"` действительно встречается в `"Equator"`, начиная с позиции символа 3.

11.1.8. gregexpr()

Вызов `gregexpr(pattern,text)` аналогичен `regexpr()`, но он находит все вхождения `pattern`. Пример:

```
> gregexpr("iss","Mississippi")
[[1]]
[1]25
```

Вызов функции обнаруживает, что "iss" дважды встречается в строке "Mississipp", начиная с позиций символов 2 и 5.

11.2. Регулярные выражения

При использовании функций для работы со строками в языках программирования иногда встречается понятие *регулярных выражений*. В R это обстоятельство должно учитываться при использовании строковых функций grep(), grepl(), regexpr(), gregexpr(), sub(), gsub() и strsplit().

Регулярное выражение представляет собой своеобразный шаблон. Это сокращенный синтаксис для определения широких классов строк. Например, выражение "[au]" соответствует любой строке, содержащей хотя бы один из символов, a или u. Пример его использования:

```
> grep("[au]",c("Equator","North Pole","South Pole"))
[1] 1 3
```

Функция сообщает, что элементы 1 и 3 списка ("Equator","North Pole","South Pole"), то есть "Equator" и "South Pole" содержат a или u.

Точка (.) представляет любой символ. Пример использования:

```
> grep("o.e",c("Equator","North Pole","South Pole"))
[1] 2 3
```

Здесь функция ищет строки из трех символов, в которых за o следует один любой символ, а потом идет символ e. Пример использования двух точек для представления произвольной пары символов:

```
> grep("N..t",c("Equator","North Pole","South Pole"))
[1] 2
```

Здесь мы ищем четырехбуквенные строки, состоящие из символа N, за которым следует произвольная пара символов, после чего идет символ t.

Точка является примером *метасимвола*, то есть символа, который не должен восприниматься буквально. Например, если точка встречается в первом аргументе grep(), она в действительности обозначает не точку, а любой символ.

Что делать, если вам потребуется найти точку вызовом grep()? Наивная попытка выглядит так:

```
> grep(".",c("abc","de","f.g"))
[1] 1 2 3
```

Результатом должно быть значение 3, а не `(1,2,3)`. Попытка завершилась неудачей, потому что точки являются метасимволами. Точку необходимо экранировать, то есть подавить ее метасимвольную интерпретацию; для этой цели применяется символ \:

```
> grep("\\.",c("abc","de","f.g"))
[1] 3
```

Разве я не сказал «символ \»? Почему здесь стоят два символа? Дело в том, что сам символ \ тоже необходимо экранировать, а для этого нужен дополнительный символ \! Как видите, регулярные выражения могут стать невероятно сложными. По теме регулярных выражений было написано немало книг (для различных языков программирования). На первых порах изучения темы обращайтесь к электронной документации R (введите команду `?regex`).

11.2.1. Расширенный пример: проверка имени файла на наличие определенного суффикса

Предположим, требуется протестировать имя файла на наличие заданного суффикса — например, чтобы найти все файлы HTML (файлы с суффиксами .html, .htm и т. д.). Код выглядит так:

```
1 testsuffix <- function(fn,suff) {
2    parts <- strsplit(fn,".",fixed=TRUE)
3    nparts <- length(parts[[1]])
4    return(parts[[1]][nparts] == suff)
5 }
```

Протестируем его:

```
> testsuffix("x.abc","abc")
[1] TRUE
> testsuffix("x.abc","ac")
[1] FALSE
> testsuffix("x.y.abc","ac")
[1] FALSE
> testsuffix("x.y.abc","abc")
[1] TRUE
```

Как работает эта функция? Для начала заметим, что вызов `strsplit()` в строке 2 возвращает список с одним элементом (потому что `fn` — одноэлементный

вектор) — вектором строк. Например, вызов `testsuffix("x.y.abc","abc")` приведет к тому, что `parts` будет представлять собой список, состоящий из трехэлементного вектора с элементами `x`, `y` и `abc`. Затем код извлекает последний элемент и сравнивает его с `suff`.

Важнейшим аспектом этого кода является аргумент `fixed=TRUE`. Без него аргумент разбивки `.` (в списке формальных аргументов `strsplit()` ему присвоено имя `split`) интерпретировался бы как регулярное выражение. Без присваивания `fixed=TRUE` функция `strsplit()` просто разделила бы все на буквы.

Конечно, с таким же успехом точку можно было бы экранировать:

```
1 testsuffix <- function(fn,suff) {
2    parts <- strsplit(fn,"\\.")
3    nparts <- length(parts[[1]])
4    return(parts[[1]][nparts] == suff)
5 }
```

Убедимся в том, что этот код работает.

```
> testsuffix("x.y.abc","abc")
[1] TRUE
```

А вот еще один вариант кода проверки суффикса — чуть более сложный, но полезный в учебных целях:

```
1 testsuffix <- function(fn,suff) {
2    ncf <- nchar(fn) # nchar() выдает длину строки
3    # Определить, где будет находиться точка, если suff – суффикс fn
4    dotpos <- ncf – nchar(suff) + 1
5    # Теперь проверить наличие суффикса
6    return(substr(fn,dotpos,ncf)==suff)
7 }
```

Рассмотрим вызов `substr()` с `fn="x.ac"` и `suff="abc"`. В этом случае значение `dotpos` будет равно 1; это означает, что с суффиксом `.abc` точка должна находиться в первом символе `fn`. Вызов `substr()` принимает вид `substr("x.ac",1,4)`; он извлекает подстроку в позициях символов с 1-й по 4-ю строки `x.ac`. Подстрока `x.ac` не совпадает с `abc`, и суффикс имени файла отличается от заданного.

11.2.2. Расширенный пример: формирование имен файлов

Предположим, вы хотите создать пять файлов от `q1.pdf` до `q5.pdf`, которые содержат гистограммы 100 случайных величин $N(0,i^2)$. Для этого можно выполнить следующий код:

```
1 for (i in 1:5) {
2    fname <- paste("q",i,".pdf")
3    pdf(fname)
4    hist(rnorm(100,sd=i))
5    dev.off()
6 }
```

В этом примере особенно важна строковая операция, используемая для создания имени файла `fname`. За дополнительной информацией о графических операциях, использованных в этом примере, обращайтесь к разделу 12.3.

Функция `paste()` объединяет конкатенацией строку `"q"` со строковой формой числа `i`. Например, при `i = 2` переменная `fname` будет равна `q 2 .pdf`. Тем не менее это не совсем то, что нам нужно. В системах Linux внутренние пробелы в именах файлов создают проблемы, поэтому пробелы нужно удалить. Одно из решений основано на использовании аргумента `sep`, определяющего пустую строку в качестве разделителя:

```
1 for (i in 1:5) {
2    fname <- paste("q",i,".pdf",sep="")
3    pdf(fname)
4    hist(rnorm(100,sd=i))
5    dev.off()
6 }
```

Другое решение основано на использовании функции `sprintf()`, позаимствованной из C:

```
1 for (i in 1:5) {
2    fname <- sprintf("q%d.pdf",i)
3    pdf(fname)
4    hist(rnorm(100,sd=i))
5    dev.off()
6 }
```

Для чисел с плавающей точкой также обратите внимание на различия между форматами `%f` и `%g`:

```
> sprintf("abc%fdef",1.5)
[1] "abc1.500000def"
> sprintf("abc%gdef",1.5)
[1] "abc1.5def"
```

Формат `%g` устраняет лишние нули.

11.3. Применение строковых функций в режиме отладки edtdbg

Во внутреннем коде программы отладки edtdbg, которая будет рассматриваться в разделе 13.4, широко используются операции со строками. Типичный пример такого применения встречается в функции dgbsendeditcmd():

```
# Отправить команду редактору

dbgsendeditcmd <- function(cmd) {
   syscmd <- paste("vim --remote-send ",cmd," --servername ",vimserver,sep="")
   system(syscmd)
}
```

Что здесь происходит? Главное — то, что edtdbg отправляет удаленные команды текстовому редактору Vim. Например, если Vim работает на сервере с именем 168 и вы хотите переместить курсор в Vim в строку 12, введите в окне терминала следующую команду:

```
vim --remote-send 12G --servername 168
```

Эффект будет таким же, как если бы вы ввели физическую команду 12G в окне Vim. Так как 12G — команда Vim для перемещения курсора в строку 12, именно это и произойдет. Рассмотрим следующий вызов:

```
paste("vim --remote-send ",cmd," --servername ",vimserver,sep="")
```

Здесь cmd содержит строку "12G", значение vimserver равно 168, а paste() объединяет все указанные строки посредством конкатенации. Аргумент sep="" включает использование пустой строки в качестве разделителя при конкатенации (то есть без разделителей). Таким образом, paste() возвращает следующий результат:

```
vim --remote-send 12G --servername 168
```

Другой важнейший аспект работы edtdbg заключается в том, что программа записала (при помощи функции R sink()) в файл dbgsink большую часть вывода отладчика R в окне R. (Программа edtdbg работает в сочетании с этим отладчиком.) Эта информация включает номера строк для ваших позиций в исходном файле при пошаговом перемещении по нему в отладчике R.

Позиционная информация в выводе отладчика выглядит примерно так:

```
debug at cities.r#16: {
```

В edtdbg присутствует код для определения последней строки dbgsink, начинающейся с «debug at». Эта строка файла помещается в переменную с именем

debugline. Следующий код извлекает номер строки (16 в данном примере) и имя исходного файла / буфера Vim (в данном примере cities.r):

```
linenumstart <- regexpr("#",debugline) + 1
buffname <- substr(debugline,10,linenumstart-2)
colon <- regexpr(":",debugline)
linenum <- substr(debugline,linenumstart,colon-1)
```

Вызов regexpr() определяет, в какой позиции debugline находится символ # (символ 18 в данном примере). Увеличение на 1 дает позицию номера строки внутри debugline.

Чтобы получить имя буфера, взяв за основу предыдущий пример, мы видим, что имя следует после debug at и завершается непосредственно перед #. Так как «debug at» содержит 9 символов, имя буфера будет начинаться в позиции 10 — отсюда 10 в вызове:

```
substr(debugline,10,linenumstart-2)
```

Конец поля с именем буфера находится в позиции linenumstart-2, так как он непосредственно предшествует символу # перед началом номера строки. Номер строки определяется аналогичным образом.

Другой характерный пример внутреннего кода edtdbg связан с использованием функции strsplit(). Например, в определенный момент она выводит приглашение для пользователя:

```
kbdin <- readline(prompt="enter number(s) of fns you wish to toggle dbg: ")
```

Как видно из кода, ответ пользователя сохраняется в kbdin. Он состоит из нескольких чисел, разделенных пробелами:

```
1 4 5
```

Числа из строки 1 4 5 нужно извлечь в целочисленный вектор. Для этого сначала вызывается функция strsplit(), которая создает три строки: "1", "4" и "5". Затем вызов as.integer() преобразует символьные данные в числовые:

```
tognums <- as.integer(strsplit(kbdin,split=" ")[[1]])
```

Вывод strsplit() представляет собой список R; в данном случае список состоит всего из одного элемента, который, в свою очередь, является вектором ("1","4","5"); отсюда и выражение [[1]] в нашем примере.

12 Графика

R обладает чрезвычайно богатым арсеналом графических средств. Несколько ярких примеров приведено на домашней странице R (http://www.r-project.org/), но чтобы в полной мере оценить графические возможности R, просмотрите галерею R Graph Gallery по адресу www.r-graph-gallery.com.

В этой главе рассматриваются основы использования базового (или традиционного) графического пакета R. С этим материалом вы сможете начать работать с графикой в R. Если вам захочется глубже изучить графику R, на эту тему написано немало превосходных книг.

12.1. Построение графиков

Для начала рассмотрим традиционную функцию создания графиков: `plot()`. Затем вы узнаете, как дополнить график новыми элементами: от линий и точек до условных обозначений.

12.1.1. Основная функция базовой графики R: plot()

Функция `plot()` образует фундамент многих базовых графических операций R и служит основой для построения разнообразных видов графиков. Как упоминалось в разделе 9.1.1, `plot()` является обобщенной функцией, то есть «временным представителем», для целого семейства функций. Фактически вызываемая функция зависит от класса объекта, для которого она вызывается.

Посмотрим, что произойдет при вызове `plot()` для векторов `X` и `Y`, которые интерпретируются как множество пар точек на плоскости (x, y).

```
> plot(c(1,2,3), c(1,2,4))
```

На экране появляется окно с точками (1, 1), (2, 2) и (3, 4) (рис. 12.1). Как видите, у нас получился предельно простой график. Позднее в этой главе вы узнаете, как добавить к нему всевозможные украшения и декоративные элементы.

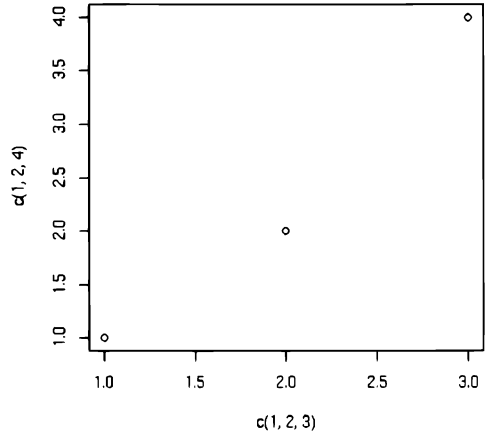


Рис. 12.1. Простейший точечный график

ПРИМЕЧАНИЕ

Точки на графике на рис. 12.1 обозначаются пустыми кружками. Если вы хотите использовать другой символ, укажите его в значении именованного аргумента pch.

Функция `plot()` работает поэтапно; это означает, что вы можете строить свой график шаг за шагом, выдавая серии команд. Например, сначала можно нарисовать пустой график, на котором нет ничего, кроме осей:

```
> plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")
```

Этот вызов рисует координатные оси с метками x и y. Горизонтальная ось (x) представляет значения в диапазоне от –3 до 3, а вертикальная (y) — в диапазоне от –1 до 5. Аргумент `type="n"` означает, что на самом графике пока ничего нет.

12.1.2. Рисование линий: функция abline()

Пустой график готов. На следующей стадии на него будет добавлена линия:

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

```
> plot(x,y)
> lmout <- lm(y ~ x)
> abline(lmout)
```

После первого вызова `plot()` на графике отображаются только три точки с осями x и y с засечками. Вызов `abline()` добавляет линию на текущий график. Но что это за линия?

Как вы узнали в разделе 1.5, результатом вызова функции линейной регрессии `lm()` является экземпляр класса, содержащий угол наклона и точку пересечения оси для подогнанной линии, а также некоторые другие величины, которые нас сейчас не интересуют. Этот экземпляр класса был присвоен `lmout`. Угол наклона и точка пересечения сейчас хранятся в `lmout$coefficients`.

Итак, что же происходит при вызове `abline()`? Функция просто рисует прямую линию, а ее аргументы интерпретируются как точка пересечения оси и угол наклона линии. Например, вызов `abline(c(2,1))` рисует на построенном вами графике следующую линию:

$$y = 2 + 1 \times x$$

Однако функция `abline()` написана так, чтобы при вызове для объекта регрессии она выполняла особые действия (хотя, как ни удивительно, она не является обобщенной функцией). Поэтому она берет необходимые аргументы наклона и пересечения из `lmout$coefficients` и рисует эту линию. Линия накладывается на текущий график (с тремя точками). Иначе говоря, на новом графике будут отображаться как точки, так и линия (рис. 12.2).

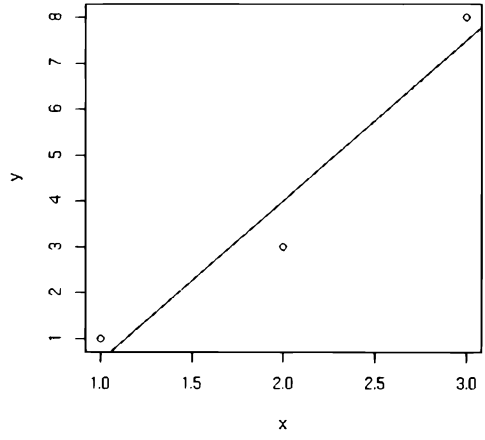


Рис. 12.2. Использование функции abline()

Вы можете добавить другие линии при помощи функции `lines()`. Функция поддерживает много аргументов, но два основных аргумента `lines()` — вектор значений `x` и вектор значений `y`. Они интерпретируются как пары `(x,y)`, которые представляют соединенные линиями точки, добавляемые на текущий график. Например, если `X` и `Y` равны `(1.5,2.5)` и `(3,3)`, вы можете использовать следующий вызов для добавления линии из (1.5, 3) в (2.5, 3) на текущий график:

```
> lines(c(1.5,2.5),c(3,3))
```

Если вы хотите, чтобы точки соединялись линиями (вместо отображения самих точек), включите аргумент `type="l"` в свой вызов `lines()` или `plot()`:

```
> plot(x,y,type="l")
```

Параметр `lty` определяет тип линии — например, сплошная или пунктирная. Чтобы просмотреть все доступные типы и их коды, введите следующую команду:

```
> help(par)
```

12.1.3. Создание нового графика при сохранении старых

Каждый раз, когда вы вызываете `plot()` (прямо или косвенно), текущее окно графика будет заменяться новым окном. Если вы не хотите, чтобы это происходило, используйте команду для своей операционной системы:

- в системах Linux вызовите `X11()`;
- на Mac вызовите `quartz()`;
- в Windows вызовите `windows()`.

Предположим, вы хотите построить две гистограммы для векторов `X` и `Y` и просмотреть их рядом друг с другом. В системах Linux это делается так:

```
> hist(x)
> x11()
> hist(y)
```

12.1.4. Расширенный пример: две оценки плотности на одном графике

Построим непараметрические оценки плотности (фактически сглаженные гистограммы) для двух наборов экзаменационных оценок на одном графике.

Оценки будут генерироваться функцией `density()`. Для этого следует ввести следующие команды:

```
> d1 = density(testscores$Exam1,from=0,to=100)
> d2 = density(testscores$Exam2,from=0,to=100)
> plot(d1,main="",xlab="")
> lines(d2)
```

Сначала мы вычисляем непараметрические оценки плотности по двум переменным, сохраняя их в объектах `d1` и `d2` для использования в будущем. Затем вызов `plot()` рисует кривую для экзамена 1; на этой стадии график выглядит так, как показано на рис. 12.3. После этого вызов `lines()` добавляет на график вторую кривую (рис. 12.4).

Обратите внимание на то, как мы приказываем R использовать пустые метки для графика в целом и для оси x. В противном случае R возьмет из `d1` метки, которые могут относиться к экзамену 1.

Также обратите внимание на то, что график экзамена 1 должен выводиться первым. На этом экзамене разброс оценок был меньше, поэтому график оценки плотности был более узким и высоким. Если бы мы начали с экзамена 2 с его более короткой кривой, то кривая экзамена 1 была бы слишком высокой для окна графика. В данном примере я сначала построил два графика по отдельности, чтобы увидеть, какой из них выше, но рассмотрим более общую ситуацию.

Допустим, вы хотите написать более универсальную функцию, которая выводит несколько оценок плотности на одном графике. Нам необходимо будет автоматизировать процесс определения самой высокой оценки плотности. Для этого

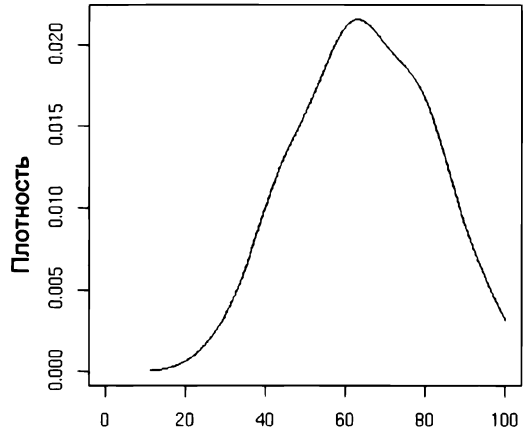


Рис. 12.3. Первый график плотности

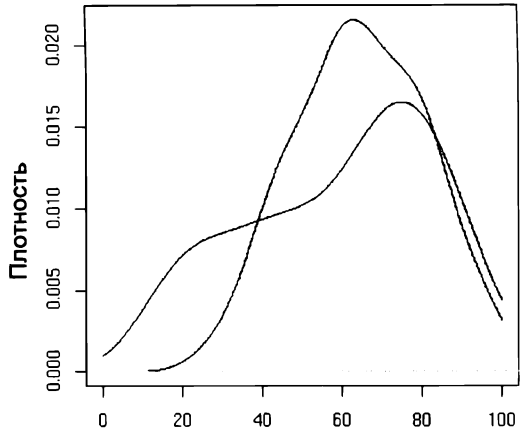


Рис. 12.4. Добавление второй плотности

мы воспользуемся тем фактом, что значения оценок плотности содержатся в компоненте y возвращаемого значения вызова `density()`. Затем мы вызовем `max()` для каждой оценки плотности и при помощи `which.max()` определим, какая оценка плотности имеет наибольшую высоту.

Вызов `plot()` создает график и рисует первую кривую. (Без аргумента `type="l"` на график были бы нанесены только точки.) Последующий вызов `lines()` добавляет вторую кривую.

12.1.5. Расширенный пример: подробнее о примере полиномиальной регрессии

В разделе 9.1.7 определяется класс `"polyreg"` для упрощения подгонки моделей полиномиальной регрессии. В наш код была включена реализация обобщенной функции `print()`. Теперь добавим такой код для обобщенной функции `plot()`:

```
# polyfit(x,maxdeg) подгоняет все полиномиальные модели до степени
# maxdeg; y – вектор переменной реакции, x – предиктор; создает
# объект класса "polyreg", содержащий вывод различных регрессионных
# моделей, а также исходные данные
polyfit <- function(y,x,maxdeg) {
   pwrs <- powers(x,maxdeg) # Вычислить степени предиктора
   lmout <- list() # Начать построение класса
   class(lmout) <- "polyreg" # Создать новый класс
   for (i in 1:maxdeg) {
      lmo <- lm(y ~ pwrs[,1:i])
      # Расширить класс lm с перекрестной проверкой прогнозов
```

```
12       lmo$fitted.xvvalues <- lvoneout(y,pwrs[,1:i,drop=F])
13       lmout[[i]] <- lmo
14    }
15    lmout$x <- x
16    lmout$y <- y
17    return(lmout)
18 }
19
20 # print() для объектов класса "polyreg": вывести среднеквадратические
21 # ошибки прогнозирования с перекрестной проверкой
22 print.polyreg <- function(fits) {
23    maxdeg <- length(fits) - 2 # Только выходные данные lm(), не $x и $y
24    n <- length(fits$y)
25    tbl <- matrix(nrow=maxdeg,ncol=1)
26    cat("mean squared prediction errors, by degree\n")
27    colnames(tbl) <- "MSPE"
28    for (i in 1:maxdeg) {
29       fi <- fits[[i]]
30       errs <- fits$y - fi$fitted.xvvalues
31       spe <- sum(errs^2)
32       tbl[i,1] <- spe/n
33    }
34    print(tbl)
35 }
36
37 # Обобщенная версия plot(); выводит подгонку по необработанным данным
38 plot.polyreg <- function(fits) {
39    plot(fits$x,fits$y,xlab="X",ylab="Y") # Точки данных выводятся как фон
40    maxdg <- length(fits) - 2
41    cols <- c("red","green","blue")
42    dg <- curvecount <- 1
43    while (dg < maxdg) {
44       prompt <- paste("RETURN for XV fit for degree",dg,"or type degree",
45          "or q for quit ")
46       rl <- readline(prompt)
47       dg <- if (rl == "") dg else if (rl != "q") as.integer(rl) else break
48       lines(fits$x,fits[[dg]]$fitted.values,col=cols[curvecount%%3 + 1])
49       dg <- dg + 1
50       curvecount <- curvecount + 1
51    }
52 }
53
54 # Формирует матрицу степеней вектора x до степени dg
55 powers <- function(x,dg) {
56    pw <- matrix(x,nrow=length(x))
57    prod <- x
58    for (i in 2:dg) {
59       prod <- prod * x
60       pw <- cbind(pw,prod)
```

```
61    }
62    return(pw)
63 }
64
65 # Находит прогнозируемые значения с перекрестной проверкой; может
66 # быть существенно ускорен методом обновления обратных матриц
67 lvoneout <- function(y,xmat) {
68    n <- length(y)
69    predy <- vector(length=n)
70    for (i in 1:n) {
71       # Регрессия с исключением i-го наблюдения
72       lmo <- lm(y[-i] ~ xmat[-i,])
73       betahat <- as.vector(lmo$coef)
74       # 1 для постоянной составляющей
75       predy[i] <- betahat %*% c(1,xmat[i,])
76    }
77    return(predy)
78 }
79
80 # Полиномиальная функция x, коэффициенты cfs
81 poly <- function(x,cfs) {
82    val <- cfs[1]
83    prod <- 1
84    dg <- length(cfs) - 1
85    for (i in 1:dg) {
86       prod <- prod * x
87       val <- val + cfs[i+1] * prod
88    }
89 }
```

Как упоминалось ранее, новым является только код `plot.polyreg()`. Для удобства повторю этот код отдельно:

```
# Обобщенная версия plot(); выводит подгонку по необработанным данным
plot.polyreg <- function(fits) {
   plot(fits$x,fits$y,xlab="X",ylab="Y") # Точки данных выводятся как фон
   maxdg <- length(fits) - 2
   cols <- c("red","green","blue")
   dg <- curvecount <- 1
   while (dg < maxdg) {
      prompt <- paste("RETURN for XV fit for degree",dg,"or type degree",
         "or q for quit ")
      rl <- readline(prompt)
      dg <- if (rl == "") dg else if (rl != "q") as.integer(rl) else break
      lines(fits$x,fits[[dg]]$fitted.values,col=cols[curvecount%%3 + 1])
      dg <- dg + 1
      curvecount <- curvecount + 1
   }
}
```

Как и прежде, наша реализация обобщенной функции получает имя класса — в данном случае `plot.polyreg()`.

Цикл `while` перебирает различные степени полинома. При этом последовательно перебираются три цвета из вектора `cols`; обратите внимание на использование выражения `curvecount %%3` для этой цели.

Пользователь может выбрать между выводом следующей по порядку степени или же выбрать другую степень. Запрос данных (как вывод приглашения, так и чтение ответа пользователя) осуществляется в следующей строке:

```
rl <- readline(prompt)
```

Строковая функциz R `paste()` строит приглашение, предлагая пользователю выбрать между выводом следующей полиномиальной подгонки, выводом модели с другой степенью или выходом. Приглашение выводится в интерактивном окне R, в котором был выполнен вызов `plot()`. Например, после двукратного подтверждения варианта по умолчанию окно команд будет выглядеть так:

```
> plot(lmo)
RETURN for XV fit for degree 1 or type degree or q for quit
RETURN for XV fit for degree 2 or type degree or q for quit
RETURN for XV fit for degree 3 or type degree or q for quit
```

Окно графика показано на рис. 12.5.

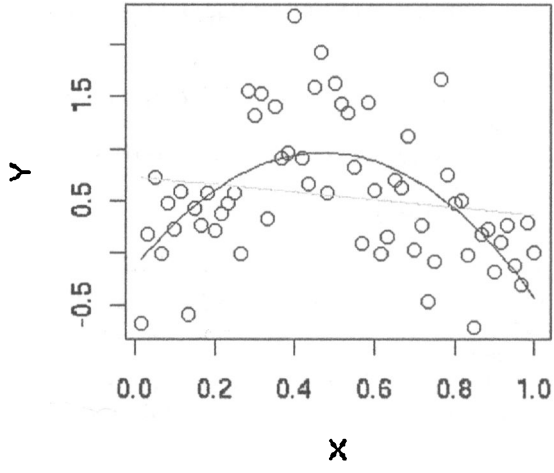


Рис. 12.5. График подгонки полиномиальных моделей

12.1.6. Добавление точек: функция points()

Функция `points()` добавляет на текущий график множество точек `(x,y)` с меткой для каждой точки. Допустим, в первом примере была введена следующая команда:

```
points(testscores$Exam1,testscores$Exam3,pch="+")
```

В результате точки результатов экзамена из нашего примера, помеченные знаками `+`, накладываются на текущий график. Как и большинство графических функций, `points()` обладает широкими возможностями настройки (например, цвет точек и цвет фона). Скажем, если вы хотите использовать желтый фон, введите следующую команду:

```
> par(bg="yellow")
```

Теперь графики будут выводиться на желтом фоне, пока вы не отмените этот режим.

Как и для других функций, для получения информации о многочисленных настройках `points()` введите следующую команду:

```
> help(par)
```

12.1.7. Добавление условных обозначений: функция legend()

Функция `legend()` добавляет условные обозначения на график с несколькими кривыми. Например, в нем может быть написано: «Зеленая кривая — график для мужчин, а красная — для женщин». Следующая команда отображает несколько эффектных примеров:

```
> example(legend)
```

12.1.8. Добавление текста: функция text()

Функция `text()` используется для размещения текста на текущем графике. Пример:

```
text(2.5,4,"abc")
```

Текст «abc» выводится у точки `(2.5,4)` на графике. В этой точке будет размещен центр строки, в данном случае символ «b».

Чтобы увидеть более практичный пример, добавим несколько меток к кривым на графике экзаменационных оценок:

```
> text(46.7,0.02,"Exam 1")
> text(12.3,0.008,"Exam 2")
```

Результат показан на рис. 12.6.

Чтобы конкретная строка размещалась именно в том месте, где нужно, возможно, вам придется немного поэкспериментировать. А может быть, функция `locator()`, рассматриваемая в следующем разделе, позволит намного быстрее прийти к желаемому результату.

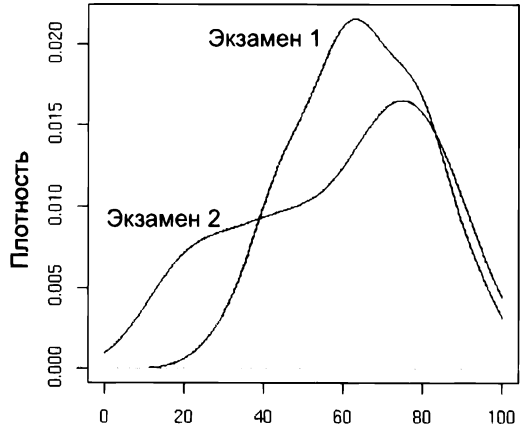


Рис. 12.6. Размещение текста

12.1.9. Функция locator()

Точно разместить текст в нужной позиции может быть непросто. Вам придется многократно опробовать разные координаты x и y, пока не будет найдена хорошая позиция, но функция `locator()` избавит вас от многих хлопот. Просто вызовите функцию и щелкните кнопкой мыши в нужной точке графика. Функция возвращает координаты x и y точки щелчка. Следующий вызов сообщит R, что вы щелкнете в одной точке на графике:

```
locator(1)
```

После щелчка R сообщит точные координаты выбранной точки. Вызов `locator(2)` вернет координаты двух точек и т. д. (Внимание: не забудьте включить аргумент при вызове!)

Простой пример:

```
> hist(c(12,5,13,25,16))
> locator(1)
$x
[1] 6.239237

$y
[1] 1.221038
```

R выведет гистограмму, а затем вызовет `locator()` с аргументом 1; это означает, что пользователь сделает один щелчок кнопкой мыши. После щелчка функция вернет список с компонентами x и y — координатами точки, в которой был сделан щелчок.

Чтобы использовать эту информацию для размещения текста, объедините ее с вызовом `text()`:

```
> text(locator(1),"nv=75")
```

Здесь функция `text()` ожидала получить координаты x и y точки для вывода текста `"nv=75"`. Возвращаемое значение `locator()` предоставляет эти координаты.

12.1.10. Восстановление графика

В R нет команды «отмена». Тем не менее если вы ожидаете, что следующий шаг при построении графика придется отменить, сохраните график вызовом `recordPlot()`, а затем восстановите его функцией `replayPlot()`.

Также существует другое, менее формальное, но более удобное решение: поместите все команды, используемые для построения графика, в файл, а затем используйте функцию `source()` или вставку/копирование для их выполнения. Если вы измените одну команду, весь график можно будет восстановить вызовом `source()` или копированием/вставкой файла.

Например, для нашего текущего графика можно создать файл с именем examplot.R и следующим содержимым:

```
d1 = density(testscores$Exam1,from=0,to=100)
d2 = density(testscores$Exam2,from=0,to=100)
plot(d1,main="",xlab="")
lines(d2)
text(46.7,0.02,"Exam 1")
text(12.3,0.008,"Exam 2")
```

Если вы решите, что метка `Exam 1` слишком смещена вправо, отредактируйте файл и выполните копирование/вставку или следующую команду:

```
> source("examplot.R")
```

12.2. Настройка графиков

Итак, вы видели, как легко строить простые графики поэтапно, начиная с вызова `plot()`. Теперь можно заняться оформлением этих графиков при помощи многочисленных настроек, предусмотренных в R.

12.2.1. Изменение размера символов: аргумент cex

Аргумент `cex` (сокращение от «character expand») позволяет увеличивать или уменьшать символы на графике, что может быть очень полезно. Он может использоваться как именованный аргумент в различных графических функциях. Допустим, вы хотите вывести текст `"abc"` в некоторой точке вашего графика (например, `(2.5,4)`), но более крупным шрифтом для привлечения внимания к этому конкретному тексту. Это можно сделать следующей командой:

```
text(2.5,4,"abc",cex = 1.5)
```

Команда выводит тот же текст, что и в предыдущем примере, но с символами в 1,5 раза больше обычного.

12.2.2. Изменение диапазонов осей: аргументы xlim и ylim

Иногда бывает нужно расширить или сузить диапазоны осей x и y вашего графика относительно значений по умолчанию. В частности, это особенно полезно при отображении нескольких кривых на одном графике.

Для настройки осей используются параметры `xlim` и/или `ylim` при вызове `plot()` или `points()`. Например, аргумент `ylim=c(0,90000)` устанавливает для оси y диапазон от 0 до 90 000.

Если при выводе нескольких кривых не указаны аргументы `xlim` и/или `ylim`, самую высокую кривую следует вывести первой, чтобы хватило места для всех остальных. В противном случае R подгонит размеры графика к первой выведенной кривой, а более высокие кривые будут обрезаны сверху! Этот подход был применен ранее, когда мы нанесли две оценки плотности на один и тот же график (см. рис. 12.3 и 12.4). Вместо этого можно было сначала определить наивысшие значения двух оценок плотности. Для `d1` результат выглядит так:

```
> d1

Call:
density.default(x = testscores$Exam1, from = 0, to = 100)

Data: testscores$Exam1 (39 obs.); Bandwidth 'bw' = 6.967

      x           y
Min.   : 0  Min.   :1.423e-07
1st Qu.: 25 1st Qu.:1.629e-03
Median : 50 Median :9.442e-03
Mean   : 50 Mean   :9.844e-03
3rd Qu.: 75 3rd Qu.:1.756e-02
Max.   :100 Max.   :2.156e-02
```

Итак, наибольшее значение y равно 0,022. Для d2 оно равно только 0,017. Это означает, что если присвоить ylim значение 0,03, останется достаточно места. Вот как можно вывести две кривые на одном графике:

```
> plot(c(0, 100), c(0, 0.03), type = "n", xlab="score", ylab="density")
> lines(d2)
> lines(d1)
```

Сначала рисуется минимальный график — только оси без данных (рис. 12.7). Первые два аргумента plot() задают xlim и ylim, так что верхний и нижний предел по оси y будет равен 0 и 0,03 соответственно. Затем двукратный вызов

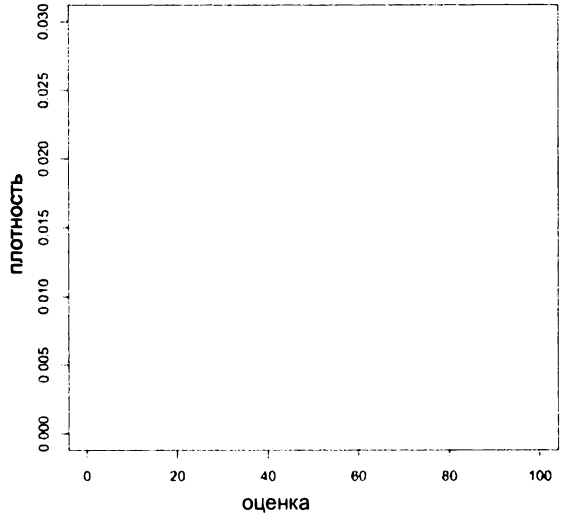


Рис. 12.7. Только оси

`lines()` заполняет график, и мы получаем рис. 12.8 и 12.9. (Вызовы `lines()` могут следовать в любом порядке, так как мы оставили достаточно места.)

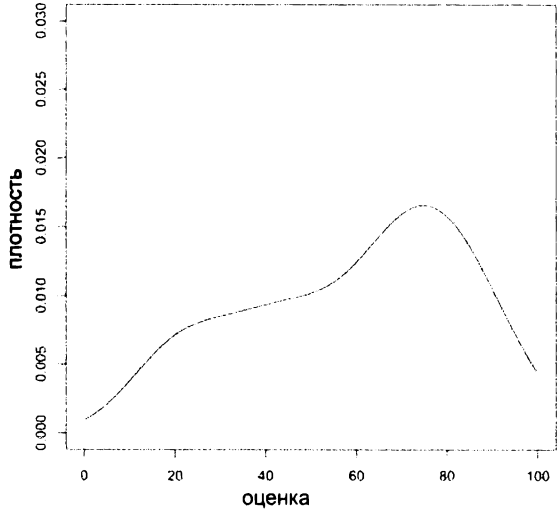


Рис. 12.8. Добавление d2

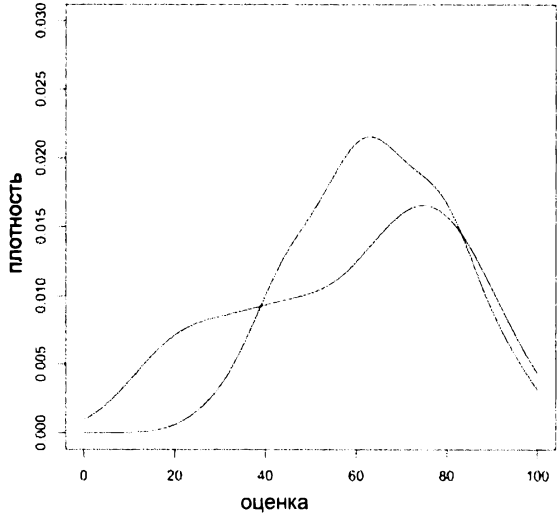


Рис. 12.9. Добавление d1

12.2.3. Добавление многоугольника: функция polygon()

Функция `polygon()` предназначена для рисования произвольных многоугольных объектов. Например, следующий код рисует график функции $f(x)=1-e^{-x}$, а затем добавляет прямоугольник, аппроксимирующий площадь под кривой от $x = 1{,}2$ до $x = 1{,}4$.

```
> f <- function(x) return(1-exp(-x))
> curve(f,0,2)
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),col="gray")
```

Результат показан на рис. 12.10.

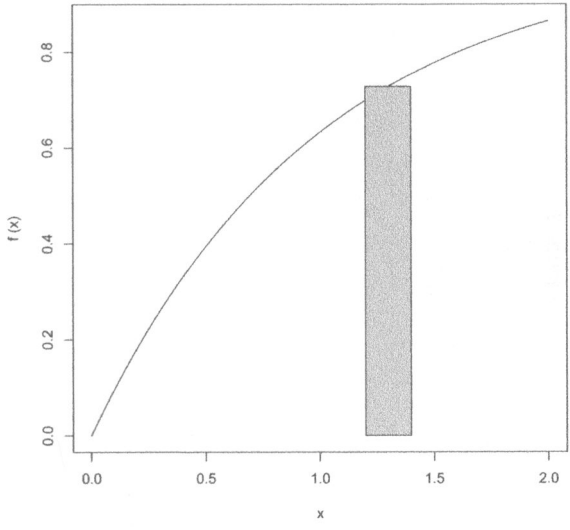


Рис. 12.10. Прямоугольное представление площади

В этом вызове `polygon()` первый аргумент содержит набор координат x для прямоугольника, а второй задает координаты y. Третий аргумент указывает, что прямоугольник должен быть закрашен сплошным серым цветом.

Другой пример: при помощи аргумента `density` можно раскрасить прямоугольник в полоску. Следующий вызов определяет 10 линий на дюйм:

```
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),density=10)
```

12.2.4. Сглаживание наборов точек: функции lowess() и loess()

Простой вывод облака точек, соединенных или нет, может превратиться в бессодержательную мешанину. Во многих случаях лучше сгладить данные, выполнив подгонку непараметрической регресионной оценки, такой как `lowess()`.

Проделаем это для данных тестов. Построим графики оценок экзамена 2 и экзамена 1:

```
> plot(testscores)
> lines(lowess(testscores))
```

Результат показан на рис. 12.11.

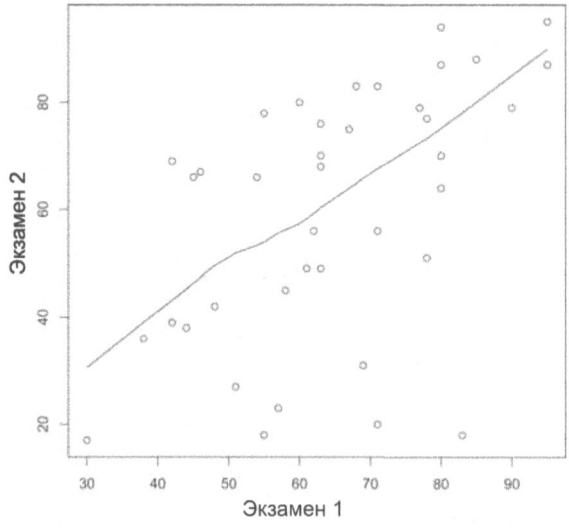


Рис. 12.11. Сглаживание данных оценок на экзаменах

У `lowess()` существует более новая альтернатива — функция `loess()`. Две функции похожи, но они имеют разные значения по умолчанию и другие настройки. Чтобы понимать, чем они отличаются, необходимо хорошо знать статистику. Используйте ту функцию, которая лучше выполнит сглаживание для ваших данных.

12.2.5. Построение графиков конкретных функций

Допустим, вы хотите построить график функции $g(t) = (t^2 + 1)^{0,5}$ для значений t от 0 до 5. Для этого можно использовать следующий код R:

```
g <- function(t) { return ((t^2+1)^0.5) } # define g()
x <- seq(0,5,length=10000) # x = [0.0004, 0.0008, 0.0012,..., 5]
y <- g(x) # y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]
plot(x,y,type="l")
```

Впрочем, можно немного упростить работу при помощи функции `curve()`, которая, по сути, использует тот же метод:

```
> curve((x^2+1)^0.5,0,5)
```

Если кривая добавляется на существующий график, укажите аргумент `add`:

```
> curve((x^2+1)^0.5,0,5,add=T)
```

Необязательный аргумент `n` имеет значение по умолчанию 101; это означает, что функция будет вычисляться в 101 точке, находящейся на равных расстояниях, в заданном диапазоне `x`.

Используйте столько точек, сколько потребуется для получения плавной кривой. Если окажется, что 101 точки недостаточно, поэкспериментируйте с более высокими значениями `n`.

Также можно воспользоваться функцией `plot()`:

```
> f <- function(x) return((x^2+1)^0.5)
> plot(f,0,5) # Аргумент должен быть именем функции
```

Здесь вызов `plot()` приводит к вызову `plot.function()`, реализации обобщенной функции `plot()` для класса `function`.

И снова выбор за вами: используйте тот метод, который вам больше нравится.

12.2.6. Расширенный пример: увеличение части кривой

После того как вы используете `curve()` для построения графика функции, возможно, вам захочется увеличить одну часть кривой. Для этого достаточно вызвать `curve()` повторно для той же функции, но с ограниченным диапазоном по оси x. Но предположим, вы хотите вывести исходный график и увеличенную область на одной диаграмме. В этом разделе будет разработана функция `inset()` для решения этой задачи.

Чтобы вам не приходилось заново выполнять работу, которую выполняла функция `curve()` при построении исходного графика, мы слегка изменим код

для сохранения этой работы с использованием возвращаемого значения. Для этого можно воспользоваться тем фактом, что вы можете легко просмотреть код функций R, написанных на R (в отличие от фундаментальных функций R, написанных на C):

```
> curve
function (expr, from = NULL, to = NULL, n = 101, add = FALSE,
    type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)
{
    sexpr <- substitute(expr)
    if (is.name(sexpr)) {
# ...Здесь пропущено много строк...
    x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {
        if (any(c(from, to) <= 0))
            stop("'from' and 'to' must be > 0 with log=\"x\"")
        exp(seq.int(log(from), log(to), length.out = n))
    }
    else seq.int(from, to, length.out = n)
    y <- eval(expr, envir = list(x = x), enclos = parent.frame())
    if (add)
        lines(x, y, type = type, ...)
    else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)
}
```

Код строит векторы x и y, состоящие из координат x и y выводимой кривой при n точках диапазона x, разделенных равными расстояниями. Поскольку мы будем использовать эти векторы в `inset()`, изменим этот код так, чтобы он возвращал x и y. Модифицированной версии будет присвоено имя `crv()`:

```
> crv
function (expr, from = NULL, to = NULL, n = 101, add = FALSE,
    type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)
{
    sexpr <- substitute(expr)
    if (is.name(sexpr)) {
#...Здесь пропущено много строк...
    x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {
        if (any(c(from, to) <= 0))
            stop("'from' and 'to' must be > 0 with log=\"x\"")
        exp(seq.int(log(from), log(to), length.out = n))
    }
    else seq.int(from, to, length.out = n)
    y <- eval(expr, envir = list(x = x), enclos = parent.frame())
    if (add)
        lines(x, y, type = type, ...)
```

```
    else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)
    return(list(x=x,y=y)) # Единственное изменение
}
```

Теперь можно переходить к функции `inset()`:

```
# savexy: список, состоящий из векторов x и y, возвращаемых crv()
# x1,y1,x2,y2: координаты прямоугольной области для увеличения
# x3,y3,x4,y4: координаты области вставки
inset <- function(savexy,x1,y1,x2,y2,x3,y3,x4,y4) {
   rect(x1,y1,x2,y2) # Нарисовать прямоугольник вокруг области увеличения
   rect(x3,y3,x4,y4) # Нарисовать прямоугольник вокруг вставки
   # Получить вектор координат ранее нарисованных точек
   savex <- savexy$x
   savey <- savexy$y
   # Получить индексы xi увеличиваемой области
   n <- length(savex)
   xvalsinrange <- which(savex >= x1 & savex <= x2)
   yvalsforthosex <- savey[xvalsinrange]
   # Проверить, что первое поле содержит всю кривую для этого диапазона X
   if (any(yvalsforthosex < y1 | yvalsforthosex > y2)) {
      print("Y value outside first box")
      return()
   }
   # Сохранить некоторые разности
   x2mnsx1 <- x2 - x1
   x4mnsx3 <- x4 - x3
   y2mnsy1 <- y2 - y1
   y4mnsy3 <- y4 - y3
   # Для i-й точки исходной кривой функция plotpt() вычислит
   # позицию этой точки в кривой в области вставки
   plotpt <- function(i) {
      newx <- x3 + ((savex[i] - x1)/x2mnsx1) * x4mnsx3
      newy <- y3 + ((savey[i] - y1)/y2mnsy1) * y4mnsy3
      return(c(newx,newy))
   }
   newxy <- sapply(xvalsinrange,plotpt)
   lines(newxy[1,],newxy[2,])
}
```

Опробуем новую функцию на практике:

```
xyout <- crv(exp(-x)*sin(1/(x-1.5)),0.1,4,n=5001)
inset(xyout,1.3,-0.3,1.47,0.3, 2.5,-0.3,4,-0.1)
```

Полученный график изображен на рис. 12.12.

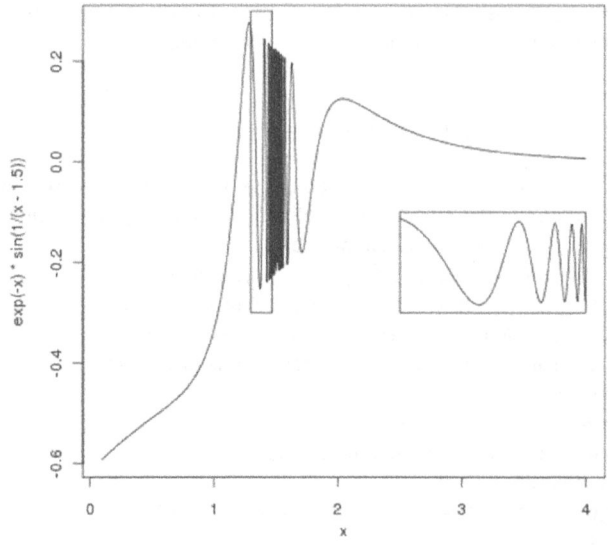


Рис. 12.12. Добавление вставки на график

12.3. Сохранение графиков в файлах

Графический вывод R может использовать различные графические устройства. Устройством по умолчанию является экран. Если вы захотите сохранить график в файле, для этого необходимо создать другое устройство.

Сначала мы рассмотрим основы работы с графическими устройствами R, а затем обсудим другой способ — более простой и удобный.

12.3.1. Графические устройства R

Откроем файл:

```
> pdf("d12.pdf")
```

Эта команда открывает файл `d12.pdf`. Теперь открыто два устройства, в чем нетрудно убедиться:

```
> dev.list()
X11 pdf
  2   3
```

При выполнении R в Linux устройство экрана называется `X11`. (В системах Windows ему присваивается имя `windows`.) В приведенном выводе это устройство

с номером 2. Наш PDF-файл является устройством с номером 3. Активным устройством является PDF-файл:

```
> dev.cur()
pdf
  3
```

Весь графический вывод будет направлен в файл, а не на экран. А если вы захотите сохранить то, что уже находится на экране?

12.3.2. Сохранение выведенного графика

Чтобы сохранить график, в настоящее время выведенный на экран, следует восстановить экран как текущее устройство, а затем скопировать его в устройство PDF-файла, которому в нашем примере присвоено значение 3:

```
> dev.set(2)
X11
  2
> dev.copy(which=3)
pdf
  3
```

Впрочем, было бы лучше создать устройство для PDF-файла (см. выше), а затем повторить весь анализ, который привел к текущему экрану. Дело в том, что операция копирования может привести к несоответствиям между экранными и файловыми устройствами.

12.3.3. Закрытие графического устройства R

Созданный PDF-файл не может использоваться до того момента, когда он будет закрыт:

```
> dev.set(3)
pdf
  3
> dev.off()
X11
  2
```

Также для закрытия устройства можно выйти из R, если вы не собираетесь продолжать работу. Однако в будущих версиях R это поведение может измениться, поэтому, вероятно, лучше будет явно закрыть устройство.

12.4. Создание трехмерных графиков

R предлагает некоторые функции для трехмерного представления данных — например, функции `persp()` и `wireframe()`, которые рисуют поверхности, и функция `cloud()`, которая рисует трехмерные диаграммы разброса. Рассмотрим простой пример использования `wireframe()`:

```
> library(lattice)
> a <- 1:10
> b <- 1:15
> eg <- expand.grid(x=a,y=b)
> eg$z <- eg$x^2 + eg$x * eg$y
> wireframe(z ~ x+y, eg)
```

Сначала загружается библиотека `lattice`. Затем вызов `expand.grid()` создает кадр данных, состоящий из двух столбцов `x` и `y`, со всеми возможными комбинациями значений двух вводов. Здесь `a` и `b` содержат 10 и 15 значений соответственно, так что полученный кадр данных будет содержать 150 строк. (Учтите, что кадр данных, подаваемый на вход `wireframe()`, не обязан создаваться вызовом `expand.grid()`.)

Затем добавляется третий столбец с именем `z`, который является функцией первых двух столбцов. Вызов `wireframe()` создает график. Аргументы, заданные в форме регрессионной модели, указывают, что значения `z` должны отображаться на графике относительно `x` и `y`. Конечно, `z`, `x` и `y` — обозначения имен столбцов в `eg`. Результат показан на рис. 12.13.

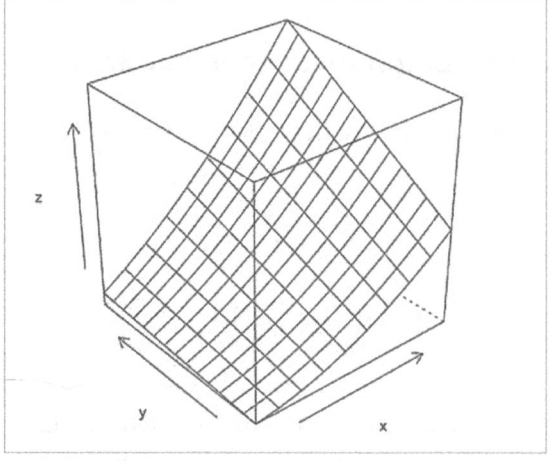


Рис. 12.13. Пример использования wireframe()

Все точки соединяются в поверхность (по аналогии с соединением точек линиями в двух измерениях). С другой стороны, при использовании `cloud()` точки отделены друг от друга. Для функции `wireframe()` пары (x, y) должны образовать прямоугольную сетку, хотя и не обязательно с равными интервалами.

Трехмерные функции графического вывода имеют много разных аргументов. Например, у `wireframe()` имеется удобный аргумент `shade=T`, который делает данные более заметными. Многие функции с нетривиальными аргументами, а также совершенно новые графические пакеты работают на более высоком (следует понимать «более мощном и удобном») уровне абстракции, чем базовый графический пакет R.

13 Отладка

Программисты часто проводят за отладкой программ больше времени, чем за их написанием. Хорошие навыки отладки чрезвычайно важны. В этой главе мы рассмотрим отладку в R.

13.1. Фундаментальные принципы отладки

> Остерегайтесь ошибок в приведенном выше коде; я только доказал его правильность, но не пробовал его запускать.
>
> *— Дональд Кнут, основоположник computer science*

Хотя отладка скорее относится к искусству, нежели к науке, у нее есть свои фундаментальные принципы. В этой главе будут рассмотрены некоторые приемы отладки.

13.1.1. Суть отладки: принцип подтверждения

Как мы с Питом Зальцманом (Pete Salzman) написали в книге, посвященной отладке («The Art of Debugging, with GDB, DDD, and Eclipse» (No Starch Press, 2008)), сутью отладки является принцип подтверждения.

«В сущности, исправление ошибок в программах — это процесс подтверждения того, что многие положения, в истинность которых вы верите в коде, действительно истинны. Когда вы обнаруживаете, что одно из ваших допущений не выполняется, вы тем самым находите ключ к местонахождению (если не точную природу) ошибки».

Это можно выразить иначе: «Сюрпризы — это хорошо!» Допустим, у вас имеется следующий код:

```
x <- y^2 + 3*g(z,2)
w <- 28
if (w+q > 0) u <- 1 else v <- 10
```

Вы думаете, что значение вашей переменной `x` равно 3 после присваивания? Подтвердите! Думаете, что будет выполнена секция `else`, а не `if` в третьей строке? Подтвердите!

Рано или поздно одно из утверждений, в истинности которых вы не сомневались, окажется неподтвержденным. Это будет означать, что вы обнаружили вероятное местонахождение ошибки, что позволит вам сосредоточиться на природе ошибки.

13.1.2. Запуск Small

Хотя бы в начале процесса отладки следует ограничиваться небольшими, простыми тестовыми сценариями. Работа с большими объектами данных усложнит анализ проблемы.

Конечно, со временем код придется тестировать для больших сложных случаев, но начинать следует с малого.

13.1.3. Модульная нисходящая отладка

Многие хорошие разработчики согласны с тем, что код следует писать по модульному принципу. Код первого уровня не должен быть длиннее, скажем, дюжины строк, большинство из которых должно состоять из вызовов функций. Эти функции должны быть не слишком длинными и вызывать другие функции в случае надобности. Это упрощает организацию кода на стадии написания и поможет в нем разобраться, когда придет время расширения кода.

Отладка должна осуществляться в нисходящем направлении. Допустим, вы установили статус отладки своей функции `f()` (то есть вызвали функцию `debug(f)`, о которой будет рассказано ниже), а `f()` содержит следующую строку:

```
y <- g(x,8)
```

К функции `g()` следует относиться по принципу «презумпции невиновности». Не торопитесь с вызовом `debug(g)` — сначала выполните строку и посмотрите,

вернет ли `g()` ожидаемое значение. Если значение будет получено, значит, вы только что избежали трудоемкого процесса пошаговой отладки `g()`. Если `g()` вернет другое значение, пришло время для вызова `debug(g)`.

13.1.4. Защитное программирование

Также вы можете применить некоторые стратегии «защитного программирования». Допустим, имеется область кода, в которой переменная `x` должна быть положительной. В нее можно вставить следующую строку:

```
stopifnot(x > 0)
```

Если ранее в этом коде происходит ошибка, из-за которой переменная `x` равна, допустим, −12, вызов `stopifnot()` прервет выполнение программы с выдачей сообщения об ошибке:

```
Error: x > 0 is not TRUE
```

(Программисты C могут заметить сходство с командой `assert` языка C.)

После исправления ошибки и тестирования нового кода можно оставить эту команду на месте — она поможет обнаружить эту ошибку, если та вдруг случайно вернется в ваш код.

13.2. Для чего использовать отладочные средства?

В прежние времена программисты для получения отладочных подтверждений вставляли в свой код временные команды вывода, снова запускали программу и смотрели, что она выведет. Например, чтобы убедиться, что в приведенном выше коде `x = 3` вы вставляли команду, которая выводила значение `x`, и делали нечто похожее для `if-else`:

```
x <- y^2 + 3*g(z,2)
cat("x =",x,"\n")
w <- 28
if (w+q > 0) {
   u <- 1
   print("the 'if' was done")
} else {
   v <- 10
   print("the 'else' was done")
}
```

Разработчик запускал программу заново и анализировал выведенный результат. После этого он удалял команды вывода и вставлял новые для обнаружения следующей ошибки.

Ручной процесс отладочного вывода хорошо работает один-два раза, но в ходе длинных сеансов отладки он становится утомительным. И что еще хуже, работа по отладке отвлекает разработчика и мешает ему сосредоточиться на поиске ошибки.

Итак, процесс отладки, основанный на вставке команд вывода в код, получается медленным и громоздким, и он отвлекает разработчика от основной работы. Если вы серьезно относитесь к программированию на любом конкретном языке, поищите хорошее отладочное средство для этого языка. Использование отладочных средств значительно упрощает получение значений переменных, проверку того, была ли выполнена секция `if` или `else`, и т. д. Более того, если ошибка приводит к сбою выполнения, отладочные средства смогут проанализировать ее за вас — и возможно, предоставить важную информацию об источнике ошибки. Все это существенно повысит производительность вашей работы.

13.3. Использование отладочных средств R

Базовый пакет R включает подборку отладочных средств; также доступны более функциональные отладочные пакеты. Далее будут рассмотрены как базовые средства, так и другие пакеты, а в расширенном примере будет представлен очень подробный сеанс отладки.

13.3.1. Пошаговое выполнение кода функциями debug() и browser()

Отладочные средства R работают на основе режима просмотра (browser). Разработчик может выполнять код в пошаговом режиме, строку за строкой, анализируя текущее состояние в процессе выполнения. Режим просмотра активизируется вызовом функции `debug()` или `browser()`.

13.3.2. Использование команд просмотра

В режиме просмотра приглашение > заменяется на `Browse[d]>` (где `d` — глубина цепочки вызовов). В этом приглашении вводятся любые из следующих команд:

- `n` (от «next») — приказывает R выполнить следующую строку, а затем сделать паузу. Нажатие клавиши Enter также выполняет это действие.
- `c` (от «continue») — аналог `n`, за исключением того, что до следующей паузы могут быть выполнены несколько строк кода. Если программа находится в цикле, эта команда выполнит оставшуюся часть цикла, а затем сделает паузу при выходе из цикла. Если программа выполняет функцию, но не находится в цикле, оставшаяся часть функции будет выполнена перед следующей паузой.
- любая команда R — в режиме просмотра вы все еще находитесь в интерактивном режиме R, а следовательно, можете запросить значение `x`, для чего достаточно ввести `x`. Конечно, если у вас имеется переменная, имя которой совпадает с командой режима просмотра, необходимо использовать явный вызов с `print()` — например, `print(n)`.
- `where` — выводит трассировку стека. Выводит последовательность вызовов функций, которая привела к текущей позиции.
- `Q` — завершает режим просмотра и возвращается к основному интерактивному режиму R.

13.3.3. Назначение точек прерывания

Вызов `debug(f)` помещает вызов `browser()` в начало `f()`. Тем не менее в некоторых случаях такой инструмент может быть слишком грубым. Если вы подозреваете, что ошибка скрыта в середине функции, вам придется долго пробираться через весь промежуточный код.

Проблема решается установкой точек прерывания в ключевых позициях вашего кода — в тех местах, в которых выполнение должно остановиться. Как сделать это в R? Вы можете вызвать режим просмотра напрямую или воспользоваться функцией `setBreakpoint()` (в R версии 2.10 и выше).

13.3.3.1. Прямой вызов browser()

Точку прерывания можно установить простым включением вызовов `browser()` в местах, представляющих для вас интерес в коде. Фактически это эквивалентно расстановке точек прерывания.

Вызов `browser()` можно сделать условным, чтобы он активизировался только в конкретных ситуациях. Используйте аргумент `expr` для определения таких ситуаций. Например, вы подозреваете, что ошибка возникает только тогда,

когда некая переменная s превышает 1. Для этого можно использовать следующий код:

```
browser(s > 1)
```

Режим просмотра будет включаться только в том случае, если s больше 1. Аналогичного эффекта можно добиться следующей командой:

```
if (s > 1) browser()
```

Прямой вызов browser() вместо входа в отладчик через debug() очень полезен в ситуациях с циклом с множеством итераций, если ошибка, скажем, всплывает только после 50-й итерации. Если переменная цикла равна i, можно включить следующую команду:

```
if (i > 49) browser()
```

И это позволит вам избежать хлопот с пошаговым выполнением первых 49 итераций!

13.3.3.2. Функция setBreakpoint()

Начиная с R 2.10, функция setBreakpoint() может использоваться в формате

```
setBreakpoint(имя_файла,номер_строки)
```

Это приведет к вызову browser() в строке *номер_строки* исходного файла *имя_файла*.

Данная возможность особенно полезна тогда, когда вы где-то в середине сеанса отладки выполняете код в пошаговом режиме. Допустим, в настоящее время вы находитесь в строке 12 исходного файла x.R и хотите установить точку прерывания в строке 28. Вместо того чтобы выходить из отладчика, добавлять вызов browser() в строке 28, а затем входить в функцию заново, достаточно ввести следующую команду:

```
> setBreakpoint("x.R",28)
```

После этого можно продолжить выполнение в отладчике — допустим, командой c.

Функция setBreakpoint() вызывает функцию trace(), рассмотренную в следующем разделе. Таким образом, для отмены точки прерывания следует отменить отслеживание. Например, если функция setBreakpoint() была вызвана в строке функции g(), ее можно отменить следующим вызовом:

```
> untrace(g)
```

Функцию `setBreakpoint()` можно вызывать независимо от того, находитесь вы в настоящий момент в отладчике или нет. Если вы не находитесь в отладчике и при выполнении соответствующей функции управление будет передано в точку прерывания, вы автоматически перейдете в режим просмотра. Происходит то же, что при вызове `browser()`, но в этом случае вам не придется изменять свой код в текстовом редакторе.

13.3.4. Функция trace()

Функция `trace()` отличается мощью и гибкостью, хотя на ее освоение у вас уйдет некоторое время. Мы рассмотрим некоторые простейшие формы, начиная со следующей:

```
> trace(f,t)
```

Этот вызов приказывает R вызывать функцию `t()` при каждом входе в функцию `f()`. Допустим, вы хотите установить точку прерывания в начале функции `gy()`. Для этого можно воспользоваться следующей командой:

```
> trace(gy,browser)
```

Произойдет то же, что происходит при включении команды `browser()` в исходный код `gy()`, но этот способ быстрее и удобнее: вам не нужно вставлять команду, сохранять файл и снова выполнять `source()` для загрузки новой версии файла. Вызов `trace()` не изменяет исходный файл, хотя и модифицирует временную версию файла, находящуюся под управлением R. Кроме того, он быстрее и проще отменяется — для этого достаточно выполнить `untrace()`:

```
> untrace(gy)
```

Отслеживание можно включать и отключать глобально вызовом `tracingState()`; с аргументом `TRUE` отслеживание включается, а с аргументом `FALSE` оно отключается.

13.3.5. Выполнение проверок после сбоя функциями traceback() и debugger()

Допустим, в вашем коде R происходит сбой, когда он не выполняется в отладчике. Даже после этого в вашем распоряжении имеется полезный инструмент отладки. Чтобы провести «вскрытие», вызовите `traceback()`. Вы узнаете, в какой функции возникла проблема и какая цепочка вызовов привела к этой функции.

Чтобы получать намного больше информации, настройте R для вывода дампа кадров в случае сбоя:

```
> options(error=dump.frames)
```

Если это было сделано, после сбоя выполните следующую команду:

```
> debugger()
```

Вам будет предложено выбрать уровни вызовов функций для просмотра. Для каждого выбранного уровня вы сможете просмотреть значения переменных на этом уровне. Чтобы после просмотра одного уровня вернуться в главное меню `debugger()`, нажмите клавишу N.

Чтобы автоматически входить в отладчик, напишите следующий код:

```
> options(error=recover)
```

Однако учтите, что если вы выберете автоматический вариант, он будет переводить вас в отладчик даже при простой синтаксической ошибке (не самый полезный момент для входа в отладчик).

Чтобы отключить все эти аспекты поведения, введите следующую команду:

```
> options(error=NULL)
```

Примеры использования этих возможностей продемонстрированы в следующем разделе.

13.3.6. Расширенный пример: два полных сеанса отладки

Итак, мы рассмотрели отладочные средства R. Теперь попробуем использовать их для поиска и исправления ошибок в коде. Начнем с простого примера, а потом перейдем к более сложному.

13.3.6.1. Поиск серий единиц

Для начала вспомните расширенный пример с поиском серий единиц из главы 2. Ниже приведена версия этого кода с ошибкой:

```
1 findruns <- function(x,k) {
2    n <- length(x)
3    runs <- NULL
4    for (i in 1:(n-k)) {
```

```
5       if (all(x[i:i+k-1]==1)) runs <- c(runs,i)
6     }
7     return(runs)
8 }
```

Проверим на небольшом тестовом сценарии:

```
> source("findruns.R")
> findruns(c(1,0,0,1,1,0,1,1,1),2)
[1] 3 4 6 7
```

Функция должна сообщить о сериях единиц с индексами 4, 7 и 8, но при запуске были обнаружены индексы, которых быть не должно, а некоторые серии были пропущены. Что-то пошло не так. Давайте войдем в отладчик и осмотримся.

```
> debug(findruns)
> findruns(c(1,0,0,1,1,0,1,1,1),2)
debugging in: findruns(c(1, 0, 0, 1, 1, 0, 1, 1, 1), 2)
debug at findruns.R#1: {
   n <- length(x)
   runs <- NULL
   for (i in 1:(n - k)) {
      if (all(x[i:i+k-1]==1))
         runs <- c(runs, i)
   }
   return(runs)
}
attr(,"srcfile")
findruns.R
```

В соответствии с принципом подтверждения мы сначала убедимся в том, что тестовый вектор был получен правильно:

```
Browse[2]> x
[1] 1 0 0 1 1 0 1 1 1
```

Пока все хорошо. Пройдем по этому коду более внимательно. Мы пару раз нажали `n`, чтобы код был выполнен в пошаговом режиме.

```
Browse[2]> n
debug at findruns.R#2: n <- length(x)
Browse[2]> n
debug at findruns.R#3: runs <- NULL
Browse[2]> print(n)
[1] 9
```

После каждого шага R сообщает, какая команда будет выполнена следующей. Другими словами, во время выполнения `print(n)` присваивание `NULL` переменной `runs` еще не было выполнено.

Также обратите внимание на то, что хотя обычно значение переменной можно вывести простым вводом имени, с переменной `n` такой способ не сработает, потому что `n` также является сокращением команды `next` отладчика. А значит, необходимо вызвать `print()`.

В любом случае выяснилось, что длина тестового вектора равна 9 — это подтверждает то, что нам уже известно. Продолжим выполнение в пошаговом режиме и войдем в цикл:

```
Browse[2]> n
debug at findruns.R#4: for (i in 1:(n-k+1)){
    if (all(x[i:i+k-1]==1))
        runs <- c(runs, i)
}
Browse[2]> n
debug at findruns.R#4: i
Browse[2]> n
debug at findruns.R#5: if (all(x[i:i+k-1]==1)) runs <- c(runs, i)
```

Так как переменная `k` равна 2 (то есть мы проверяем серии длины 2), команда `if()` должна проверить первые два элемента `x`, то есть (1,0). Убедимся в этом:

```
Browse[2]> x[i:i+k-1]
[1] 0
```

Подтверждение *не* прошло. Проверим правильность диапазона индексов — это должен быть диапазон `1:2`. Так?

```
Browse[2]> i:i+k-1
[1] 2
```

Снова неправильно. Тогда как насчет переменных `i` и `k`? Они должны быть равны 1 и 2 соответственно. Так ли это?

```
Browse[2]> i
[1] 1
Browse[2]> k
[1] 2
```

Что ж, это предположение подтвердилось. Следовательно, проблема должна быть с выражением `i:i+k-1`. После некоторых размышлений становится ясно, что здесь возникает проблема с приоритетом операторов. Исправим ее и приведем к правильной форме `i:(i+k-1)`.

Теперь нормально?

```
> source("findruns.R")
> findruns(c(1,0,0,1,1,0,1,1,1),2)
[1] 4 7
```

Нет. Как упоминалось ранее, должно быть (4, 7, 8).

Установим точку прерывания в цикле и присмотримся повнимательнее:

```
> setBreakpoint("findruns.R",5)
/home/nm/findruns.R#5:
 findruns step 4,4,2 in <environment: R_GlobalEnv>
> findruns(c(1,0,0,1,1,0,1,1,1),2)
findruns.R#5
Called from: eval(expr, envir, enclos)
Browse[1]> x[i:(i+k-1)]
[1] 1 0
```

Хорошо, мы имеем дело с первыми двумя элементами вектора, так что первое исправление ошибки работает. Перейдем ко второй итерации цикла:

```
Browse[1]> c
findruns.R#5
Called from: eval(expr, envir, enclos)
Browse[1]> i
[1] 2
Browse[1]> x[i:(i+k-1)]
[1] 0 0
```

Тоже правильно. Можно перейти к другой итерации, но вместо этого рассмотрим последнюю итерацию — одно из тех мест, в котором ошибки часто встречаются в циклах. Создадим условную точку прерывания:

```
findruns <- function(x,k) {
   n <- length(x)
   runs <- NULL
   for (i in 1:(n-k)) {
      if (all(x[i:(i+k-1)]==1)) runs <- c(runs,i)
      if (i == n-k) browser() # Прервать в последней итерации цикла
   }
   return(runs)
}
```

А теперь повторим попытку:

```
> source("findruns.R")
> findruns(c(1,0,0,1,1,0,1,1,1),2)
Called from: findruns(c(1, 0, 0, 1, 1, 0, 1, 1, 1), 2)
Browse[1]> i
[1] 7
```

Последняя итерация выполняется для i=7. Но вектор состоит из девяти элементов, и k=2, поэтому при последней итерации должно быть i=8. Поразмыслив, мы понимаем, что диапазон в цикле должен быть записан в следующей форме:

```
for (i in 1:(n-k+1)) {
```

Кстати говоря, обратите внимание на то, что точка прерывания, установленная вызовом `setBreakpoint()`, после замены старой версии объекта `findruns` уже недействительна.

Дальнейшее тестирование (здесь не показанное) доказывает, что код работает правильно. Перейдем к более сложному примеру.

13.3.6.2. Поиск пары городов

Вспомните пример из раздела 3.4.2 с поиском пары городов, удаленных на минимальное расстояние. Версия этого кода с ошибкой:

```
 1 # Возвращает минимальное значение d[i,j], i != j, и строку/столбец,
 2 # для которых достигается минимум; случай равенства не обрабатывается.
 3 #
 4 # По мотивам примера с матрицами расстояний
 5 mind <- function(d) {
 6    n <- nrow(d)
 7     # добавить столбец для определения номера строки для apply()
 8    dd <- cbind(d,1:n)
 9    wmins <- apply(dd[-n,],1,imin)
10     # Размер wmins равен 2xn: 1 строка – индексы, 2 – значения
11    i <- which.min(wmins[1,])
12    j <- wmins[2,i]
13    return(c(d[i,j],i,j))
14 }
15
16 # Найти позицию и значение минимума в строке x
17 imin <- function(x) {
18    n <- length(x)
19    i <- x[n]
20    j <- which.min(x[(i+1):(n-1)])
21    return(c(j,x[j]))
22 }
```

Воспользуемся средствами отладки R для поиска и исправления проблем. Сначала опробуем их на маленьком тестовом сценарии:

```
> source("cities.R")
> m <- rbind(c(0,12,5),c(12,0,8),c(5,8,0))
> m
```

```
     [,1] [,2] [,3]
[1,]    0   12    5
[2,]   12    0    8
[3,]    5    8    0
> mind(m)
Error in mind(m) : subscript out of bounds
```

Не лучшее начало! К сожалению, сообщение об ошибке не говорит о том, где именно сломался код. Однако отладчик предоставит всю необходимую информацию:

```
> options(error=recover)
> mind(m)
Error in mind(m) : subscript out of bounds

Enter a frame number, or 0 to exit

1: mind(m)

Selection: 1
Called from: eval(expr, envir, enclos)

Browse[1]> where

where 1: eval(expr, envir, enclos)
where 2: eval(quote(browser()), envir = sys.frame(which))
where 3 at cities.R#13: function ()
{
    if (.isMethodsDispatchOn()) {
        tState <- tracingState(FALSE)
...
```

Значит, проблема возникла в `mind()`, а не в `imin()`, и конкретно в строке 13. Это не значит, что корень проблемы не кроется в `imin()`, но пока начнем с первой функции.

ПРИМЕЧАНИЕ

Определить, что сбой произошел в строке 13, можно было и другим способом. Войдите в отладчик так же, как прежде, но проверьте локальные переменные. Если бы ошибка индексирования произошла в строке 9, то значение переменной wmins не было бы задано и при обращении к ней было бы получено сообщение об ошибке вида «Error: object 'wmins' not found». С другой стороны, если сбой произошел в строке 13, то была бы задана даже переменная j.

Так как ошибка произошла с d[i,j], рассмотрим эти переменные:

```
Browse[1]> d
     [,1] [,2] [,3]
[1,]    0   12    5
[2,]   12    0    8
[3,]    5    8    0
Browse[1]> i
[1] 2
Browse[1]> j
[1] 12
```

Очевидная проблема: у d только три столбца, но индекс столбца j равен 12.

Рассмотрим переменную, из которой была получена переменная j, то есть wmins:

```
Browse[1]> wmins
     [,1] [,2]
[1,]    2    1
[2,]   12   12
```

Код спроектирован так, что столбец k переменной wmins должен содержать информацию о минимальном значении в строке k матрицы d. Таким образом, здесь wmins говорит, что в первой строке (k =1) матрицы d, то есть (0,12,5), минимальное значение равно 12, и оно встречается в позиции с индексом 2. Но на самом деле это должно быть значение 5 в позиции с индексом 3! Значит, что-то пошло не по плану в этой строке:

```
wmins <- apply(dd[-n, ], 1, imin)
```

Есть несколько возможностей. Но поскольку в конечном итоге вызывается функция imin(), все эти возможности можно проверить из этой функции. Итак, установим статус отладки для imin(), прервем выполнение отладчика и выполним код повторно:

```
Browse[1]> Q
> debug(imin)
> mind(m)
debugging in: FUN(newX[, i], ...)
debug at cities.R#17: {
    n <- length(x)
    i <- x[n]
    j <- which.min(x[(i + 1):(n - 1)])
    return(c(j, x[j]))
}
...
```

Мы находимся в `imin()`. Проверим, правильно ли была получена первая строка `dd`, которая должна состоять из элементов (0,12,5,1):

```
Browse[4]> x
[1] 0 12 5 1
```

Предположение подтверждено. Похоже, можно сделать вывод, что первые два аргумента `apply()` были правильными, а проблема связана с `imin()`, хотя это еще нужно проверить.

Проведем пошаговое выполнение, время от времени вводя запросы для подтверждения:

```
Browse[2]> n
debug at cities.r#17: n <- length(x)
Browse[2]> n
debug at cities.r#18: i <- x[n]
Browse[2]> n
debug at cities.r#19: j <- which.min(x[(i + 1):(n - 1)])
Browse[2]> n
debug at cities.r#20: return(c(j, x[j]))
Browse[2]> print(n)
[1] 4
Browse[2]> i
[1] 1
Browse[2]> j
[1] 2
```

Напомню, что вызов `which.min(x[(i + 1):(n - 1)]` был спроектирован так, что поиск ведется только над диагональю. Это связано с тем, что матрица симметрична и рассматривать расстояние между городом и этим же городом не нужно.

Значение `j`=2 проверку не проходит. Минимальное значение в (0, 12, 5) равно 5, и оно встречается в позиции с индексом 3, а не с индексом 2. Следовательно, проблема кроется в следующей строке:

```
j <- which.min(x[(i + 1):(n - 1)])
```

Что здесь не так?

Немного поразмыслив, мы понимаем, что хотя минимальное значение в (0, 12, 5) встречается в позиции с индексом 3 этого вектора, это не то, что мы потребовали найти у функции `which.min()`. Выражение `i+1` означает, что запрашивается индекс минимума в (12, 5), который равен 2.

Мы запросили у `which,min()` правильную информацию, но неправильно использовали функцию, потому что нам нужен минимум в (0, 12, 5). Необходимо внести соответствующую корректировку в вывод `which.min()`:

```
j <- which.min(x[(i+1):(n-1)])
k <- i + j
return(c(k,x[k]))
```

Внесем исправление и попробуем снова:

```
> mind(m)
Error in mind(m) : subscript out of bounds

Enter a frame number, or 0 to exit

1: mind(m)

Selection:
```

О нет, *еще одна* ошибка выхода за границы! Чтобы увидеть, где сбой произошел на этот раз, мы введем ту же команду `where`. Обнаруживается, что он снова произошел в строке 13. Что теперь происходит с `i` и `j`?

```
Browse[1]> i
[1] 1
Browse[1]> j
[1] 5
```

Значение `j` все еще ошибочно: оно не может быть больше 3, так как матрица состоит всего из трех столбцов. С другой стороны, переменная `i` правильна. Глобальный минимум в `dd` равен 5, и это значение находится в строке 1, столбец 3.

Еще раз проверим источник `j` — матрицу `wmins`:

```
Browse[1]> wmins
     [,1] [,2]
[1,]    3    3
[2,]    5    8
```

Столбец 1 содержит значения 3 и 5, как и должно быть.

Вспомните, что столбец 1 содержит информацию для строки 1 в `d`; таким образом, `wmins` говорит, что минимальное значение в строке 1 равно 5 и находится в позиции с индексом 3 этой строки, и это правильно.

После новых размышлений становится ясно, что хотя матрица `wmins` верна, мы неправильно используем ее: перепутаны строки и столбцы этой матрицы.

Код:

```
i <- which.min(wmins[1,])
j <- wmins[2,i]
```

должен выглядеть так:

```
i <- which.min(wmins[2,])
j <- wmins[1,i]
```

После внесения изменения и повторной загрузки файла опробуем его на практике.

```
> mind(m)
[1] 5 1 3
```

Теперь все правильно, и последующие тесты с матрицами большего размера тоже успешно работают.

13.4. На пути прогресса: более удобные средства отладки

Как вы уже видели, средства отладки R достаточно эффективны. Тем не менее они не очень удобны. К счастью, существуют различные инструменты, которые упрощают процесс отладки. Перечислю их приблизительно в хронологическом порядке разработки:

- пакет `debug` Марка Бревингтона (Mark Bravington);
- мой пакет `edtdbg`, работающий с текстовыми редакторами Vim и Emacs;
- пакет `ess-tracebug` Витали Спину (Vitalie Spinu), работающий в Emacs (решает те же задачи, что и `edtdbg`, но в большей степени ориентирован на специфическую функциональность Emacs);
- интегрированная среда разработки (IDE) от REvolution Analytics.

ПРИМЕЧАНИЕ

На момент написания книги (июль 2011 года) команды, разрабатывающие IDE StatET и RStudio, работали над добавлением отладочных средств в свои продукты.

Все эти средства являются кроссплатформенными: они работают в Linux, Windows и на компьютерах Mac. Исключением является продукт REvolution Analytics: эта IDE доступна только в системах Windows с Microsoft Visual Studio. Все эти средства распространяются с открытым кодом или бесплатно (снова за исключением продукта REvolution Analytics).

Что же предлагают эти пакеты? Для меня одним из самых больших недостатков встроенных отладочных средств R является отсутствие окна с «общей картиной» — окна с кодом R и курсором, который перемещается по коду в ходе пошагового выполнения. Для примера возьмем следующий фрагмент из приведенного ранее вывода:

```
Browse[2]> n
debug at cities.r#17: n <- length(x)
Browse[2]> n
debug at cities.r#18: i <- x[n]
```

Все хорошо, но где эти строки размещаются в нашем коде? Многие отладчики с графическим интерфейсом для других языков создают окно, в котором выводится исходный код пользователя, а следующая выполняемая строка обозначается специальным маркером. Все средства R, перечисленные в начале этого раздела, исправляют этот недостаток базовой функциональности R. Пакет `debug` Бревингтона создает для этой цели отдельное окно. Другие средства заставляют ваш текстовый редактор воспроизводить функциональность этого окна, что обеспечивает экономию места на экране по сравнению с использованием `debug`.

Кроме того, эти средства позволяют устанавливать точки прерывания и выполнять другие отладочные операции без перемещения указателя мыши из окна редактора в окно выполнения R. Это удобно, это сокращает количество нажатий клавиш и расширяет вашу возможность сосредоточиться на основной задаче — поиске ошибок.

Вернемся к примеру с городами. Я открыт текстовый редактор GVim для своего исходного файла в сочетании с `edtdbg`, выполнил подготовительные действия для `edtdbg`, после чего дважды нажал клавишу [(левая квадратная скобка) для пошагового выполнения кода. Полученное окно GVim показано на рис. 13.1.

ПРИМЕЧАНИЕ

С Emacs пакет `edtdbg` работает так же, но с другими клавишами для выполнения команд. Например, для пошагового выполнения используется клавиша F8 вместо [.

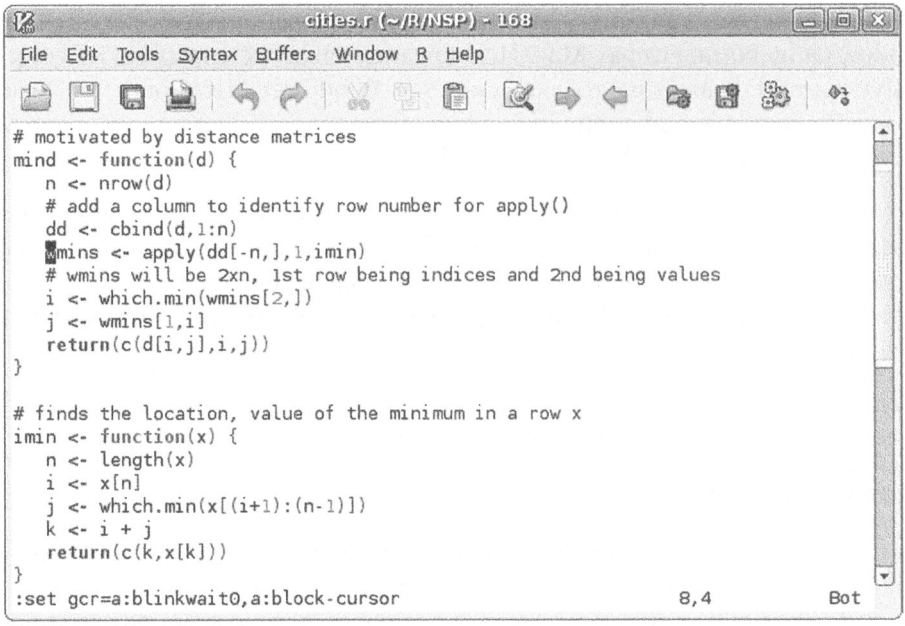


Рис. 13.1. Окно исходного кода в edtdbg

Но сначала обратите внимание на то, что курсор редактора установлен в следующей строке:

```
wmins <- apply(dd[-n, ], 1, imin)
```

Он обозначает следующую выполняемую строку.

Каждый раз, когда я хочу выполнить строку, я нажимаю клавишу [в окне редактора. Редактор приказывает режиму просмотра выполнить команду `n`, причем мне не приходится перемещать указатель мыши в окно выполнения R. Редактор переводит курсор к следующей строке. Также можно нажать клавишу] для выполнения команды `c` режима просмотра. Каждый раз, когда я выполняю одну или несколько строк подобным образом, происходит соответствующее смещение курсора в редакторе.

Каждый раз, когда я вношу изменения в исходный код, команда `,src` (запятая является частью команды) в окне GVim приказывает R вызвать для него `source()`. Каждый раз, когда потребуется заново выполнить код, я ввожу команду `,dt`. При этом мне практически никогда не приходится переводить указатель мыши из окна редактора в окно R и наоборот. В сущности, редактор становится отладчиком в дополнение к его основной функциональности.

13.5. Обеспечение согласованности в отладочном коде

Если в вашем коде используются случайные числа, необходимо иметь возможность воспроизводить один и тот же поток чисел при каждом запуске программы в ходе сеанса отладки. Без этого могут возникнуть трудности с воспроизведением ошибок, что усложняет их исправление.

Функция `set.seed()` предоставляет такую возможность, позволяя инициализировать последовательность случайных чисел конкретным значением.

Рассмотрим следующий пример:

```
[1] 0.8811480 0.2853269 0.5864738
> runif(3)
[1] 0.5775979 0.4588383 0.8354707
> runif(3)
[1] 0.4155105 0.4852900 0.6591892
> runif(3)
> set.seed(8888)
> runif(3)
[1] 0.5775979 0.4588383 0.8354707
> set.seed(8888)
> runif(3)
[1] 0.5775979 0.4588383 0.8354707
```

Вызов `runif(3)` генерирует три случайных числа из равномерного распределения в интервале (0,1). Каждый раз, когда в программе выполняется этот вызов, вы будете получать новый набор из трех чисел. Но функция `set.seed()` позволяет вернуться к началу и воспроизвести ту же серию чисел.

13.6. Синтаксические ошибки и ошибки времени выполнения

Самые распространенные синтаксические ошибки — пропущенные круглые, квадратные и фигурные скобки или кавычки. Когда вы сталкиваетесь с синтаксической ошибкой, это первое, что следует проверить и перепроверить. Я настоятельно рекомендую использовать текстовый редактор, который находит парные скобки и выделяет элементы синтаксиса R, такой как Vim или Emacs.

Учтите, что при получении сообщения, уведомляющего о наличии синтаксической ошибки в определенной строке, ошибка может скрываться в другой, более ранней

строке. Подобные проблемы возможны в любом языке, но похоже, R особенно подвержен им. Если точное местонахождение ошибки не очевидно, я рекомендую закомментировать часть кода, чтобы точнее локализовать синтаксическую ошибку. В общем случае полезно использовать метод бинарного поиска: закомментируйте половину кода (будьте внимательны, чтобы не нарушить целостность синтаксиса) и посмотрите, сохраняется ли ошибка. Если ошибка остается, значит, она находится в оставшейся половине; в противном случае она находится в удаленной половине. Затем поделите надвое выбранную половину и т. д.

Возможно, вы получите сообщения следующего вида:

```
There were 50 or more warnings (use warnings() to see the first 50)
```

На них стоит обратить внимание — введите команду `warnings()`, как и предложено. Причины возникших проблем могут быть самыми разными, от несходимости алгоритма до неверного определения матричного аргумента функции. Во многих случаях вывод программы оказывается недействительным, хотя и может быть вполне нормальным, как в следующем примере:

```
Fitted probabilities numerically 0 or 1 occurred in: glm...
```

В некоторых случаях может быть полезно ввести следующую команду:

```
> options(warn=2)
```

Эта команда приказывает R преобразовать предупреждения в ошибки и упрощает нахождение позиций, в которых они были обнаружены.

13.7. Применение GDB с кодом R

Этот раздел может представлять интерес даже в том случае, если вы не пытаетесь исправить ошибку в коде R. Например, вы написали код C, который должен взаимодействовать с R (см. главу 15), и ваш код не работает. Чтобы выполнить GDB с этой функцией C, необходимо сначала выполнить сам код R под управлением GDB.

Или же вы интересуетесь внутренним строением R, скажем, чтобы научиться писать эффективный код R, и теперь вы хотите выполнять исходный код R в пошаговом режиме в такой отладочной программе, как GDB.

И хотя R можно вызвать через GDB из командной строки (см. раздел 15.1.4), для наших целей я рекомендую использовать разные окна для R и GDB. Это делается так:

1. Запустите R в одном окне, как обычно.
2. В другом окне определите идентификатор своего процесса R. Например, в системах семейства UNIX это делается командой вида `ps -a`.
3. Во втором окне введите команду GDB `attach` с номером процесса R.
4. Введите команду GDB `continue`.

Теперь вы можете расставить точки прерывания в исходном коде R, прежде чем продолжить выполнение или прервать GDB позднее клавишами Ctrl+C. За подробностями об отладке кода C, вызываемого из R, обращайтесь к разделу 15.1.4. С другой стороны, если вы хотите использовать GDB для анализа исходного кода R, учтите следующее.

В исходном коде R ключевую роль играют указатели на выражения S (SEXP) — указатели на структуры языка C, содержащие значение переменной R, ее тип и т. д. Для анализа значений SEXP можно использовать внутреннюю функцию R `Rf_PrintValue(s)`. Например, если SEXP соответствует имя `s`, то в GDB следует ввести следующую команду:

```
call Rf_PrintValue(s)
```

Она выведет искомое значение.

14 Улучшение быстродействия: скорость и память

В учебных курсах информатики часто говорят о компромиссе между временем выполнения и затратами памяти. Чтобы программа быстрее работала, возможно, вам придется использовать больше памяти. С другой стороны, для экономии памяти иногда приходится мириться с замедлением работы кода. В языке R этот компромисс особенно важен по следующим причинам:

- R является интерпретируемым языком. Многие команды написаны на C, а следовательно, выполняются в виде быстрого машинного кода. Однако другие команды (а также ваш собственный код R) написаны на «чистом» R, а следовательно, они интерпретируются. Таким образом, возникает риск того, что ваше приложение R может работать медленнее, чем вам хотелось бы.
- Все объекты в сеансе R хранятся в памяти. А если говорить точнее, все объекты хранятся в адресном пространстве памяти R. R устанавливает лимит в $2^{31}-1$ байт для размера любого объекта, даже на 64-разрядных машинах и даже при большом объеме памяти. Тем не менее в некоторых приложениях встречаются объекты большего размера.

В этой главе предложены некоторые способы улучшения быстродействия кода R, которые учитывают упомянутый компромисс между временем и затратами памяти.

14.1. Написание быстрого кода R

Как ускорить выполнение кода R? Вам доступны следующие основные средства:

- Оптимизация кода R посредством векторизации, применение компиляции в байт-код и т. д.

- Написание ключевых частей кода, создающих интенсивную нагрузку на процессор, на компилируемом языке (таком, как C/C++).
- Написание своего кода на одной из разновидностей параллельного R.

Первый метод будет рассмотрен в этой главе, а другие методы — в главах 15 и 16.

Чтобы оптимизировать код R, необходимо понимать природу функционального программирования в R, а также механизм использования памяти в R.

14.2. Ужасающий цикл for

В списке рассылки `r-help` часто встречаются вопросы о том, как выполнять различные задачи без циклов `for`. Похоже, существует мнение, что программисты должны любой ценой избегать использования этих циклов[1]. Важно понимать, что простое переписывание кода для исключения циклов не обязательно ускорит работу кода. Тем не менее в некоторых случаях можно добиться кардинального ускорения работы программы, обычно за счет векторизации.

14.2.1. Векторизация и ускорение выполнения кода

Иногда вместо циклов можно воспользоваться векторизацией. Например, если `x` и `y` — векторы равной длины, можно использовать следующую команду:

```
z <- x + y
```

Такая запись не только более компактна — что еще важнее, она работает быстрее, чем следующий цикл:

```
for (i in 1:length(x)) z[i] <- x[i] + y[i]
```

Проведем сравнительный хронометраж:

```
> x <- runif(1000000)
> y <- runif(1000000)
> z <- vector(length=1000000)
> system.time(z <-x+y)
   user system elapsed
  0.052  0.016  0.068
```

[1] Пользователи, обращающиеся с такими вопросами, обычно стремятся ускорить работу своего кода.

```
> system.time(for (i in 1:length(x)) z[i] <- x[i] + y[i])
   user system elapsed
  8.088  0.044  8.175
```

Вот это различие! Версия без цикла работает более чем в 120 раз быстрее. Хотя конкретные измерения могут изменяться от запуска к запуску (при втором запуске версии с циклом было получено время 22,958), в некоторых случаях «расцикливание» кода R действительно окупается.

Стоит обсудить некоторые причины замедления версии с циклом. Это может быть неочевидно для программистов, пришедших в R из других языков, но в версии с циклом задействованы многочисленные вызовы функций:

- Хотя с точки зрения синтаксиса код выглядит безобидно, `for()` на самом деле является функцией.
- Двоеточие : выглядит еще безобиднее, но и оно является функцией. Например, `1:10` в действительности означает вызов функции : для аргументов 1 и 10:

  ```
  > ":"(1,10)
  [1]  1  2  3  4  5  6  7  8  9 10
  ```

- Каждая операция индексирования вектора представляет собой вызов функции: [для двух чтений и [<- для записи.

Вызовы функций могут занимать много времени, так как они требуют создания кадра стека и подобных операций. Временные потери при каждой итерации цикла приводят к значительному замедлению.

С другой стороны, если написать весь этот код на C, то никаких вызовов функций в нем не будет. На самом деле именно это происходит в первом фрагменте кода. В нем присутствуют вызовы функций (а именно + и ->), но каждая функция вызывается только один раз, а не 1 000 000 раз, как в версии с циклом. Из-за этого первая версия кода работает намного быстрее.

Одной из разновидностей векторизации является фильтрация векторов. Для примера перепишем функцию `oddcount()` из раздела 1.3:

```
oddcount <- function(x) return(sum(x%%2==1))
```

Явный цикл здесь не задействован, и хотя R во внутренней реализации перебирает содержимое массива, это будет сделано в машинном коде. И снова выполнение программы ускоряется, как и следовало ожидать.

```
> x <- sample(1:1000000,100000,replace=T)
> system.time(oddcount(x))
```

```
   user  system elapsed
  0.012   0.000   0.015
> system.time(
+    {
+    c <- 0
+    for (i in 1:length(x))
+       if (x[i] %% 2 == 1) c <- c+1
+    return(c)
+    }
+ )
   user  system elapsed
  0.308   0.000   0.310
```

Насколько это важно — ведь даже версия с циклом выполняется менее секунды? Но если код будет включен во внешний цикл с множеством итераций, различия могут быть достаточно важными.

Другие примеры векторизованных функций, которые могут ускорять выполнение кода: `ifelse()`, `which()`, `where()`, `any()`, `all()`, `cumsum()` и `cumprod()`. Для матриц можно использовать функции `rowSums()`, `colSums()` и т. д. В задачах типа «для всех возможных комбинаций» функции `combin()`, `outer()`, `lower.tri()`, `upper.tri()` или `expand.grid()` могут делать именно то, что вам нужно.

Хотя функция `apply()` устраняет внешний цикл, она в действительности реализована на R, а не на C, так что ее применение не ускорит ваш код. Но, как и другие разновидности `apply` (такие, как `lapply()`), может существенно способствовать ускорению кода.

14.2.2. Расширенный пример: ускорение моделирования методом Монте-Карло

В некоторых приложениях код моделирования может выполняться по несколько часов, дней и даже месяцев, так что возможности ускорения очень важны. В этом разделе рассматриваются два примера моделирования.

Для начала рассмотрим следующий код из раздела 8.6:

```
sum <- 0
nreps <- 100000
for (i in 1:nreps) {
   xy <- rnorm(2) # generate 2 N(0,1)s
   sum <- sum + max(xy)
}
print(sum/nreps)
```

Измененная версия того же кода (хочется надеяться, ускоренная):

```
nreps <- 100000
xymat <- matrix(rnorm(2*nreps),ncol=2)
maxs <- pmax(xymat[,1],xymat[,2])
print(mean(maxs))
```

В этом коде все случайные переменные генерируются одновременно и сохраняются в матрице `xymat`, с одной парой (X,Y) на строку:

```
xymat <- matrix(rnorm(2*nreps),ncol=2)
```

Затем мы находим все значения max(X,Y), сохраняем эти значения в `maxs`, после чего просто вызываем `mean()`.

Такое решение проще реализуется, и скорее всего, оно будет быстрее. Давайте убедимся в этом. Исходный код хранится в файле MaxNorm.R, а улучшенная версия — в файле MaxNorm2.R.

```
> system.time(source("MaxNorm.R"))
[1] 0.5667599
   user  system  elapsed
  1.700   0.004    1.722
> system.time(source("MaxNorm2.R"))
[1] 0.5649281
   user  system  elapsed
  0.132   0.008    0.143
```

И снова выполнение кода значительно ускоряется.

ПРИМЕЧАНИЕ

Повышение скорости достигается за счет увеличения затрат памяти — мы сохраняем случайные числа в массиве, вместо того чтобы генерировать и терять их пары. Как упоминалось ранее, компромисс между временем/затратами памяти достаточно часто встречается в мире программирования и в мире программирования R в частности.

В этом примере был достигнут превосходный прирост скорости, но простота обманчива. Рассмотрим более сложный пример.

Наш следующий пример — классическое упражнение из начального курса теории вероятностей. В урне 1 лежат 10 синих и 8 желтых шаров, а в урне 2 — 6 синих и 6 желтых. Мы наугад тянем шар из урны 1, перемещаем его в урну 2, а затем наугад тянем шар из урны 2. С какой вероятностью этот второй шар

окажется синим? Ответ легко найти аналитическим способом, но мы воспользуемся моделированием. Тривиальное решение выглядит так:

```
# Выполнить nreps повторений эксперимента для оценки
# P(извлечение синего шара из урны 2)
sim1 <- function(nreps) {
   nb1 <- 10 # 10 синих шаров в урне 1
   n1 <- 18 # Количество шаров в урне 1 при первом выборе
   n2 <- 13 # Количество шаров в урне 2 при втором выборе
   count <- 0 # Количество попыток с извлечением синего шара из урны 2
   for (i in 1:nreps) {
      nb2 <- 6 # Изначально 6 синих шаров в урне 2
      # Взять шар из урны 1 и положить в урну 2; шар синий?
      if (runif(1) < nb1/n1) nb2 <- nb2 + 1
      # Взять шар из урны 2; шар синий?
      if (runif(1) < nb2/n2) count <- count + 1
   }
   return(count/nreps) # Оценка P(извлечение синего шара из урны 2)
}
```

А вот как то же самое делается без циклов, с использованием `apply()`:

```
sim2 <- function(nreps) {
   nb1 <- 10
   nb2 <- 6
   n1 <- 18
   n2 <- 13
   # Заранее сгенерировать случайные числа, по одной строке на повторение
   u <- matrix(c(runif(2*nreps)),nrow=nreps,ncol=2)
   # Определить simfun для apply(); моделирует одно повторение
   simfun <- function(rw) {
      # rw ("row") – пара случайных чисел
      # Выбрать из урны 1
      if (rw[1] < nb1/n1) nb2 <- nb2 + 1
      # Выбрать из урны 2 и вернуть флаг выбора синего шара
      return (rw[2] < nb2/n2)
   }
   z <- apply(u,1,simfun)
   # z – вектор логических значений, интерпретируемых как 1, 0 и т. д.
   return(mean(z))
}
```

Здесь создается матрица `u` с двумя столбцами U(0,1) случайных переменных. Первый столбец используется для моделирования извлечения шара из урны 1, а второй — из урны 2. Таким образом мы генерируем все случайные числа заранее, что может сэкономить значительное время, и все же главное здесь — это использование `apply()`. Для достижения этой цели наша функция `simfun()` работает

с одним повторением эксперимента, то есть с одной строкой u. После этого можно определить вызов для повторения всех nreps повторений при помощи apply().

Обратите внимание: так как функция simfun() объявляется с sim2(), локальные переменные sim2() — n1, n2, nb1 и nb2 — доступны как глобальные переменные simfun(). Кроме того, поскольку логический вектор будет автоматически преобразован R в 1 и 0, для определения части значений TRUE в векторе достаточно вызвать mean().

А теперь сравним быстродействие.

```
> system.time(print(sim1(100000)))
[1] 0.5086
   user system elapsed
  2.465 0.028 2.586
> system.time(print(sim2(10000)))
[1] 0.5031
   user system elapsed
  2.936  0.004   3.027
```

Несмотря на все преимущества функционального программирования, это решение с apply() не работает. Собственно, стало только хуже. Так как эффект может быть обусловлен дисперсией случайной выборки, я несколько раз выполнил код, но результат был неизменным.

Итак, рассмотрим векторизацию этого кода.

```
 1 sim3 <- function(nreps) {
 2    nb1 <- 10
 3    nb2 <- 6
 4    n1 <- 18
 5    n2 <- 13
 6    u <- matrix(c(runif(2*nreps)),nrow=nreps,ncol=2)
 7    # Создать вектор условий
 8    cndtn <- u[,1] <= nb1/n1 & u[,2] <= (nb2+1)/n2 |
 9             u[,1] > nb1/n1 & u[,2] <= nb2/n2
10    return(mean(cndtn))
11 }
```

Основная работа выполняется в следующем фрагменте:

```
cndtn <- u[,1] <= nb1/n1 & u[,2] <= (nb2+1)/n2 |
         u[,1] >  nb1/n1 & u[,2] <= nb2/n2
```

Чтобы разобраться в том, что здесь происходит, мы определили, какие условия приводят к выбору синего шара при второй попытке, запрограммировали их, а затем присвоили cndtn.

Вспомните, что <= и & являются функциями; по сути это векторные функции, поэтому они должны работать быстро. Безусловно, это ускоряет работу программы:

```
> system.time(print(sim3(10000)))
[1] 0.4987
   user  system elapsed
  0.060   0.016   0.076
```

Вообще говоря, метод, который мы использовали для ускорения кода, применим во многих других ситуациях с моделированием Монте-Карло. Тем не менее достаточно ясно, что аналог этой команды, вычисляющий `cndtn`, быстро усложняется даже для простых на первый взгляд приложений.

Кроме того, такой метод не подходит для ситуаций с неограниченным количеством фаз. Пример с шарами считается двухфазным (два столбца в матрице `u`).

14.2.3. Расширенный пример: генерирование матрицы степеней

Вспомните, как в разделе 9.1.7 генерировалась матрица степеней переменной-предиктора. Тогда мы использовали следующий код:

```
 1 # Формирует матрицу степеней вектора x до степени dg
 2 powers1 <- function(x,dg) {
 3    pw <- matrix(x,nrow=length(x))
 4    prod <- x # Текущее произведение
 5    for (i in 2:dg) {
 6       prod <- prod * x
 7       pw <- cbind(pw,prod)
 8    }
 9    return(pw)
10 }
```

Одна очевидная проблема заключается в том, что функция `cbind()` используется для построения выходной матрицы, столбец за столбцом. Такой способ построения обходится очень дорого в отношении времени выделения памяти. Намного эффективнее выделить сразу всю матрицу, хотя она и будет пустой, так как в этом случае время будет затрачено только на одну операцию выделения памяти.

```
 1 # Формирует матрицу степеней вектора x до степени dg
 2 powers2 <- function(x,dg) {
 3    pw <- matrix(nrow=length(x),ncol=dg)
```

```
 4    prod <- x # Текущее произведение
 5    pw[,1] <- prod
 6    for (i in 2:dg) {
 7       prod <- prod * x
 8       pw[,i] <- prod
 9    }
10    return(pw)
11 }
```

И в самом деле, `powers2()` работает гораздо быстрее:

```
> x <- runif(1000000)
> system.time(powers1(x,8))
   user  system elapsed
  0.776   0.356   1.334
> system.time(powers2(x,8))
   user  system elapsed
  0.388   0.204   0.593
```

Тем не менее `powers2()` все равно содержит цикл. Можно ли улучшить решение? Казалось бы, ситуация идеально подходит для функции `outer()`, которая вызывается в форме

```
outer(X,Y,FUN)
```

Этот вызов применяет функцию `FUN()` ко всем возможным парам элементов X и элементов Y. По умолчанию используется операция умножения.

Здесь можно использовать следующую запись:

```
powers3 <- function(x,dg) return(outer(x,1:dg,"^"))
```

Для каждой комбинации элемента `x` и элемента `1:dg` (что дает в сумме `length(x)×dg` комбинаций) `outer()` вызывает функцию возведения в степень `^` и помещает результаты в матрицу `length(x)×dg`. Это именно то, что нам нужно; вдобавок код получается довольно компактным. Но стал ли он работать быстрее?

```
> system.time(powers3(x,8))
   user  system elapsed
  1.336   0.204   1.747
```

Какое разочарование! Мы используем необычную функцию R, получаем очень компактный код — и при этом худшую производительность из всех трех функций.

И это еще не все. Вот что произойдет при попытке использовать функцию `cumprod()`:

```
> powers4
function(x,dg) {
   repx <- matrix(rep(x,dg),nrow=length(x))
   return(t(apply(repx,1,cumprod)))
}
> system.time(powers4(x,8))
   user  system elapsed
 28.106   1.120  83.255
```

В этом примере мы создали несколько копий `x`, поскольку степени числа `n` равны `cumprod(c(1,n,n,n...))`. Но несмотря на послушное использование двух функций R уровня C, быстродействие было катастрофическим.

Мораль: вопросы быстродействия порой бывают непредсказуемыми. Вам остается лишь вооружиться пониманием основных принципов, векторизации и аспектов работы с памятью, рассмотренных ниже, а затем экспериментировать с разными подходами.

14.3. Функциональное программирование и работа с памятью

Многие объекты R *неизменяемы*. А следовательно, операции R реализуются в виде функций, которые присваивают новое значение заданному объекту, — эта особенность может иметь последствия для быстродействия.

14.3.1. Присваивание векторов

В качестве примера некоторых из возникающих проблем рассмотрим простую на первый взгляд команду:

```
z[3] <- 8
```

Как упоминалось в главе 7, такое присваивание сложнее, чем может показаться. В действительности оно реализуется функцией замены `"[<-"` следующим вызовом и присваиванием:

```
z <- "[<-"(z,3,value=8)
```

Здесь создается внутренняя копия `z`, элемент 3 копии заменяется на 8, а полученный вектор снова присваивается `z`. Как говорилось ранее, это означает, что в `z` сохраняется ссылка на копию.

Другими словами, хотя на первый взгляд изменяется всего один элемент вектора, в соответствии с семантикой весь вектор будет вычислен заново. Для длинного вектора это может существенно замедлить работу программы. То же произойдет и с более коротким вектором, если присваивание будет выполняться в цикле.

В некоторых ситуациях R принимает меры к тому, чтобы свести к минимуму эти последствия, но о них необходимо помнить, если вы стремитесь к быстроте кода. Помните об этом при работе с векторами (включая массивы). Если ваш код начинает работать неожиданно медленно, одним из первых «подозреваемых» должно стать присваивание векторов.

14.3.2. Копирование при изменении

С этой проблемой связана другая: R (обычно) следует политике *копирования при изменении*. Например, если выполнить следующую команду в описанной выше ситуации:

```
> y <- z
```

то изначально y занимает ту же область памяти, что и z. Но если одна из двух переменных изменится, то R создаст копию в другой области памяти и измененная переменная будет занимать новую область памяти. Тем не менее изменение коснется только первой копии, так как перемещение переменной исключает все проблемы, связанные с общей памятью. Функция `tracemem()` сообщает о таких перемещениях в памяти.

Хотя R обычно придерживается семантики копирования при изменении, из этого правила существуют исключения. Например, R не применяет перемещение в следующей ситуации:

```
> z <- runif(10)
> tracemem(z)
[1] "<0x88c3258>"
> z[3] <- 8
> tracemem(z)
[1] "<0x88c3258>"
```

Местонахождение z не изменилось; данные находились по адресу 0x88c3258 как до, так и после присваивания `z[3]`. А значит, хотя вы должны учитывать возможность перемещения, быть полностью уверенным в нем нельзя. Посмотрим, сколько времени занимает операция.

```
> z <- 1:10000000
> system.time(z[3] <- 8)
   user  system elapsed
  0.180  0.084   0.265
> system.time(z[33] <- 88)
   user  system elapsed
      0       0       0
```

Как бы то ни было, копирование выполняется внутренней функцией R `duplicate()`. (В последних версиях R эта функция называется `duplicate1()`.) Если вы знакомы с отладочной программой GDB, а ваша сборка R включает отладочную информацию, вы сможете исследовать обстоятельства, при которых выполнялось копирование.

Запустите R с GDB в соответствии с инструкциями в разделе 15.1.4, проведите пошаговое выполнение R через GDB и установите точку прерывания в `duplicate1()`. Каждый раз, когда в этой функции срабатывает точка прерывания, вводите следующую команду GDB:

```
call Rf_PrintValue(s)
```

Она выведет значение `s` (или другой интересующей вас переменной).

14.3.3. Расширенный пример: предотвращение копирования в памяти

Рассмотренный пример может показаться искусственным, но он демонстрирует проблемы копирования в памяти, рассмотренные в предыдущем разделе.

Допустим, имеется множество не связанных друг с другом векторов, и среди прочего вы хотите задать третий элемент каждого вектора равным 8. Векторы можно сохранить в матрице, по одному на строку. Но поскольку векторы не связаны, а возможно, даже имеют разную длину, можно подумать о том, чтобы сохранить их в списке.

Но когда речь заходит о вопросах быстродействия R, появляется множество нюансов, поэтому проведем эксперимент.

```
> m <- 5000
> n <- 1000
> z <- list()
> for (i in 1:m) z[[i]] <- sample(1:10,n,replace=T)
> system.time(for (i in 1:m) z[[i]][3] <- 8)
   user  system elapsed
  0.288   0.024   0.321
```

```
> z <- matrix(sample(1:10,m*n,replace=T),nrow=m)
> system.time(z[,3] <- 8)
   user  system elapsed
  0.008   0.044   0.052
```

Если не считать системного времени (опять), матричная версия работает более эффективно.

Одна из причин заключается в том, что в списковой версии мы сталкиваемся с проблемой копирования в памяти при каждой итерации цикла. Но в матричной версии она встречается только один раз. И конечно, матричная версия векторизована. А как насчет использования `lapply()` в списковой версии?

```
>
> set3 <- function(lv) {
+    lv[3] <- 8
+    return(lv)
+ }
> z <- list()
> for (i in 1:m) z[[i]] <- sample(1:10,n,replace=T)
> system.time(lapply(z,set3))
   user  system elapsed
  0.100   0.012   0.112
```

Очень трудно превзойти векторизованный код по быстродействию.

14.4. Использование Rprof() для поиска мест замедления в коде

Если вы думаете, что код R работает слишком медленно, для поиска виновника можно воспользоваться удобной функцией `Rprof()`, которая выдает (приблизительный) отчет с информацией о том, сколько времени ваш код проводит в каждой из вызываемых функций. Это очень важная информация, потому что оптимизировать все до исключения части вашей программы может быть неразумно. За оптимизацию приходится расплачиваться временем программирования и чистотой кода, поэтому будет полезно знать, где оптимизация действительно поможет.

14.4.1. Мониторинг с использованием Rprof()

Применим `Rprof()` с тремя версиями нашего кода для нахождения матрицы степеней из предыдущего расширенного примера. Вызовите `Rprof()`, чтобы

запустить программу-монитор, запустите код, а затем вызовите `Rprof()` с аргументом `NULL`, чтобы остановить мониторинг. Наконец, вызовите `summaryRprof()`, чтобы просмотреть результаты.

```
> x <- runif(1000000)
> Rprof()
> invisible(powers1(x,8))
> Rprof(NULL)
> summaryRprof()
$by.self
         self.time self.pct total.time total.pct
"cbind"       0.74     86.0       0.74      86.0
"*"           0.10     11.6       0.10      11.6
"matrix"      0.02      2.3       0.02       2.3
"powers1"     0.00      0.0       0.86     100.0

$by.total
         total.time total.pct self.time self.pct
"powers1"       0.86     100.0      0.00      0.0
"cbind"         0.74      86.0      0.74     86.0
"*"             0.10      11.6      0.10     11.6
"matrix"        0.02       2.3      0.02      2.3

$sampling.time
[1] 0.86
```

Сразу же видно, что во времени выполнения кода основное место занимают вызовы `cbind()`, которые, как было показано в расширенном примере, действительно замедляют выполнение.

Кстати говоря, вызов `invisible()` в этом примере использовался для подавления вывода. Поверьте, вам не захочется видеть матрицу с 1 000 000 строк, которую возвращает `powers1()`!

Профилирование `powers2()` не показывает никаких явных узких мест при выполнении.

```
> Rprof()
> invisible(powers2(x,8))
> Rprof(NULL)
> summaryRprof()
$by.self
          self.time self.pct total.time total.pct
"powers2"      0.38     67.9       0.56     100.0
"matrix"       0.14     25.0       0.14      25.0
"*"            0.04      7.1       0.04       7.1

$by.total
```

```
         total.time total.pct self.time self.pct
"powers2"       0.56     100.0      0.38     67.9
"matrix"        0.14      25.0      0.14     25.0
"*"             0.04       7.1      0.04      7.1
$sampling.time
[1] 0.56
```

А как насчет `powers3()` — многообещающего решения, которое не сработало?

```
> Rprof()
> invisible(powers3(x,8))
> Rprof(NULL)
> summaryRprof()
$by.self
           self.time self.pct total.time total.pct
"FUN"            0.94     56.6       0.94      56.6
"outer"          0.72     43.4       1.66     100.0
"powers3"        0.00      0.0       1.66     100.0

$by.total
           total.time total.pct self.time self.pct
"outer"          1.66     100.0      0.72     43.4
"powers3"        1.66     100.0      0.00      0.0
"FUN"            0.94      56.6      0.94     56.6

$sampling.time
[1] 1.66
```

Функция сообщает, что самые большие затраты времени приходятся на функцию `FUN()`, которая, как указано в расширенном примере, просто выполняла умножение. Для каждой пары элементов `x` один из элементов умножался на другой; так вычисляется произведение двух скаляров. А значит, никакой векторизации! Неудивительно, что код работает медленно.

14.4.2. Как работает Rprof()

Давайте чуть подробнее разберемся в том, что делает `Rprof()`. Каждые 0,02 секунды (по умолчанию) R проверяет стек вызовов, чтобы определить, какие вызовы функций активны на данный момент. Результат каждого анализа записывается в файл — по умолчанию Rprof.out. Фрагмент этого файла для нашего запуска `powers3()`:

```
...
"outer" "powers3"
"outer" "powers3"
"outer" "powers3"
```

```
"FUN" "outer" "powers3"
"FUN" "outer" "powers3"
"FUN" "outer" "powers3"
"FUN" "outer" "powers3"
...
```

Итак, `Rprof()` часто обнаруживает, что во время проверки `powers3()` вызывает функцию `outer()`, которая, в свою очередь, вызывает `FUN()`; последняя и является текущей выполняемой функцией. Удобная функция `summaryRprof()` выдает сводку содержимого файла, но, возможно, в каких-то случаях просмотр самого файла предоставит больше полезной информации.

К сожалению, `Rprof()` не панацея. Если код, который вы профилируете, инициирует много вызовов функций (включая косвенные вызовы, когда ваш код вызывает некоторую функцию, которая вызывает другую функцию в R), вывод профилировщика может быть достаточно трудно расшифровать. Пожалуй, с выводом `powers4()` дело обстоит именно так:

```
$by.self
             self.time self.pct total.time total.pct
"apply"           19.46     67.5      27.56      95.6
"lapply"           4.02     13.9       5.68      19.7
"FUN"              2.56      8.9       2.56       8.9
"as.vector"        0.82      2.8       0.82       2.8
"t.default"        0.54      1.9       0.54       1.9
"unlist"           0.40      1.4       6.08      21.1
"!"                0.34      1.2       0.34       1.2
"is.null"          0.32      1.1       0.32       1.1
"aperm"            0.22      0.8       0.22       0.8
"matrix"           0.14      0.5       0.74       2.6
"!="               0.02      0.1       0.02       0.1
"powers4"          0.00      0.0      28.84     100.0
"t"                0.00      0.0      28.10      97.4
"array"            0.00      0.0       0.22       0.8

$by.total
             total.time total.pct self.time self.pct
"powers4"         28.84     100.0      0.00      0.0
"t"               28.10      97.4      0.00      0.0
"apply"           27.56      95.6     19.46     67.5
"unlist"           6.08      21.1      0.40      1.4
"lapply"           5.68      19.7      4.02     13.9
"FUN"              2.56       8.9      2.56      8.9
"as.vector"        0.82       2.8      0.82      2.8
"matrix"           0.74       2.6      0.14      0.5
"t.default"        0.54       1.9      0.54      1.9
"!"                0.34       1.2      0.34      1.2
```

```
"is.null"           0.32       1.1      0.32      1.1
"aperm"             0.22       0.8      0.22      0.8
"array"             0.22       0.8      0.00      0.0
"!="                0.02       0.1      0.02      0.1

$sampling.time
[1] 28.84
```

14.5. Компиляция в байт-код

Начиная с версии 2.13, в R был включен компилятор в байт-код, который может использоваться для ускорения кода. Возьмем пример из раздела 14.2.1. В качестве тривиального примера мы показали, что команда

```
z <- x + y
```

работает намного быстрее

```
for (i in 1:length(x)) z[i] <- x[i] + y[i]
```

И это было очевидно, но просто для того, чтобы вы получили представление о том, как работает компиляция в байт-код, проведем эксперимент:

```
> library(compiler)
> f <- function() for (i in 1:length(x)) z[i] <<- x[i] + y[i]
> cf <- cmpfun(f)
> system.time(cf())
   user  system elapsed
  0.845   0.003   0.848
```

Мы создали новую функцию `cf()` на базе исходной функции `f()`. Новый код выполняется за 0,848 секунды — намного быстрее 8,175 секунды для некомпилированной версии. Да, новая версия все равно выполнялась медленнее прямолинейного векторизированного кода, но совершенно ясно, что компиляция в байт-код перспективна. Попробуйте воспользоваться ею, если вам понадобится ускорить выполнение кода.

14.6. О нет, данные не помещаются в памяти!

Как упоминалось ранее, все объекты в сеансе R хранятся в памяти. R устанавливает предел $2^{31}-1$ байт для размера любого объекта независимо от размера слова (32- или 64-разрядное) и объема памяти на вашем компьютере. Тем не менее не стоит считать это серьезным препятствием. При некоторых дополнительных

усилиях R нормально справляется с высокими требованиями к памяти. Среди стандартных методов можно выделить блочную загрузку данных и использование пакетов R для управления памятью.

14.6.1. Создание блоков

Одно из решений, не требующих дополнительных пакетов R, — чтение данных из файла на диске по блокам. Представьте, что вам понадобилось вычислить среднее значение или пропорции для некоторых переменных. В таком случае можно воспользоваться аргументом `skip` функции `read.table()`.

Допустим, набор данных состоит из 1 000 000 записей, которые будут разделены на 10 блоков (или больше — сколько потребуется для разбиения данных до такого размера, чтобы они помещались в памяти). При первом чтении задается аргумент `skip = 0`, при втором — `skip = 100000` и т. д. Каждый раз при чтении фрагмента программа вычисляет количество вхождений или сумму для данного блока и сохраняет их. После чтения всех блоков все счетчики или суммы объединяются для вычисления общего среднего значения или пропорций.

Другой пример: представьте, что вы выполняете статистическую операцию (допустим, вычисление основных компонентов) с очень большим количеством строк (то есть наблюдений), но количество переменных при этом невелико. И снова блоки могут стать хорошим решением. Статистическая операция применяется к каждому блоку, после чего результаты усредняются по всем блокам. Мои математические исследования показывают, что полученные оценки статистически эффективны для широкого класса статистических методов.

14.6.2. Применение пакетов R для управления памятью

Если вы ищете более технологичное решение, для ситуаций с высокими требованиями к памяти существует альтернатива в виде специализированных пакетов R.

Одним из таких пакетов является RMySQL, R-интерфейс для баз данных SQL. Для использования RMySQL требуется некоторый опыт работы с базами данных, но этот пакет предоставляет намного более эффективные и удобные средства для работы с большими наборами данных. Идея заключается в том, чтобы операции выбора выполнялись в SQL на уровне базы данных, а полученные выбранные данные читались как результаты выполнения команд SQL.

Так как последние обычно намного меньше общего набора данных, скорее всего, вам удастся обойти ограничения памяти R.

Другой полезный пакет `biglm` выполняет регрессионный анализ и анализ обобщенных линейных моделей для очень больших наборов данных. Он также использует блочную загрузку, но несколько иным способом: каждый блок используется для обновления накапливаемых сумм, необходимых для регрессионного анализа, а затем удаляется.

Наконец, некоторые пакеты организуют собственное управление памятью независимо от R, а следовательно, могут справиться с очень большими наборами данных. Самыми распространенными из них являются `ff` и `bigmemory`. Первый обходит ограничения памяти за счет хранения данных на диске вместо памяти, причем происходит это прозрачно для программиста. Чрезвычайно гибкий пакет `bigmemory` делает то же самое, но может хранить данные не только на диске, но и в основной памяти компьютера; это решение идеально подходит для многоядерных машин.

15 Взаимодействие R с другими языками

R — замечательный язык, но он не может хорошо справляться с любыми задачами. А значит, иногда возникает необходимость в вызове из R кода, написанного на других языках. И наоборот, при работе на других замечательных языках иногда встречаются задачи, которые лучше было бы реализовать на R.

Интерфейсы R были разработаны для множества других языков, от повсеместно распространенных (таких, как C) до экзотических и узкоспециализированных — например, системы компьютерной алгебры Yacas. В этой главе будут рассмотрены два интерфейса: для вызова кода C/C++ из R и для вызова R из Python.

15.1. Написание функций C/C++ для вызова из R

Возможно, вы захотите написать на C/C++ собственную функцию, которая должна вызываться из R. Как правило, целью является повышение быстродействия, поскольку код C/C++ может выполняться намного быстрее кода R даже при применении векторизации и других средств оптимизации R для ускорения кода.

Другая возможная причина для перехода на уровень C/C++ — специализированный ввод/вывод. Например, R использует протокол TCP на уровне 3 стандартной коммуникационной системы интернета, но протокол UDP быстрее работает в некоторых ситуациях. Для работы с UDP необходим язык C/C++, а для этого потребуется интерфейс R для таких языков.

На самом деле R предоставляет два интерфейса C/C++ в виде функций `.C()` и `.Call()`. Вторая функция более универсальна, но она требует некоторого знания внутренних структур R, поэтому здесь мы ограничимся `.C()`.

15.1.1. Что нужно знать для взаимодействия R с C/C++

В языке C двумерные массивы хранятся по строкам — в отличие от способа хранения по столбцам, принятого в R. Например, если у вас имеется массив 3 × 4, элемент во второй строке и втором столбце будет элементом 5 при линейном просмотре, так как первый столбец содержит три элемента, а искомый элемент будет вторым во втором столбце. Также следует помнить, что индексы C начинаются с 0, а не с 1, как в R.

Все аргументы, передаваемые из R в C, передаются C в виде указателей. Учтите, что сама функция C должна возвращать `void`. Значения, которые обычно должны возвращаться функциями, передаются через аргументы функций (как `result` в следующем примере).

15.1.2. Пример: извлечение поддиагоналей квадратной матрицы

В этом разделе мы напишем код C для извлечения поддиагоналей квадратной матрицы (спасибо моему бывшему аспиранту Минь-Ю Хуану (Min-Yu Huang), написавшему более раннюю версию этой функции). Код из файла **sd.c**:

```
#include <R.h> // Обязательно
// Аргументы:
//     m: квадратная матрица
//     n: количество строк/столбцов в m
//     k: индекс поддиагонали--0 для главной диагонали, 1 для первой
//         поддиагонали, 2 для второй и т. д.
//     result: место для запрашиваемой поддиагонали
void subdiag(double *m, int *n, int *k, double *result)
{
   int nval = *n, kval = *k;
   int stride = nval + 1;
   for(inti=0,j= kval; i < nval-kval; ++i, j+= stride)
      result[i] = m[j];
}
```

Переменная `stride` относится к одной из концепций параллельной обработки. Допустим, имеется матрица из 1000 столбцов, а код C перебирает все элементы заданного столбца сверху донизу. И снова, поскольку C использует порядок хранения по строкам, смежные элементы столбца расположены в 1000 элементах друг от друга, если рассматривать матрицу как один длинный вектор.

Здесь обход длинного вектора осуществляется с шагом 1000, то есть обращением к каждому 1000-му элементу.

15.1.3. Компиляция и запуск кода

Для компиляции кода используется R. Например, в окне терминала Linux файл компилируется примерно так:

```
% R CMD SHLIB sd.c
gcc -std=gnu99 -I/usr/share/R/include -fpic -g -O2 -c sd.c -o sd.o
gcc -std=gnu99 -shared -o sd.so sd.o -L/usr/lib/R/lib -lR
```

В результате будет создан файл динамической общей библиотеки sd.so.

Обратите внимание: в выводе этого примера R сообщает о вызове GCC. Эти команды также можно выполнить вручную при наличии особых требований — например, необходимости компоновки специальных библиотек. Также следует учесть, что местонахождение каталогов include и lib также может зависеть от системы.

ПРИМЕЧАНИЕ

GCC для систем Linux легко найти в интернете. Для Windows компилятор включен в Cygwin — пакет с открытым кодом, доступный по адресу http://www.cygwin.com/.

Далее остается загрузить библиотеку в R и вызвать функцию C:

```
> dyn.load("sd.so")
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
>k<-2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
result=double(dim(m)[1]-k))
[[1]]
 [1] 1 6 11 16 21 2 7 12 17 22 3 8 13 18 23 4 9 14 19 24 5 10 15 20 25

[[2]]
[1] 5

[[3]]
[1] 2

$result
[1] 11 17 23
```

Для удобства мы присвоили имя `result` как формальному аргументу (в коде C), так и фактическому аргументу (в коде R). Обратите внимание на необходимость выделения памяти в `result` в коде R.

Как видно из примера, возвращаемое значение принимает форму списка, состоящего из аргументов в вызове R. В данном случае вызов имеет четыре аргумента (кроме имени функции), так что возвращаемый список содержит четыре компонента. Обычно некоторые аргументы изменяются в ходе выполнения кода C, как в данном случае с `result`.

15.1.4. Отладка кода R/C

В главе 13 рассмотрены некоторые инструменты и методы отладки кода R. Тем не менее интерфейс R/C создает дополнительные трудности. Проблема при использовании таких отладочных инструментов, как GDB, заключается в том, что сначала их необходимо применить к самой среде R.

Ниже приведен протокол действий по отладке R/C при использовании GDB с предшествующим кодом sd.c:

```
$ R -d gdb
GNU gdb 6.8-debian
...
(gdb) run
Starting program: /usr/lib/R/bin/exec/R
...
> dyn.load("sd.so")
> # hit ctrl-c here
Program received signal SIGINT, Interrupt.
0xb7ffa430 in __kernel_vsyscall ()
(gdb) b subdiag
Breakpoint 1 at 0xb77683f3: file sd.c, line 3.
(gdb) continue
Continuing.

Breakpoint 1, subdiag (m=0x92b9480, n=0x9482328, k=0x9482348, result=0x9817148)
    at sd.c:3
3           int nval = *n, kval = *k;
(gdb)
```

Что же произошло в этом сеансе отладки?

1. Мы запустили отладчик GDB и загрузили в нем R следующей командой в окне терминала:

   ```
   R -d gdb
   ```

2. Мы приказали GDB запустить R:
   ```
   (gdb) run
   ```
3. Откомпилированный код C был загружен в R обычным способом:
   ```
   > dyn.load("sd.so")
   ```
4. Нажатие клавиш Ctrl+C приостанавливает выполнение R и возвращает нас к приглашению GDB.
5. На входе в `subdiag()` устанавливается точка прерывания:
   ```
   (gdb) b subdiag
   ```
6. Мы приказываем GDB продолжить выполнение R (для возвращения к приглашению R необходимо повторное нажатие клавиши Enter):
   ```
   (gdb) continue
   ```

Затем был выполнен код C:

```
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
>k<-2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
+ result=double(dim(m)[1]-k))

Breakpoint 1, subdiag (m=0x942f270, n=0x96c3328, k=0x96c3348, result=0x9a58148)
   at subdiag.c:46
46 if (*n < 1) error("n < 1\n");
```

На этой стадии можно использовать GDB для отладки как обычно. Если вы не знакомы с GDB, воспользуйтесь одним из многочисленных учебников в интернете. Самые полезные команды перечислены в табл. 15.1.

Таблица 15.1. Основные команды GDB

Команда	Описание
l	Вывести строки кода
b	Установить точку прерывания
r	Запустить / запустить повторно
n	Перейти к следующей команде
s	Зайти в вызов функции
p	Вывести переменную или выражение
c	Продолжить
h	Вывести справку
q	Выйти

15.1.5. Расширенный пример: прогнозирование временных рядов с дискретными значениями

Вспомните пример из раздела 2.5.2, в котором мы наблюдали данные со значениями 0 и 1 в течение некоторого промежутка времени, а потом пытались спрогнозировать значение за любой период времени по k предыдущим значениям. Мы разработали две альтернативные функции для решения этой задачи, `preda()` и `predb()`:

```
# Прогнозирование дискретных временных рядов из 0 и 1; k последовательных
# наблюдений используются для предсказания следующего значения; вычисление
# частоты ошибок
preda <- function(x,k) {
   n <- length(x)
   k2 <- k/2
   # Вектор pred будет содержать прогнозируемые значения
   pred <- vector(length=n-k)
   for (i in 1:(n-k)) {
      if (sum(x[i:(i+(k-1))]) >= k2) pred[i] <- 1 else pred[i] <- 0
   }
   return(mean(abs(pred-x[(k+1):n])))
}

predb <- function(x,k) {
   n <- length(x)
   k2 <- k/2
   pred <- vector(length=n-k)
   sm <- sum(x[1:k])
   if (sm >= k2) pred[1] <- 1 else pred[1] <- 0
   if (n-k >= 2) {
      for (i in 2:(n-k)) {
         sm <- sm + x[i+k-1] - x[i-1]
         if (sm >= k2) pred[i] <- 1 else pred[i] <- 0
      }
   }
   return(mean(abs(pred-x[(k+1):n])))
}
```

Так как вторая функция избегала повторяющихся вычислений, мы предположили, что она будет работать быстрее. Пришло время убедиться в этом.

```
> y <- sample(0:1,100000,replace=T)
> system.time(preda(y,1000))
   user  system elapsed
  3.816   0.016   3.873
```

```
> system.time(predb(y,1000))
   user  system elapsed
  1.392   0.008   1.427
```

Неплохо! Улучшение весьма заметное.

Тем не менее всегда следует поинтересоваться, нет ли в R оптимизированной функции, подходящей для ваших целей. Так как фактически речь идет о вычислении скользящего среднего, попробуем воспользоваться функцией `filter()` с постоянным вектором коэффициентов:

```
predc <- function(x,k) {
   n <- length(x)
   f <- filter(x,rep(1,k),sides=1)[k:(n-1)]
   k2 <- k/2
   pred <- as.integer(f >= k2)
   return(mean(abs(pred-x[(k+1):n])))
}
```

Код получается еще более компактным, чем в первой версии. При этом читается она намного хуже и по причинам, которые будут изложены ниже, может работать медленнее. Проверим, так ли это:

```
> system.time(predc(y,1000))
   user  system elapsed
  3.872   0.016   3.945
```

Вторая версия пока остается непревзойденной. На самом деле этого следовало ожидать, как показывает просмотр исходного кода. Следующая команда выводит код этой функции:

```
> filter
```

Из кода (здесь не показан) видно, что в нем вызывается функция `filter1()`. Последняя написана на C, что должно обеспечить некоторый прирост скорости, но и она страдает от проблемы дублирующихся вычислений, отсюда и замедление.

Напишем собственный код C:

```
#include <R.h>
void predd(int *x, int *n, int *k, double *errrate)
{
   int nval = *n, kval = *k, nk = nval - kval, i;
   int sm = 0; // Скользящая сумма
   int errs = 0; // Счетчик ошибок
   int pred; // Прогнозируемое значение
```

```
   double k2 = kval/2.0;
   // Инициализировать вычислением исходного окна
   for (i = 0; i < kval; i++) sm += x[i];
   if (sm >= k2) pred = 1; else pred = 0;
   errs = abs(pred-x[kval]);
   for (i = 1; i < nk; i++) {
      sm = sm + x[i+kval-1] - x[i-1];
      if (sm >= k2) pred = 1; else pred = 0;
      errs += abs(pred-x[i+kval]);
   }
   *errrate = (double) errs / nk;
   }
```

Фактически это та же функция `predb()`, «вручную» переведенная на C. Посмотрим, превзойдет ли она `predb()`:

```
> system.time(.C("predd",as.integer(y),as.integer(length(y)),as.integer(1000),
+  errrate=double(1)))
  user  system elapsed
 0.004   0.000   0.003
```

Ускорение просто потрясающее!

Как видите, программирование некоторых функций на C может оправдать потраченные усилия. Это особенно справедливо для функций, связанных с перебором, поскольку собственные циклические конструкции R (такие, как `for()`) работают медленно.

15.2. Использование R из Python

Python — элегантный и мощный язык, но ему не хватает встроенных средств для статистических вычислений и обработки данных — двух областей, в которых язык R особенно силен. В этом разделе показано, как вызывать код R из Python при помощи `RPy`, одного из самых популярных интерфейсов между двумя языками.

15.2.1. Установка RPy

RPy — модуль Python, предоставляющий доступ к R из Python. Для повышения эффективности его можно использовать в сочетании с `NumPy`.

Модуль можно построить из исходного кода (http://rpy.sourceforge.net) или загрузить его заранее построенную версию. Если вы работаете в Ubuntu, просто введите следующую команду:

```
sudo apt-get install python-rpy
```

Чтобы загрузить RPy из Python (в интерактивном режиме Python или из кода), выполните следующую команду:

```
from rpy import *
```

Команда загружает переменную r — экземпляр класса Python.

15.2.2. Синтаксис RPy

Выполнять R из Python, в принципе, несложно. Пример выполнения команды из приглашения Python >>>:

```
>>> r.hist(r.rnorm(100))
```

Команда вызывает R функцию rnorm() для получения 100 величин со стандартным нормальным распределением, а потом передает эти значения функции построения R гистограмм hist(). Как видите, имена R снабжаются префиксом r.; он отражает тот факт, что Python-обертки для функций R являются членами экземпляра класса r.

Приведенный код, если его не усовершенствовать, выдает уродливый вывод: ваши данные (возможно, довольно объемистые!) используются в заголовке гистограммы и в метке оси x. Чтобы избежать этого, передайте заголовок и метку, как в следующем примере:

```
>>> r.hist(r.rnorm(100),main='',xlab='')
```

Синтаксис RPy несколько сложнее, чем может показаться. Проблема в том, что синтаксис R может вступить в конфликт с синтаксисом Python. Возьмем вызов R функции линейной модели lm(). В нашем примере b прогнозируется на основании a.

```
>>> a = [5,12,13]
>>> b = [10,28,30]
>>> lmout = r.lm('v2 ~ v1',data=r.data_frame(v1=a,v2=b))
```

Вызов получается более сложным, чем если бы он был напрямую реализован на R. В чем дело?

Во-первых, так как синтаксис Python не включает символ ~, формулу модели пришлось задавать в виде строки. Так как это все равно делается в R, вряд ли это можно считать серьезным препятствием.

Во-вторых, нам нужен кадр данных для хранения данных. Этот кадр создается R функцией `data.frame()`. Чтобы сформировать точку в имени функции R, необходимо использовать символ подчеркивания на стороне Python. Таким образом вызывается функция `r.data_frame()`. В этом вызове столбцам кадра данных присвоены имена `v1` и `v2`, которые используются в формуле модели.

Выходным объектом является словарь Python (аналог спискового типа R), как показано ниже (приведена лишь часто вывода):

```
>>> lmout
{'qr': {'pivot': [1, 2], 'qr': array([[ -1.73205081, -17.32050808],
       [ 0.57735027, -6.164414 ],
       [ 0.57735027,  0.78355007]]), 'qraux':
```

Возможно, вы узнали различные атрибуты объектов `lm()`. Например, коэффициенты подогнанной линии регрессии, которые в R содержались бы в `lmout$coefficients`, в Python содержатся в `lmout['coefficients']`. К этим коэффициентам можно обращаться соответственно — например, так:

```
>>> lmout['coefficients']
{'v1': 2.5263157894736841, '(Intercept)': -2.5964912280701729}
>>> lmout['coefficients']['v1']
2.5263157894736841
```

Вы также можете выполнять команды R для работы с переменными в пространстве имен R при помощи функции `r()`. Это особенно удобно при большом количестве конфликтов синтаксиса. Вот как пример с `wireframe()` из раздела 12.4 был бы выполнен в RPy:

```
>>> r.library('lattice')
>>> r.assign('a',a)
>>> r.assign('b',b)
>>> r('g <- expand.grid(a,b)')
>>> r('g$Var3 <- g$Var1^2 + g$Var1 * g$Var2')
>>> r('wireframe(Var3 ~ Var1+Var2,g)')
>>> r('plot(wireframe(Var3 ~ Var1+Var2,g))')
```

Сначала `r.assign()` используется для копирования переменной из пространства имен Python в R. Затем выполняется `expand.grid()` (с точкой в имени вместо подчеркивания, так как мы работаем в пространстве имен R), а результат присваивается `g`. Последняя переменная также находится в пространстве имен R. Вызов `wireframe()` не приводит к автоматическому отображению графика, поэтому необходимо вызвать `plot()`.

Официальная документация `RPy` доступна по адресу http://rpy.sourceforge.net/rpy/doc/rpy.pdf. Полезная презентация «RPy—R from Python» доступна по адресу http://www.daimi.au.dk/~besen/TBiB2007/lecture-notes/rpy.html.

16 Параллелизм в R

У многих пользователей R вычислительные потребности очень велики, поэтому были разработаны различные средства реализации параллельных операций в R. Эта глава посвящена параллелизму в R.

Многие новички в области параллельного программирования с большими надеждами пишут параллельный код, чтобы потом обнаружить, что параллельная версия работает медленнее последовательной. По причинам, которые будут изложены в этой главе, данная проблема особенно актуальна для R.

А значит, для успеха в параллельном мире исключительно важно понимать аппаратное и программное обеспечение параллельной обработки. Эти вопросы будут рассматриваться здесь в контексте распространенных платформ для параллельной обработки R. Мы начнем с нескольких примеров кода, а затем перейдем к общим проблемам быстродействия.

16.1. Проблема взаимных исходящих связей

Возьмем сетевой граф — например, представляющий набор веб-страниц или ссылок в социальной сети. Пусть A — *матрица смежности* графа; например, элемент A[3,8] равен 1 или 0 в зависимости от того, существует ли ссылка из узла 3 в узел 8.

Для любых двух вершин (скажем, двух веб-сайтов) нас могут интересовать *взаимные исходящие связи*, то есть исходящие связи, общие для двух сайтов. Предположим, вы хотите найти среднее число исходящих связей, усредненное по всем парам сайтов в наборе данных. Нужное значение может быть вычислено по следующей схеме для матрицы $n \times n$:

```
1 sum = 0
2 for i = 0...n-1
```

```
3    for j = i+1...n-1
4       for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
5 mean = sum / (n*(n-1)/2)
```

А если учесть, что граф может содержать тысячи — и даже миллионы — сайтов, то такая задача потребует весьма значительных вычислений. Стандартный метод решения подобных задач заключается в разделении вычислений на меньшие блоки и последующей одновременной обработке всех блоков, например, на разных компьютерах.

Допустим, в вашем распоряжении два компьютера. Один компьютер может обрабатывать нечетные значения `i` в цикле `for` из строки 2, а второй компьютер обрабатывает четные значения. Или, поскольку двухъядерные компьютеры стали достаточно рядовым явлением в наши дни, тот же подход можно применить на одном компьютере. Звучит просто, но как вы узнаете в этой главе, в такой схеме могут возникнуть некоторые серьезные проблемы.

16.2. Пакет snow

Пакет Люка Тьерни (Luke Tierney) `snow` (Simple Network of Workstations), доступный в репозитории кода R CRAN, многими считается самой простой, удобной в использовании и одной из самых популярных форм реализации параллелизма в R.

ПРИМЕЧАНИЕ

На странице репозитория CRAN, посвященной параллелизму в R (http://cran.r-project.org/web/views/HighPerformanceComputing.html), приведен более или менее актуальный список параллельных пакетов R.

Чтобы вы поняли, как работает `snow`, рассмотрим код для задачи взаимных исходящих связей из предыдущего раздела:

```
1 # Версия задачи взаимных исходящих связей для snow
2
3 mtl <- function(ichunk,m) {
4    n <- ncol(m)
5    matches <- 0
6    for (i in ichunk) {
7       if(i<n){
8          rowi <- m[i,]
9          matches <- matches +
```

```
10              sum(m[(i+1):n,] %*% rowi)
11          }
12      }
13      matches
14  }
15
16  mutlinks <- function(cls,m) {
17      n <- nrow(m)
18      nc <- length(cls)
19      # Определить, какому рабочему потоку достается блок
20      options(warn=-1)
21      ichunks <- split(1:n,1:nc)
22      options(warn=0)
23      counts <- clusterApply(cls,ichunks,mtl,m)
24      do.call(sum,counts) / (n*(n-1)/2)
25  }
```

Предположим, этот код хранится в файле SnowMutLinks.R. Давайте для начала разберемся, как его запустить.

16.2.1. Выполнение кода snow

Выполнение приведенного выше кода состоит из следующих шагов:

1. Загрузить код.
2. Загрузить библиотеку snow.
3. Сформировать кластер snow.
4. Задать матрицу смежности.
5. Запустить код для матрицы и сформированного вами кластера.

Предположим, вы работаете на двухъядерной машине. R нужно передать следующие команды:

```
> source("SnowMutLinks.R")
> library(snow)
> cl <- makeCluster(type="SOCK",c("localhost","localhost"))
> testm <- matrix(sample(0:1,16,replace=T),nrow=4)
> mutlinks(cl,testm)
[1] 0.6666667
```

Здесь мы приказываем snow запустить на машине два новых процесса R (localhost — стандартное сетевое имя для локальной машины). Далее эти

процессы будут называться *рабочими потоками* (workers), а исходный процесс R — тот, в котором вводятся предыдущие команды, — *управляющим потоком* (manager). Итак, на данный момент на машине выполняются три экземпляра R (например, если вы работаете в среде Linux, их можно просмотреть командой `ps`). Рабочие потоки образуют *кластер* (cluster) в терминологии `snow`, которому было присвоено имя `cl`.

Пакет `snow` использует парадигму, которая в мире параллельной обработки называется *парадигмой фрагментации/дефрагментации*. Она работает следующим образом:

1. Управляющий поток разбивает данные на блоки и передает их рабочим потокам (фаза фрагментации).
2. Рабочие потоки обрабатывают свои фрагменты.
3. Управляющий поток собирает результаты у рабочих потоков (фаза дефрагментации) и объединяет их способом, подходящим для конкретной ситуации.

Также следует указать, что взаимодействие между управляющим потоком и рабочими потоками будет происходить через сетевые сокеты (см. главу 10).

Тестовая матрица для тестирования кода:

```
> testm
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    1
[2,]    0    0    0    0
[3,]    1    0    1    1
[4,]    0    1    0    1
```

Строка 1 имеет нуль исходящих связей, общих со строкой 2, две общие записи со строкой 3 и одну общую запись со строкой 4. Строка 2 имеет нуль исходящих связей, общих с остальными, но у строки 3 есть одна общая связь со строкой 4. Итого получаем четыре взаимных исходящих связи из $4 \times 3/2 = 6$ пар, а следовательно, среднее значение 4,6 = 0,6666667, как было приведено ранее.

Кластеры могут иметь произвольный размер, лишь бы вы располагали необходимыми машинами. Например, в моем отделе у меня три машины с сетевыми именами `pc28`, `pc29` и `pc30`. Каждая машина является двухъядерной, поэтому кластер для шести рабочих может быть создан так:

```
> cl6 <- makeCluster(type="SOCK",c("pc28","pc28","pc29","pc29",
"pc30","pc30"))
```

16.2.2. Анализ кода snow

А теперь разберемся, как работает функция `mutlinks()`. Сначала определяется количество строк в матрице `m` (строка 17) и количество рабочих потоков в кластере (строка 18).

Затем нужно определить, какой рабочий поток будут обрабатывать те или иные значения `i` цикла `for` из схемы кода, представленной выше в разделе 16.1. Для этой цели хорошо подходит функция R `split()`. Например, для матрицы из четырех строк и рабочего кластера из двух строк этот вызов выдает следующий результат:

```
> split(1:4,1:2)
$`1`
[1]13

$`2`
[1] 2 4
```

Возвращается список R, первым элементом которого является вектор (1, 3), а вторым — вектор (2, 4). Тем самым мы указываем, что один процесс R будет работать с нечетными значениями `i`, а другой — с четными значениями, как обсуждалось выше. Мы подавляем предупреждения, которые выдает `split()` («Длина данных не кратна переменной split»), вызовом `options()`.

Настоящая работа выполняется в строке 23, где вызывается функция `clusterApply()` пакета `snow`. Эта функция инициирует вызов одноименной функции (здесь `mtl()`), с аргументами, специфичными для каждого рабочего потока, и необязательными аргументами, общими для всех. Итак, вот что делает вызов в строке 23:

1. Рабочий поток 1 получает приказание вызвать функцию `mtl()` с аргументами `ichunks[[1]]` и `m`.
2. Рабочий поток 2 будет вызывать `mtl()` с аргументами `ichunks[[2]]` и `m` и т. д. для всех рабочих потоков.
3. Каждый рабочий поток выполняет порученную задачу и возвращает результат управляющему потоку.
4. Управляющий поток собирает все результаты в список R, который мы присваиваем переменной `counts`.

В этой точке достаточно всего лишь просуммировать все элементы `counts`. Вообще-то я не должен говорить «всего лишь», потому что в строке 24 необходимо исправить маленькую проблему. Функция R `sum()` может использовать несколько векторных аргументов:

```
> sum(1:2,c(4,10))
[1] 17
```

Но здесь counts является списком R, а не (числовым) вектором. По этой причине мы используем do.call() для извлечения векторов из counts, а затем вызываем для них sum().

Обратите внимание на строки 9 и 10. Как вы знаете, в R для улучшения быстродействия там, где это возможно, вычисления следует векторизировать. Преобразуя данные, мы заменяем циклы for j и for k в схеме из раздела 16.1 одним векторным выражением.

16.2.3. Какого ускорения можно добиться?

Я опробовал этот код на матрице 1000 × 1000 с именем m1000. Сначала он был выполнен в кластере с четырьмя рабочими потоками, а затем в кластере с 12 рабочими потоками. Теоретически вроде бы ускорение должно быть 4- и 12-кратным соответственно. Но фактически затраченное время составило 6,2 секунды и 5,0 секунды. Сравните эти показатели с 16,9 секунды выполнения в непараллельной форме (с вызовом mtl(1:1000,m1000)). Таким образом, было достигнуто ускорение 2,7 вместо теоретических 4,0 для кластера из четырех рабочих потоков и 3,4 вместо 12,0 в системе с 12 узлами. (Также следует учитывать, что возможны некоторые расхождения между запусками.) Что пошло не так?

Почти в любом приложении с параллельной обработкой существуют *непроизводительные затраты*, или «потери» времени на операции, не связанные с непосредственными вычислениями. В этом примере имеются потери в форме времени, необходимого для передачи матрицы от управляющего потока рабочим потокам. Также некоторые потери связаны с отправкой самой функции mtl() рабочим потокам. А когда рабочие потоки завершают свои задачи, возвращение результатов управляющему потоку также связано с некоторыми потерями. Эта тема обсуждается более подробно при рассмотрении общих аспектов быстродействия в разделе 16.4.1.

16.2.4. Расширенный пример: кластеризация методом k-средних

Чтобы вы лучше поняли возможности snow, мы рассмотрим еще один пример, в котором используется кластеризация методом k-средних (KMC).

КМС — метод исследовательского анализа данных. При просмотре диаграмм разброса данных может возникнуть впечатление, что наблюдения делятся на группы; метод КМС предназначен для поиска таких групп. Результатом является набор центроидов групп.

Ниже приведена схема алгоритма:

```
for iter = 1,2,...,niters
   Присвоить элементам векторов и счетчикам 0
   for i = 1,...,nrow(m)
      Присвоить j = индекс центра ближайшей группы m[i,]
      добавить m[i,] к сумме вектора для группы j, v[j]
      Увеличить на 1 счетчик группы j, c[j]
   for j = 1,...,ngrps
      Назначить новый центр группы j = v[j] / c[j]
```

Мы задаем `niters` итераций с `initcenters` как исходными предположениями относительно центров групп. Данные хранятся в матрице `m`, количество групп равно `ngrps`.

В следующем листинге приведен код `snow` для параллельного вычисления КМС.

```
# Версия кластеризации методом k-средних с использованием snow

library(snow)

# Возвращает расстояния от x до каждого вектора в y;
# здесь x - один вектор, а y - набор векторов;
# расстояние между двумя точками определяется как сумма абсолютных значений
# их покомпонентных разностей; например, расстояние между (5,4.2) и
# (3,5.6) равно 2+1.4=3.4
dst <- function(x,y) {
   tmpmat <- matrix(abs(x-y),byrow=T,ncol=length(x)) # С переработкой
   rowSums(tmpmat)
}

# Проверяет матрицу mchunk рабочего потока по currctrs, текущим центрам
# групп и возвращает матрицу; строка j матрицы состоит из векторной
# суммы точек mchunk, ближайших к j-му текущему центру, и количества
# таких точек.
findnewgrps <- function(currctrs) {
   ngrps <- nrow(currctrs)
   spacedim <- ncol(currctrs) # Определить размерность
   # Создать возвращаемую матрицу
   sumcounts <- matrix(rep(0,ngrps*(spacedim+1)),nrow=ngrps)
   for (i in 1:nrow(mchunk)) {
      dsts <- dst(mchunk[i,],t(currctrs))
```

```
26        j <- which.min(dsts)
27        sumcounts[j,] <- sumcounts[j,] + c(mchunk[i,],1)
28     }
29     sumcounts
30  }
31
32  parkm <- function(cls,m,niters,initcenters) {
33     n <- nrow(m)
34     spacedim <- ncol(m) # Определить размерность
35     # Определить, какому рабочему потоку достается блок строк m
36     options(warn=-1)
37     ichunks <- split(1:n,1:length(cls))
38     options(warn=0)
39     # Сформировать блоки строк
40     mchunks <- lapply(ichunks,function(ichunk) m[ichunk,])
41     mcf <- function(mchunk) mchunk <<- mchunk
42     # Передать блоки строк рабочим потокам; каждый блок будет
43     # представлен в рабочем потоке глобальной переменной mchunk
44     invisible(clusterApply(cls,mchunks,mcf))
45     # Передать dst() рабочим потокам
46     clusterExport(cls,"dst")
47     # Запустить итерации
48     centers <- initcenters
49     for (i in 1:niters) {
50        sumcounts <- clusterCall(cls,findnewgrps,centers)
51        tmp <- Reduce("+",sumcounts)
52        centers <- tmp[,1:spacedim] / tmp[,spacedim+1]
53        # Если группа пуста, заполнить ее центр нулями
54        centers[is.nan(centers)] <- 0
55     }
56     centers
57  }
```

Приведенный код имеет много общего с более ранним примером с взаимными исходящими связями. Тем не менее в нем встречается пара новых вызовов `snow` и другая форма использования старого вызова.

Начнем со строк 39–44. Так как матрица `m` не изменяется между итерациями, ее наверняка не следует повторно отправлять рабочим потокам, чтобы не усугублять проблему непроизводительных затрат. Таким образом, сначала нужно отправить каждому рабочему потоку выделенный ему блок `m` — всего один раз. Это делается в строке 44 при помощи функции `clusterApply()` пакета `snow`; эта функция уже использовалась ранее, но здесь нужно проявить творческий подход. В строке 41 определяется функция `mcf()`, которая при выполнении в рабочем потоке получает блок от управляющего потока и сохраняет его в глобальной переменной `mchunk` данного рабочего потока.

В строке 46 используется новая функция пакета snow — clusterExport(). Она предназначена для копирования глобальных переменных управляющего потока в рабочих потоках. Переменной, о которой идет речь, в данном случае является функция dst(). Почему ее приходится отправлять отдельно? Вызов в строке 50 отправляет функцию findnewgrps() рабочим потокам, но хотя эта функция вызывает dst(), snow не знает, что последнюю тоже нужно отправить. Поэтому мы отправляем ее самостоятельно.

В самой строке 50 используется новая функция snow с именем clusterCall(). Она приказывает каждому рабочему потоку вызвать findnewgrps() с аргументом centers.

Вспомните, что каждый рабочий поток использует свой блок матрицы, поэтому этот вызов будет работать с разными данными в разных рабочих потоках. При этом снова проявляется неоднозначность, связанная с использованием глобальных переменных (раздел 7.8.4). Некоторых разработчиков может беспокоить использование скрытого аргумента в findnewgrps(). С другой стороны, как упоминалось ранее, использование mchunk как аргумента будет означать его многократную отправку рабочим потокам с ущербом для быстродействия.

Наконец, обратимся к строке 51. Функция clusterApply() пакета snow всегда возвращает список R. В данном случае возвращаемое значение хранится в списке sumcounts, каждый элемент которого является матрицей. Матрицы необходимо просуммировать, чтобы получить матрицу сумм. Функция R sum() для этого не подойдет, так как она суммирует все элементы матриц в одно число. В данном случае нам нужна операция сложения матриц.

Вызов функции R Reduce() выполнит матричное сложение. Вспомните, что любая арифметическая операция в R реализована в виде функции; в данном случае она реализована как функция "+". Вызов Reduce() последовательно применяет "+" к элементам списка sumcounts. Конечно, то же самое можно было сделать, просто написав цикл, но использование Reduce() может обеспечить небольшой выигрыш по быстродействию.

16.3. Переход на уровень C

Как вы уже видели, использование параллельной обработки в R может значительно ускорить ваш код. Такое решение позволяет сохранить удобство и выразительность R, при этом успешно справляясь с большим временем выполнения в больших приложениях. Если параллельные вычисления в R обеспечивают достаточно хорошее быстродействие — все хорошо.

Тем не менее параллельный R — все еще R, а значит, ему присущи проблемы быстродействия, представленные в главе 14. Вспомните, что одно из решений предлагало написать часть кода, критичную по времени, на языке C, а затем вызвать этот код из основной программы R. (Все упоминания C относятся к C и C++.) Исследуем эту возможность с точки зрения параллельной обработки. Вместо того чтобы писать параллельный код R, мы напишем обычный код R, который вызывает параллельный код C. (Этот раздел потребует знания языка C.)

16.3.1. Использование многоядерных машин

Код C, описанный в этом разделе, работает только в многоядерных системах, поэтому сначала необходимо обсудить природу таких систем.

Вероятно, вы знакомы с двухъядерными машинами. Любой компьютер включает центральный процессор — компонент, который непосредственно выполняет ваши программы. По сути двухъядерная машина оснащается двумя процессорами, четырехъядерная — четырьмя и т. д. Наличие нескольких ядер позволяет выполнять параллельные вычисления!

Параллельные вычисления осуществляются в *программных потоках* (threads) — аналогах рабочих потоков пакета `snow`. В приложениях, требующих интенсивных вычислений, обычно создаются программные потоки по количеству ядер (например, два потока на двухъядерной машине). В идеале такие потоки выполняются одновременно, хотя на практике возникают проблемы с непроизводительными операциями, как будет показано при рассмотрении общих проблем быстродействия в разделе 16.4.1.

Машина, оснащенная несколькими ядрами, по своей структуре близка к системам с общей памятью. Все ядра работают с одной оперативной памятью. Совместное использование памяти упрощает программирование взаимодействий между ядрами. Если поток записывает данные по адресу памяти, то изменения будут видны всем остальным потокам, а программисту не придется писать для этого специальный код.

16.3.2. Расширенный пример: задача взаимных исходящих связей в OpenMP

OpenMP — очень популярный пакет для программирования на многоядерных машинах. Чтобы увидеть, как он работает, вернемся к примеру с взаимными исходящими связями — на этот раз в коде OpenMP, вызываемом из R:

```
#include <omp.h>
#include <R.h>

int tot; // Общее количество совпадений по всем потокам

// Обработать пары строк (i,i+1), (i,i+2), ...
int procpairs(int i, int *m, int n)
{  int j,k,sum=0;
   for (j = i+1; j < n; j++) {
     for(k = 0; k < n; k++)
        // Найти m[i][k]*m[j][k]. Не забудьте, что в R данные
        sum += m[n*k+i] * m[n*k+j]; // хранятся по столбцам.
   }
   return sum;
}

void mutlinks(int *m, int *n, double *mlmean)
{ int nval = *n;
   tot=0;
   #pragma omp parallel
   { int i,mysum=0,
         me = omp_get_thread_num(),
         nth = omp_get_num_threads();
      // При проверке всех пар (i,j) работа разбивается по i;
      // этот поток me обрабатывает все i, равные me mod nth
      for (i = me; i < nval; i += nth) {
           mysum += procpairs(i,m,nval);
      }
      #pragma omp atomic
      tot += mysum;
   }
   int divisor = nval * (nval-1) / 2;
   *mlmean = ((float) tot)/divisor;
}
```

16.3.3. Выполнение кода OpenMP

И снова компиляция выполняется по схеме из главы 15. Впрочем, на этот раз нужно подключить библиотеку OpenMP при помощи ключей `-fopenmp` и `-lgomp`. Допустим, исходный код хранится в файле romp.c. Тогда для выполнения кода будут использоваться следующие команды:

```
gcc -std=gnu99 -fopenmp -I/usr/share/R/include -fpic -g -O2 -c
     romp.c -o romp.o
gcc -std=gnu99 -shared -o romp.so romp.o -L/usr/lib/R/lib -lR -lgomp
```

Тестовая программа R:

```
> dyn.load("romp.so")
> Sys.setenv(OMP_NUM_THREADS=4)
> n <- 1000
> m <- matrix(sample(0:1,n^2,replace=T),nrow=n)
> system.time(z <- .C("mutlinks",as.integer(m),as.integer(n),result=double(1)))
   user  system elapsed
  0.830   0.000   0.218
> z$result
[1] 249.9471
```

Количество потоков для OpenMP чаще всего задается переменной среды операционной системы `OMP_NUM_THREADS`. R позволяет задавать переменные среды функцией `Sys.setenv()`. Здесь количество потоков задается равным 4, потому что код выполняется на четырехъядерной машине.

Обратите внимание на время выполнения — всего 0,2 секунды! Сравните с 5,0 секунды для решения с системой `snow` из 12 узлов. Некоторым читателям это может показаться удивительным, так как код `snow`-версии был векторизован в значительной степени, как упоминалось ранее. Векторизация — это хорошо, но я еще раз подчеркну: в R много скрытых источников непроизводительных затрат, поэтому C может справиться еще лучше.

ПРИМЕЧАНИЕ

Я опробовал новую функцию R компиляции в байт-код cmpfun(), но функция mtl() стала работать медленнее.

Итак, если вы напишете часть своего кода на параллельном C, это может привести к радикальному ускорению его выполнения.

16.3.4. Анализ кода OpenMP

Код OpenMP написан на C с добавлением *директив* (pragma), приказывающих компилятору вставить библиотечный код для выполнения операций OpenMP. Например, взгляните на строку 20. Когда выполнение достигнет этой точки, потоки будут активизированы. Каждый поток выполняет следующий блок (строки с 21 по 31) параллельно с другими.

Ключевым моментом здесь является видимость переменных. Все переменные в блоке, начинающемся со строки 21, локальны для своих конкретных потоков.

Например, мы присвоили переменной в строке 21 имя `mysum`, потому что каждый поток поддерживает собственную сумму. С другой стороны, глобальная переменная `tot` в строке 4 является общей для всех потоков. Каждый поток вносит свой вклад в накапливаемую сумму в строке 30.

Но даже переменная `nval` в строке 18 остается общей для всех потоков (во время выполнения `mutlinks()`), поскольку она объявляется за пределами блока, начинающегося со строки 21. Итак, хотя она и является локальной переменной в контексте областей видимости C, эта переменная глобальна для всех потоков. Собственно, `tot` можно было бы объявить в этой строке. Переменная должна быть общей для всех потоков, но поскольку она не используется за пределами `mutlinks()`, ее можно было бы объявить в строке 18.

Строка 29 также содержит еще одну директиву `atomic`. Она относится не ко всему блоку, а только к одной строке, следующей за ней (строка 30 в данном случае). Директива `atomic` предотвращает явление, называемое *состоянием гонки* (race condition) в области параллельной обработки. Этот термин описывает ситуацию, в которой два потока одновременно обновляют переменную, что может привести к некорректным результатам. Директива `atomic` гарантирует, что строка 30 будет выполняться только одним потоком в любой момент времени. Отсюда следует, что в этой секции кода наша параллельная программа временно становится последовательной, что может стать потенциальным источником замедления.

Какую роль во всем этом играет управляющий поток? Именно он является исходным программным потоком, и именно он выполняет строки 18 и 19, а также `.C()` — функцию R, которая вызывает `mutlinks()`. Когда в строке 21 активизируются рабочие потоки, управляющий поток временно «засыпает». Рабочие потоки «засыпают» при достижении строки 31, и в этот момент управляющий поток возобновляет выполнение. Поскольку управляющий поток неактивен во время выполнения рабочих потоков, количество рабочих потоков должно соответствовать количеству ядер на компьютере.

Функция `procpairs()` проста, но в ней стоит обратить внимание на обращения к матрице `m`. Вспомните, что говорилось в главе 15 при обсуждении взаимодействия R с C: в этих двух языках матрицы хранятся по-разному. В R используется порядок хранения по столбцам, а в C — по строкам. Необходимо помнить об этом различии. Кроме того, матрица `m` рассматривалась как одномерный массив, как это обычно бывает в параллельном коде C. Другими словами, если переменная `n`, скажем, равна 4, то `m` будет рассматриваться как вектор из 16 элементов. Вследствие порядка хранения по столбцам в R вектор сначала будет содержать

первые четыре элемента столбца 1, затем четыре элемента столбца 2 и т. д. Дело дополнительно усложняется тем, что индексирование массивов в C начинается с 0, а не с 1, как в R.

Объединяя все сказанное, мы получаем операцию умножения в строке 12. Множителями здесь являются элементы `(k,i)` и `(k,j)` версии `m` в коде C, которые соответствуют элементам `(i+1,k+1)` и `(j+1,k+1)` в коде R.

16.3.5. Другие директивы OpenMP

OpenMP включает обширный набор возможных операций — слишком обширный для того, чтобы перечислить их здесь. В этом разделе приведена сводка некоторых директив OpenMP, которые мне кажутся особенно полезными.

16.3.5.1. Директива omp barrier

«Барьером» (barrier) в области параллельной обработки называется строка кода, в которой происходит «встреча» потоков. Синтаксис директивы `omp barrier` прост:

```
#pragma omp barrier
```

Когда поток достигает барьера, его выполнение приостанавливается до тех пор, пока все остальные потоки не достигнут этой строки. Это особенно полезно для итеративных алгоритмов; потоки ожидают у барьера в конце каждой итерации.

Учтите, что в дополнение к явному определению барьеров некоторые директивы, такие как `single` и `parallel`, устанавливают неявный барьер после своих блоков. Также неявный барьер располагается сразу же за строкой 31 в предыдущем листинге, поэтому управляющий поток продолжает ожидание до того момента, когда все рабочие потоки завершат работу.

16.3.5.2. Директива omp critical

Блок, следующий за этой директивой, является *критической секцией* (critical section); это означает, что в любой момент времени она может выполняться только одним потоком. Директива `omp critical` фактически служит той же цели, что и директива `atomic`, рассмотренная ранее (хотя последняя ограничивается одной командой).

ПРИМЕЧАНИЕ

Проектировщики OpenMP определили специальную директиву для ситуации с одной командой в надежде, что компилятор сможет преобразовать ее в быструю машинную команду.

Синтаксис директивы `omp critical`:

```
1 #pragma omp critical
2 {
3    // Здесь размещается одна или несколько команд
4 }
```

16.3.5.3. Директива omp single

Блок, следующий за директивой, может выполняться только в одном из потоков. Синтаксис директивы `omp single`:

```
1 #pragma omp single
2 {
3    // Здесь размещается одна или несколько команд
4 }
```

Например, это может быть полезно для инициализации переменных сумм, общих для потоков. Как упоминалось ранее, после блока автоматически устанавливается барьер. И это вполне логично: если один поток инициализирует сумму, другие потоки не должны пытаться использовать эту переменную для продолжения выполнения, пока сумма не будет правильно инициализирована.

OpenMP более подробно рассматривается в моем свободно распространяемом учебнике по параллельной обработке (http://heather.cs.ucdavis.edu/parprocbook).

16.3.6. Программирование графических процессоров

Другую разновидность параллельного оборудования с общей памятью представляют графические процессоры (GPU, Graphics Processing Unit). Если на вашей машине установлена современная видеокарта, возможно, вы даже не осознаете, что она также является очень мощным вычислительным устройством, — настолько мощным, что лозунг «Суперкомпьютер на вашем рабочем столе!» часто применяется для PC с современными графическими процессорами.

Как и в случае с OpenMP, идея заключается в том, чтобы вместо параллельного кода R написать код R, взаимодействующий с параллельным кодом C. (По ана-

логии с примером для OpenMP, C здесь означает слегка усовершенствованную версию языка C.) Технические подробности довольно сложны, так что я не буду приводить примеры кода, но общий обзор платформы все же будет полезен.

Как упоминалось ранее, графические процессоры следуют модели общей памяти/программных потоков, но во много большем масштабе. Они оснащаются десятками и даже сотнями ядер (в зависимости от того, как определять *ядро*). Одно серьезное различие заключается в том, что несколько потоков могут вместе выполняться в одном блоке, что может способствовать повышению эффективности.

Программы, работающие с графическими процессорами, начинают свое выполнение на центральном процессоре машины, который называется *хостом* (host). Затем они запускают код на графическом процессоре, или *устройстве* (device). Это означает, что данные должны быть переданы с хоста на устройство, и после того как устройство завершит вычисления, результаты должны быть переданы обратно хосту.

На момент написания книги GPU-программирование еще не получило широкого распространения среди пользователей R. Пожалуй, чаще всего оно используется при помощи пакета CRAN `gputools`, содержащего ряд операций матричной алгебры и статистики, доступных для вызова из R. Для примера возьмем обращение матриц. R предоставляет для этой цели функцию `solve()`, но в `gputools` имеется параллельная альтернатива с именем `gpuSolve()`.

В качестве дополнительной информации о программировании графических процессоров я снова порекомендую свой учебник по параллельному программированию (http://heather.cs.ucdavis.edu/parprocbook).

16.4. Общие факторы быстродействия

В этом разделе обсуждаются вопросы, которые могут быть полезны при параллелизации приложений R. Сначала я немного расскажу об основных источниках непроизводительных затрат, а затем мы обсудим пару алгоритмических аспектов.

16.4.1. Источники непроизводительных затрат

Успешное параллельное программирование требует хотя бы приблизительного представления о физических причинах непроизводительных затрат. Рассмо-

трим их в контексте двух основных платформ: общей памяти и компьютерной сети.

16.4.1.1. Машины с общей памятью

Как упоминалось ранее, наличие общей памяти на многоядерных машинах упрощает программирование. Тем не менее она также создает непроизводительные затраты, так как при попытке двух ядер одновременно обратиться к памяти произойдет конфликт. Это означает, что одному из них придется ожидать, а это снижает эффективность. Как правило, такие потери измеряются сотнями наносекунд (миллиардных долей секунды). На первый взгляд кажется, что это не много, но процессор работает на субнаносекундных скоростях, так что доступ к памяти часто становится узким местом приложений.

У каждого ядра также может быть свой *кэш*, в котором хранится локальная копия некоторой части общей памяти. Кэширование должно сократить конкуренцию за память между ядрами, но оно вводит собственные непроизводительные затраты из-за времени, потраченного на согласование содержимого кэшей.

Вспомните, что графические процессоры сами являются своего рода многоядерными машинами. Соответственно, они страдают от перечисленных, а также ряда других проблем. Прежде всего для графических процессоров характерна достаточно высокая *задержка* (интервал времени перед получением графическим процессором первого бита из памяти после запроса).

Также присутствуют затраты, связанные с передачей данных между хостом и устройством. Задержка составляет микросекунды (миллионные доли секунд) — целая вечность по сравнению с наносекундным масштабом центрального и графического процессора.

Графические процессоры предоставляют огромный потенциал повышения быстродействия для некоторых классов приложений, но непроизводительные затраты могут стать серьезной проблемой. Авторы `gputools` замечают, что ускорение матричных операций начинает наблюдаться только при размерах матриц от 1000 × 1000. Я написал GPU-версию приложения взаимных исходящих связей, время выполнения которой составило 3,0 секунды — около половины времени `snow`-версии, но все равно намного медленнее реализации для OpenMP.

Еще раз подчеркну: у этих проблем существуют свои решения, но они требуют очень тщательного, творческого программирования и досконального знания физической структуры графического процессора.

16.4.1.2. Сетевые компьютерные системы

Как упоминалось ранее, параллельные вычисления также могут быть реализованы посредством сетевых компьютерных систем. Система также содержит несколько процессоров, но в данном случае они находятся на совершенно разных компьютерах, каждый из которых обладает собственной памятью. Передача данных по сети также создает непроизводительные затраты, а задержка тоже измеряется микросекундами. Таким образом, обращение даже к небольшому объему данных по сети сопряжено с серьезной задержкой.

Также обратите внимание на то, что `snow` вводит дополнительную задержку из-за преобразования числовых объектов (векторов, матриц и т. д.) в символьную форму перед отправкой их, допустим, от управляющего потока рабочим потокам. Потери обусловлены даже не столько временем преобразования (из числовой формы в символьную и обратного перехода к числовой форме на стороне получателя), сколько символьным представлением, которое существенно увеличивает длину сообщения, а следовательно, повышает время передачи по сети.

Системы с общей памятью могут объединяться в сеть, как, собственно, было сделано в предыдущем примере. В нем использовалась гибридная конфигурация, в которой кластеры `snow` формировались из нескольких двухъядерных компьютеров, объединенных в сеть.

16.4.2. Тривиальная параллелизуемость

> Бедность не порок, но и деньги в доме не лишнее.
>
> — *Тевье. Скрипач на крыше*

> Человек — единственное животное, которое краснеет или при определенных обстоятельствах должно краснеть.
>
> — *Марк Твен*

Термин «чрезвычайная параллельность» часто встречается при обсуждении параллельного программирования на R (и в области параллельной обработки вообще). Имеется в виду, что задачи параллелизуются настолько легко, что это не требует никаких интеллектуальных усилий, все слишком просто.

Оба примера, рассмотренные в этой главе, можно считать тривиально параллелизуемыми. Параллелизация цикла `for i` для задачи взаимных исходящих связей в разделе 16.1 была достаточно очевидной. Разбиение работы в примере KMC в разделе 16.2.4 тоже было естественным и простым.

С другой стороны, многие параллельные алгоритмы сортировки требуют значительного взаимодействия. Возьмем алгоритм сортировки слиянием — стандартный метод сортировки числовых данных. Сортируемый вектор разбивается на две (и более) независимые части — условно левую и правую половины, которые затем параллельно сортируются двумя процессами. До этого момента задача относится к категории тривиального параллелизма, по крайней мере после деления вектора надвое. Но затем две отсортированные половины должны быть объединены для формирования отсортированной версии исходного вектора, и этот процесс уже не является тривиально параллельным. Он может быть параллелизован, но это делается сложнее.

Конечно, перефразируя слова Тевье, тривиальный параллелизм — не порок! Может, это и не заслуга, но и не лишнее. По крайней мере здесь можно только порадоваться, потому что такие решения легко программируются. Что еще важнее, задачи из категории тривиального параллелизма обычно имеют низкие коммуникационные потери, что очень важно для высокого быстродействия, как упоминалось выше. Собственно, когда многие специалисты говорят о тривиально параллелизируемых приложениях, они имеют в виду именно низкие непроизводительные затраты.

А как насчет нетривиально параллельных приложений? К сожалению, параллельный код R попросту не подходит для многих из них по очень простой причине: природе R как языка функционального программирования. Как обсуждалось в разделе 14.3, команда следующего вида:

```
x[3] <- 8
```

обманчиво проста, потому что она может потребовать перезаписи всего вектора `x`. На самом деле она только усугубляет проблемы с трафиком. Соответственно, если ваше приложение не является тривиально параллельным, вашей лучшей стратегией будет написание вычислительно-интенсивных частей своего кода на C, скажем, с использованием OpenMP или программирования графического процессора.

Также следует учитывать, что даже тривиальная параллелизуемость не означает эффективности алгоритма. Некоторые алгоритмы такого рода могут иметь значительный коммуникационный трафик, что вредит быстродействию.

Возьмем задачу кластеризации методом k-средних, запущенную с использованием snow. Предположим, вы хотите создать достаточное количество рабочих потоков, чтобы каждый рабочий поток выполнял относительно небольшую работу. В этом случае взаимодействие с управляющим потоком после каждой итерации окажет значительное влияние на время выполнения. В этой ситуации можно сказать, что детализация задачи завышена и стоит сократить количество рабочих потоков. Тогда объем работы на один поток увеличивается.

16.4.3. Статическое и динамическое распределение задач

Взгляните на цикл, начинающийся со строки 26 нашего примера OpenMP, — повторю его для удобства:

```
for (i = me; i < nval; i += nth) {
   mysum += procpairs(i,m,nval);
}
```

Переменная me — номер потока, поэтому в результате выполнения этого кода различные потоки будут работать над неперекрывающимися наборами значений i. Эти значения не должны перекрываться во избежание дублирования работы и ошибочного подсчета связей, так чтобы этот код работал нормально. Но суть в том, что здесь задачи, которые будут обрабатываться каждым потоком, фактически назначаются заранее. Такой механизм называется *статическим* назначением.

Альтернативное решение заключается в пересмотре цикла и приведении его к следующему виду:

```
int nexti = 0; // Глобальная переменная
...
for ( ; myi < n; ) { // Переработанный цикл "for"
   #pragma omp critical
   {
      nexti += 1;
      myi = nexti;
   }
   if (myi < n) {
      mysum += procpairs(myi,m,nval);
      ...
   }
}
...
```

Здесь назначение задач осуществляется *динамически*, то есть программа не определяет заранее, какой поток будет обрабатывать те или иные значения `i`. Задачи назначаются в процессе выполнения. На первый взгляд динамическое назначение вроде бы обладает потенциалом для повышения быстродействия. Предположим, например, что при статическом назначении один поток быстро завершил обработку своего последнего значения `i`, тогда как у другого потока остались необработанными еще два значения `i`. Следовательно, программа завершит работу позднее, чем могла бы. В терминологии параллельной обработки возникает проблема *распределения нагрузки*. При динамическом назначении поток, который завершает работу при двух оставшихся значениях `i`, мог бы взять одно из этих значений для себя. Это обеспечило бы улучшенное распределение нагрузки и (теоретически) сократило бы общее время выполнения.

Однако не стоит торопиться с выводами. Как обычно, приходится считаться с проблемой непроизводительных затрат. Вспомните, что директива `critical`, используемая в динамической версии кода, по сути, временно переводит программу из параллельного режима в последовательный, вызывая ее замедление. Кроме того, по причинам, слишком техническим для того, чтобы их здесь обсуждать, эти директивы могут создавать существенные дополнительные затраты при операциях с кэшем. Таким образом, в конечном итоге динамический код может работать намного медленнее своей статической версии.

Были разработаны различные решения этой проблемы, в числе которых конструкция OpenMP с именем `guided`. Но вместо того чтобы описывать их здесь, я хочу подчеркнуть другое: они не являются необходимыми. В большинстве ситуаций вполне достаточно статического назначения. Почему?

Вспомните, что среднеквадратическое отклонение суммы независимых случайных переменных с одинаковым распределением, разделенное на среднее значение этой суммы, стремится к 0, когда количество слагаемых стремится к бесконечности. Другими словами, суммы приблизительно постоянны. Это имеет прямые следствия для наших проблем с распределением нагрузки: так как общее рабочее время для потока при статическом присваивании представляет собой сумму времен отдельных задач, общее рабочее время будет приблизительно постоянным; вариативность между потоками очень невелика. Таким образом, все они завершатся приблизительно в одно и то же время, и беспокоиться о дисбалансе нагрузки не нужно. Динамическое планирование становится излишним.

Это обоснование зависит от статистического предположения, но на практике оно достаточно хорошо выполняется для результата: в отношении однородно-

сти суммарного времени работы между потоками статическое планирование справляется с задачей так же хорошо, как и динамическое. А поскольку у статического планирования нет проблем непроизводительных затрат, присущих динамическому, в большинстве случаев статический механизм обеспечит лучшее быстродействие.

У этой проблемы есть еще один аспект. Чтобы продемонстрировать его суть, вернемся к примеру с взаимными исходящими связями. Давайте рассмотрим структуру алгоритма:

```
1 sum=0
2 for i = 0...n-1
3    for j = i+1...n-1
4       for k = 0...n-1 sum = sum + a[i][k]*a[j][k]
5 mean = sum / (n*(n-1)/2)
```

Допустим, значение `n` равно 10 000, используются четыре потока, и вы рассматриваете возможность разбиения цикла `for i`. Первая наивная попытка — поручить потоку 0 обрабатывать значения `i` с 0 до 2499, потоку 1 — с 2500 до 4999 и т. д.

Тем не менее это приведет к серьезному дисбалансу нагрузки, так как поток, обрабатывающий заданное значение `i`, выполняет объем работы, пропорциональный `n-i`. Именно по этой причине значения `i` в коде чередовались: поток 0 обрабатывал значения `i` 0, 4, 8..., поток 1 работал над 1, 5, 9, ... и т. д., что обеспечивало хорошее распределение нагрузки.

Дело в том, что статическое назначение может потребовать несколько большего планирования. Одно из возможных общих решений — случайное закрепление задач (значений `i` в нашем конкретном случае) за потоками (причем это также делается заранее, до начала работы). Если обдумать ситуацию заранее, статическое назначение в большинстве случаев будет работать достаточно хорошо.

16.4.4. Программная алхимия: преобразование общих задач в тривиально параллельные

Как упоминалось ранее, трудно обеспечить хорошее быстродействие в нетривиальных параллельных алгоритмах. К счастью, для статистических приложений существует механизм преобразования нетривиально параллельных задач в тривиально параллельные. Ключом здесь становится использование некоторых статистических свойств.

Для демонстрации метода мы снова обратимся к задаче о взаимных исходящих связях. Метод, применяемый с `w` рабочими потоками к матрице связей `m`, выглядит так:

1. Разбить строки `m` на `w` блоков.
2. Поручить каждому рабочему потоку вычислить среднее количество взаимных исходящих связей для пар вершин в его блоке.
3. Усреднить результаты, возвращаемые рабочими потоками.

Можно показать с математических позиций, что для больших задач (единственного класса, для которых вообще имеет смысл параллельное обработка) это блочное решение дает оценки с такой же статистической точностью, как и неблочное. Но при этом непараллельная задача преобразуется не только в параллельную, но и в тривиально параллельную! Рабочие потоки в приведенной выше схеме выполняются полностью независимо друг от друга.

Этот метод не следует путать с обычным блочным подходом к параллельной обработке. В таких случаях — как, скажем, в примере сортировки слиянием на с. 402 — разбиение на блоки является тривиально параллельным, но к объединению результатов это не относится. Напротив, благодаря математической теории объединение результатов здесь состоит из простого усреднения.

Я опробовал этот метод на задаче взаимных исходящих связей в четырехпотоковом кластере `snow`. Время выполнения сократилось до 1,5 секунды. Такой результат намного лучше 16 секунд при последовательном выполнении, он вдвое превосходит ускорение, полученное применением графического процессора, и сравним с временем OpenMP. Теория также показывает, что эти два метода обладают одинаковой статистической точностью. У блочного метода среднее количество взаимных исходящих связей было равно 249,2881 по сравнению с 249,2993 у исходной оценки.

16.5. Отладка параллельного кода R

Пакеты параллельных вычислений R, такие как `Rmpi`, `snow`, `foreach` и т. д., не создают терминального окна для каждого процесса, что усложняет использование отладчика R с рабочими потоками. (Мой пакет `Rdsm`, добавляющий функциональность работы с программными потоками в R, является исключением.)

Что же можно сделать при отладке приложений для таких пакетов? Для примера возьмем `snow`.

Сначала нужно отладить используемую однопоточную функцию, такую как `mtl()` в разделе 16.2. Сгенерируйте некоторые искусственные значения аргументов, а затем используйте обычные отладочные средства R.

Отладки базовой функции может быть достаточно. Однако ошибка может скрываться в самих аргументах или в способе их создания. Тогда ситуация усложняется.

Еще больше проблем возникает с выводом трассировочной информации (например, значений переменных), так как `print()` не работает в рабочих процессах. Функция `message()` может работать в некоторых из этих пакетов; если она не работает, возможно, придется использовать функцию `cat()` для записи файла.

Приложение A
Установка R

В приложении рассматриваются способы установки R в системе. Вы можете легко загрузить и установить предварительно откомпилированные двоичные файлы, использовать менеджер пакетов в системах на базе UNIX и даже построить R из исходного кода, если вы предпочитаете этот вариант.

A.1. Загрузка R из CRAN

R — как в базовой форме, так и в пакетах, написанных пользователями, — доступен в репозитории CRAN (Comprehensive R Archive Network) на домашней странице R (http://www.r-project.org/).

Щелкните на ссылке CRAN и выберите ближайший сайт для загрузки базового пакета, соответствующего вашей операционной системе (ОС). Для большинства пользователей установка R проходит очень просто независимо от платформы. В репозитории CRAN можно найти предварительно откомпилированные двоичные файлы для Windows, Linux и Mac OS X. Просто загрузите подходящий файл и установите R.

A.2. Установка из менеджера пакетов Linux

Вместо установки заранее откомпилированных двоичных файлов, если вы работаете в дистрибутиве Linux с централизованным репозиторием пакетов (скажем, Fedora или Ubuntu), вы сможете установить R из менеджера пакетов ОС. Например, если вы работаете с Fedora, R можно установить следующей командой:

```
$ yum install R
```

Для систем на базе Debian (например, Ubuntu) команда выглядит так:

```
$ sudo apt-get install r-base
```

За дополнительной информацией об установке и удалении пакетов обращайтесь к документации своего дистрибутива.

А.3. Установка из исходного кода

В Linux и на других машинах на базе UNIX (вероятно, включая Mac OS X) вы также можете откомпилировать исходный код R самостоятельно. Просто распакуйте архив с исходным кодом и проведите классический процесс установки из трех команд:

```
$ configure
$ make
$ make install
```

Возможно, вам придется выполнить команду `make install` от имени `root` — это зависит от прав на запись и каталога, в который вы устанавливаете R. Чтобы использовать нестандартный каталог (скажем, /a/b/c), выполните `configure` с параметром `--prefix`:

```
$ configure --prefix=/a/b/c
```

Это может быть удобно, если вы работаете на общей машине и не обладаете правами на запись в стандартные установочные каталоги (например, /usr).

Приложение Б
Установка и использование пакетов

Одна из самых сильных сторон R — тысячи пакетов, написанных пользователями и доступных в репозитории CRAN (Comprehensive R Archive Network) на домашней странице R (http://www.r-project.org/). В большинстве случаев пакеты устанавливаются просто, но у некоторых специализированных пакетов есть нюансы, которые необходимо учитывать.

Это приложение начинается с описания основ работы с пакетами. Затем вы узнаете, как загрузить пакеты с вашего жесткого диска и из интернета.

Б.1. Основы работы с пакетами

R использует пакеты для хранения групп взаимосвязанных фрагментов программного кода. Пакеты, включенные в дистрибутив R, находятся в подкаталогах каталога library в дереве установки R — например, /usr/lib/R/library.

ПРИМЕЧАНИЕ

В сообществе R пакеты часто называют «библиотеками». Некоторые пакеты загружаются автоматически при запуске R (как, например, подкаталог base). Тем не менее для экономии памяти и времени R не загружает автоматически все доступные пакеты.

Чтобы проверить, какие пакеты загружены в настоящий момент, введите следующую команду:

```
> .path.package()
```

Б.2. Загрузка пакета с жесткого диска

Если вам нужен пакет, присутствующий в вашей установке R, но еще не загруженный в память, загрузите его функцией `library()`. Допустим, вы хотите

генерировать многомерные нормальные случайные векторы. Это делает функция `mvrnorm()` из пакета `MASS`. Итак, команда загрузки пакета выглядит так:

```
> library(MASS)
```

Функция `mvrnorm()` готова к использованию — как и ее документация (до загрузки `MASS` при выполнении команды `help(mvrnorm)` вы бы получили сообщение об ошибке).

Б.3. Загрузка пакета из интернета

Возможен и другой вариант: нужный вам пакет отсутствует в установке R. Одно из больших преимуществ программных продуктов с открытым кодом заключается в том, что люди любят делиться плодами своих трудов. Программисты со всего мира пишут собственные специализированные пакеты R, размещая их в репозитории CRAN и в других местах.

ПРИМЕЧАНИЕ

Пользовательские публикации в CRAN проходят процесс отбора и обычно содержат качественный код. Тем не менее они не тестируются настолько тщательно, как сам R.

Б.3.1. Автоматическая установка пакетов

Один из способов установки пакетов основан на использовании функции `install_packages()`. Допустим, вы хотите использовать пакет `mvtnorm`, который вычисляет кумулятивные функции многомерного нормального распределения и другие величины.

Сначала выберите каталог, в котором должен быть установлен пакет (и возможно, другие пакеты в будущем), — например, /a/b/c. Затем в приглашении R введите следующую команду:

```
> install.packages("mvtnorm","/a/b/c/")
```

При выполнении этой команды R автоматически переходит в репозиторий CRAN, загружает пакет, компилирует его и записывает в новый каталог: /a/b/c/mvtnorm.

После того как пакет будет установлен, вы должны сообщить R, где его следует искать. Для этого можно воспользоваться функцией `.libPaths()`:

```
> .libPaths("/a/b/c/")
```

Команда добавляет новый каталог к тем, которые уже использует R. Если этот каталог используется достаточно часто, возможно, этот вызов `.libPaths()` стоит включить в файл запуска `.Rprofile` в домашнем каталоге.

Вызов `.libPaths()` без аргумента выводит список всех каталогов, в которых R в настоящее время будет искать пакет при получении запроса на загрузку.

Б.3.2. Ручная установка пакетов

Иногда пакеты приходится устанавливать «вручную» для внесения изменений, необходимых для работы конкретного пакета R в вашей системе. Следующий пример демонстрирует, как это было сделано для одного конкретного случая; по этой схеме вы сможете действовать в тех ситуациях, в которых обычные способы не работают.

ПРИМЕЧАНИЕ

Ситуации, в которых возникает необходимость в ручной установке пакетов, обычно зависят от операционной системы. Они требуют более серьезных знаний в области компьютеров, чем предполагается в большей части этой книги.

Список рассылки `r-help` — бесценный источник информации для особых случаев. Чтобы обратиться к нему, откройте домашнюю страницу R (http://www.r-project.org/), щелкните на ссылке FAQs, затем на ссылке R FAQ и прокрутите страницу до раздела 2.9 «What mailing lists exist for R?».

Я решил установить пакет `Rmpi` на учебных машинах нашего отдела в каталоге /home/matloff/R. Сначала я попытался воспользоваться вызовом `install.packages()`, но выяснилось, что автоматизированный процесс не может найти библиотеку `MPI` на наших машинах. Проблема заключалась в том, что R ищет эти файлы в каталоге /usr/local/lam, тогда как я знал, что они находятся в каталоге /usr/local/LAM. Поскольку это были не мои личные, а общие машины, я не имел полномочий для переименования. Тогда я загрузил файлы `Rmpi` в упакованной форме Rmpi_0.5-3.tar.gz и распаковал этот файл в свой каталог ~/tmp, получив каталог с именем ~/tmp/Rmpi.

Если бы не проблема с именем, на этой стадии можно было просто выполнить следующую команду в терминальном окне из каталога ~/tmp:

```
R CMD INSTALL -l /home/matloff/R Rmpi
```

Команда устанавливает пакет из каталога ~/tmp/Rmpi и помещает его в каталог /home/matloff/R. Произошло бы то же самое, что и при вызове `install.packages()`.

Но как я уже сказал, мне пришлось решать проблему. В каталоге ~/tmp/Rmpi находится файл configure, поэтому я выполнил следующую команду в командной строке Linux:

```
configure --help
```

Команда сообщила, что местонахождение файлов MPI для configure задается следующим образом:

```
configure --with-mpi=/usr/local/LAM
```

Это относится к прямому запуску программы configure, но я запустил ее через R:

```
R CMD INSTALL -l /home/matloff/R Rmpi --configure-args=--with-mpi=
                /usr/local/LAM
```

Мое решение вроде бы сработало — по крайней мере, пакет был установлен. Однако R также выдал сообщение о проблеме с библиотекой потоков на наших машинах. И действительно, при попытке загрузить Rmpi произошла ошибка времени выполнения — некоторые функции потоков найти не удалось.

Я знал, что с библиотекой было все в порядке, поэтому заглянул в файл configure и закомментировал две строки:

```
# if test $ac_cv_lib_pthread_main = yes; then
     MPI_LIBS="$MPI_LIBS -lpthread"
#fi
```

Иначе говоря, я заставил configure использовать то, в чем был на сто процентов уверен (или почти уверен). Затем я снова выполнил команду R CMD INSTALL, и пакет загрузился без каких-либо проблем.

Б.4. Вывод списка функций в пакете

Функция library() с аргументом help выводит список функций в пакете. Например, для получения информации о пакете mvtnorm введите одну из следующих команд:

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

Норман Мэтлофф

Искусство программирования на R. Погружение в большие данные

Перевел с английского Е. Матвеев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Бульченко*
Художественный редактор	*С. Заматевская*
Корректоры	*С. Беляева, Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 08.02.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 33,540. Тираж 1200. Заказ 1341.

Отпечатано в АО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru
тел: 8(499) 270-73-59

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

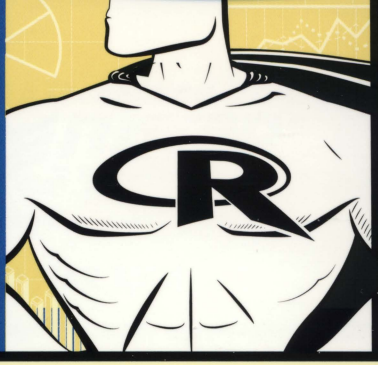
ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ПРИРУЧИТЕ СВОИ ДАННЫЕ



R является самым популярным в мире языком статистических вычислений: археологи используют его, изучая древние цивилизации, фармацевтические компании выясняют, какие лекарства наиболее безопасны и эффективны, а финансисты задействуют его для оценки рисков и удержания позиций на рынке.

«Искусство программирования на R» — это путешествие, в которое вы отправляетесь с опытным гидом, готовым поделиться всей информацией о разработке ПО, от типов и структур данных до таких продвинутых тем, как замыкания, рекурсия и анонимные функции. Вам не понадобятся специальные знания в области статистики, а программистский опыт может варьироваться от начинающего до профессионала. Вы познакомитесь с функциональным и объектно-ориентированным программированием, математическим моделированием и преобразованием сложных данных в простые и удобные форматы.

ИЗ ЭТОЙ КНИГИ ВЫ УЗНАЕТЕ, КАК:

- создавать визуализации сложных наборов данных и функций;
- эффективно использовать параллелизацию и векторизацию;
- взаимодействовать с C/C++ и Python, повышая скорость или функциональность ваших решений;
- искать подходящие пакеты для анализа текста, работы с изображениями и пр.;
- избегать раздражающих ошибок, используя продвинутые возможности отладки.

Проектируете ли вы самолет, прогнозируете ли вы погоду или просто хотите «приручить» свои данные, «Искусство программирования на R» станет руководством по использованию всей мощи статистических вычислений.

ОБ АВТОРЕ

Норман Мэтлофф — профессор Computer Science и статистики, специализирующийся на параллельной обработке и регрессионном анализе.

ISBN: 978-5-4461-1101-5

no starch press

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг и интернет-магазин

instagram.com/piterbooks
youtube.com/ThePiterBooks
vk.com/piterbooks
facebook.com/piterbooks