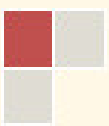
У. Н. Венэбльз, Д. М. Смит и
Рабочая группа разработки R

Введение в R

Заметки по R: среда программирования для анализа данных и графики

Версия 3.1.1 (2014-07-10)

Москва, 2014



An Introduction to R

Notes on R: A Programming Environment for Data Analysis and Graphics

Version 3.1.1 (2014-07-10)

W. N. Venables, D. M. Smith

and the R Core Team

Это руководство для R, версия 3.1.1 (2014-07-10).

ISBN

Перевод с английского А.А.Фоменко
По общим вопросам обращаться по адресу:
http://www.aafomenko@yandex.ru

У. Н. Венэбльз, Д. М. Смит
и Рабочая группа разработки *R*

Введение в R

Заметки по R: среда программирования для анализа данных и графики

Версия 3.1.1 (2014-07-10)

Перевод и редакция А.А.Фоменко. – Москва, 2014. 109 с. – (серия технической документации).

Данная книга является переводом одноименно книги из комплекта технической документации, поставляемой в составе поставки ***R,*** и призвана восполнить пробел в русской локализации системы ***R***.

Оглавление

Предисловие

Данное введение в ***R*** получено из исходного набора примечаний, описывающих среду ***S*** и *SPlus*, написанных в 1990-2 Биллом Венэбльзом и Дэвидом М. Смитом в университете Аделаиды. Сделано много небольших изменений для отражения различий между программами ***R*** и ***S,*** и развернули часть материала.

Выражаем искреннюю благодарность Биллу Венэбльзу (и Дэвиду Смиту), гарантировавших разрешение распространения этой модифицированной версии заметок, поддержав, таким образом, ***R*** от пути назад.

Комментарии и исправления всегда приветствуются. Пожалуйста, адресуйте корреспонденцию на электронную почту ***R****-core@R -project.org*.

Предложения читателю

Большинство новичков ***R*** начнет с вводного сеанса в Приложении А. Он познакомит со стилем сеансов ***R*** и, что еще более важно, даст некоторое впечатление о том, что фактически происходит.

Многие пользователи придут в ***R,*** главным образом, из-за его графических средств. Смотри Главу 12 [Графика], которую можно прочесть в почти любое время и не следует ожидать усвоения всех предыдущих разделов.

О переводе

Данная книга является переводом документации, доступной на английской языке в составе дистрибуции ***R***. После установки оригинал перевода доступен по адресу *\каталог R\doc\manual\R-intro*. Если данный файл перевода переименовать в *R-intro* и заменить оригинальный файл на данный, то из справки по ***R*** будет доступен данный перевод.

Перевод выполнен полностью за некоторыми отличиями:

- в некоторых местах исключены тексты, относящиеся к иным ОС, кроме Windows;
- исключены справочные приложения, в которых были собраны ссылки на функции и термины в английском тексте;
- расширен раздел по пакетам за счет описания пакетов, применяемым в эконометрике.

Переводчик будет благодарен за выявленные ошибки и неточности.

1. Введение и предварительные замечания

1.1. Среда *R*

R представляет собой набор программных средств для манипулирования данными, вычисления и графического отображения. Кроме этого возможно:

- эффективная обработка и хранение данных;
- набор операторов для вычислений на массивах, особенно матрицах,
- цельная, непротиворечивая, комплексная коллекция утилит для анализа данных,
- графические средства для анализа данных и отображения или непосредственно на компьютере или при выводе на печать, и
- хорошо разработанный, простой и эффективный язык программирования (называемый '***S***'), который включает условные выражения, циклы, определяемые пользователем рекурсивные функции и средства ввода и вывода. Действительно, большинство поддерживаемых системой функций сами написаны на языке ***S***.

Термин "окружение/среда" предназначен, чтобы характеризовать ее как полностью запланированную и последовательную систему, а не постепенно возникшего конгломерата весьма специфических и негибких инструментов, как часто имеет место с другим программным обеспечением анализа данных.

R является средством разработки методов интерактивного анализа данных. Она была разработана быстро и была расширена большим количеством пакетов. Однако, большинство программ, написанных в ***R***, принципиально являются программами-однодневками, написанными для конкретного случая анализа данных.

1.2. Связанное программное обеспечение и документация

R можно рассмотреть как реализацию языка ***S***, который разработан в Bell Laboratories Риком Беккером, Джоном Чемберсом и Алланом Уилксом, и который собственно лежит в основе систем *S-Plus*.

Эволюция основ языка ***S*** характеризуется четырьмя книгами Джона Чемберса с соавторами. Для ***R*** основой является «Новый Язык S: Среда программирования для анализа данных и графики», написанной Ричардом А. Беккером, Джоном М. Чемберсом и Алланом Р. Уилксом. Новые функции ***S,*** опубликованные 1991, даны в «Статистических моделях в S», отредактированном Джоном М. Чемберсом и Тревором Дж. Хэсти. Формальные методы и классы пакета методов основаны на описанных в «Программировании с данными» Джоном М. Чемберсом. См. Приложение F [Ссылки], для точной ссылки.

Сейчас есть много книг, которые описывают использование ***R*** для анализа данных и статистики, и документация для S/S-Plus может, как правило, использоваться с ***R,*** если помнить различия между реализациями ***S***. См. Раздел, "Какая документация существует для ***R***?" в статистическом системном FAQ ***R*** .

1.3. *R* и статистика

Наше введение в среду ***R*** не упоминает статистику, но много людей используют ***R*** в качестве системы статистики. Мы предпочитаем думать о ней как о среде, в пределах которой были реализованы много классических и современных статистических методов. Некоторые из них встроены в основу среды ***R,*** но многие предоставлены как пакеты. В составе ***R*** существует около 25 пакетов (названных "стандартными" и "рекомендуемыми" пакетами), и еще больше доступно через семейство сайтов CRAN (через *http://CRAN.R -project.org)* и из других источников. Более подробную информацию о пакетах рассмотрим позже (см. Главу 13 [Пакеты]).

Большинство классических статистик и многое из последних методик доступно для использования в ***R,*** но пользователи должны быть готовы к небольшим усилиям, чтобы найти нужное.

Есть важное различие в философии между ***S*** (и, следовательно, ***R***) и другими основными статистическими системами. В ***S*** статистический анализ обычно делается как ряд шагов с промежуточными результатами, сохраненными в объектах. Таким образом, тогда как SAS и SPSS дадут обильные результаты регрессионного или дискриминантного анализа, ***R*** выведет минимум результатов и сохранит их в подогнанном объекте для последующего использования функциями ***R.***

1.4. *R* и оконная система

Самый удобный способ пользоваться ***R*** – это использовать графическую рабочую станцию с окнами. Это руководство нацелено на пользователей, у которых есть это средство. В особенности мы будем иногда обращаться к использованию ***R*** на Windows XP, хотя обширный объем того, что сказано, обычно применим к любой реализации среды ***R.***

Большинство пользователей, время от времени, непосредственно сталкивается с операционной системой на своем компьютере. В этом руководстве, главным образом, обсуждается взаимодействие с операционной системой на машинах UNIX. Если ***R*** исполняется под Windows или Mac OS, то будет необходимо внести некоторые небольшие корректировки.

Установка рабочей станции, чтобы в полной мере воспользоваться настраиваемыми функциями ***R,*** является простой, хотя и несколько утомительной процедурой и здесь рассматриваться не будет. При трудностях пользователи должны искать местного опытного специалиста.

1.5. Использование *R* в интерактивном режиме

При использовании программы ***R*** она выдает запрос ожидания входных команд. Запрос по умолчанию **‘>’**, который на UNIX совпадает с запросом оболочки, и таким образом, может казаться, что ничего не происходит. Однако, как увидим, при желании легко изменить на другой запрос ***R.*** Мы предположим, что запрос оболочки UNIX - ‘$’.

В использовании ***R*** под UNIX предложенная процедура для первого случая следующая:

1. Создать отдельный подкаталог, скажем '*work*' для файлов с данными, на которых Вы будете использовать ***R*** для своей задачи. Это будет рабочим каталогом всякий раз при использовании ***R*** для этой определенной задачи.

   $ mkdir work

   $ cd work

2. Начать программу ***R*** командой

   $ R

3. Здесь можно давать команды
4. Для завершения программы ***R*** введите:

   q()

В этом этапе Вас спросят, хотите ли Вы сохранить данные своего сеанса ***R.*** На некоторых системах это будет сделано с помощью диалогового окна, а на других Вы получите текстовый запрос, на который Вы можете ответить «да», «нет» или «отмена» (достаточно будет ввести первую букву), чтобы сохранить данные перед выходом, выйти без сохранения, или вернуться в сеанс ***R.*** Сохраненные данные будут доступны в будущем сеансе ***R***.

Дальнейшие сеансы ***R*** требуют меньше действий.

1. Сделайте '*work*' рабочим каталогом и запустите программу как прежде:

$ cd work

$ R

2. Используйте программу ***R,*** которая завершится командой *q()* в конце сеанса.

Для использования ***R*** под Windows процедура в основном такая же. Создайте папку как рабочий каталог, и установите его в поле '*Start In*' ярлыка ***R.*** Затем запустите ***R,*** дважды щелкая по иконке.

1.6. Первый сеанс

Читателям, желающим испытать ***R*** на компьютере, прежде чем приступить настоятельно советуем проработать вводный сеанс, данный в Приложении А [Сеанс выборки].

1.7. Получение справки по функциям и средствам

У ***R*** есть встроенное справочное средство. Чтобы получить больше информации о любой определенной именованной функции, например *solve*, напишите команду:

> help(solve)

Альтернатива:

>? solve

Для средств, указанных специальными символами, параметр должен быть включен в двойные или одинарные кавычки, делая его "символьной строкой": это также необходимо для нескольких слов с синтаксическим значением, включая *if, for* и *function.*

> help("[[")

Может использоваться любая форма символа кавычки для исключения других кавычек, как в строке, "It's important". Наше соглашение состоит в предпочтительности использования символа двойной кавычки.

На большинстве установок ***R*** справка доступна в формате HTML, достаточно выполнить:

> help.start ()

что запустит Веб-браузер, предоставляющий возможность использовать гиперссылки в справке. Ссылки *'Search Engine and Keywords'* на странице, загруженной *help.start(),* особенно полезны, так как содержат высокоуровневый список понятий, используемый при поиске доступных функций. Это может оказаться отличным способом быстро решить проблемы и понять широкий спектр возможностей ***R*** .

Команда *help.search* (альтернативно ??) позволяет искать справку различными способами.

Например,

>?? solve

Попробуйте *?help.search* для деталей и большего количества примеров.

Пример на теме справки обычно можно выполнить:

> example(topic)

У версий ***R*** для Windows есть другие дополнительные системы справочной информации. Используйте:

>?help

для получения дальнейшей информации.

1.8. Команды *R*, учет регистра и т.д.

Технически ***R*** является языком выражений с очень простым синтаксисом. Он учитывает регистр, как большинство других программ UNIX, таким образом, ***A*** и ***a*** являются различными символами и ссылаются на разные переменные. Набор символов, которые могут использоваться для имен ***R,*** зависит от операционной системы и страны, в которой ***R*** исполняется (технически говоря, от используемой локали - *locale*). Обычно разрешены все алфавитно-цифровые символы плюс '.' и '_', с ограничением, что имя должно начинаться с «.» или буквы, и если начинается с «.», то второй символ не может быть цифрой. Имена в настоящий момент фактически неограниченны, но были ограничены 256 байтами до ***R 2.13.0.***

Простые команды состоят из выражений (*expression*), либо присвоений (*assignments*) . Если выражение вводится как команда, то оно вычисляется, выводится (пока специально не сделано невидимым) и значение теряется. Присвоение также вычисляет выражение и передает значение переменной, но результат автоматически не выводится.

Команды разделены либо точкой с запятой (';'), либо новой строкой. Простые команды могут группироваться в одно составное выражение фигурными скобками (' {' и'}'). Комментарии могут быть помещены практически где угодно, начинаясь со знака "решетки" ('#'), при этом все до конца строки является комментарием.

Если команда не полна в конце строки, то ***R*** даст особое приглашение, по умолчанию:

+

на второй и последующих строках и продолжит читать ввод, пока команда синтаксически не полна. Этот запрос может быть изменен пользователем. Мы, как правило, будем опускать приглашение продолжения ввода и обозначим продолжение простым отступом.

Командные строки, вводимые на консоли, ограничены в размере доя 4095 байт (не символов).

1.9. Повтор и коррекция предыдущих команд

R обеспечивает механизм для повторного вызова и выполнения предыдущих команд. Вертикальные клавиши со стрелками на клавиатуре могут использоваться для прокрутки вперед и назад по истории команд. Как только команда локализована таким способом, курсор может быть перемещен в пределах команды, используя горизонтальные клавиши со стрелками, и символы могут быть удалены клавишей DEL или добавлены другими клавишами. Более подробная информация предусмотрена далее: см. Приложение C [Редактор командной строки].

Кроме того, редактор текста Emacs предоставляет более полный механизм поддержки (через ESS – Emacs Speaks Statistic) для интерактивной работы с ***R.*** Смотри раздел "***R*** and Emacs" в The ***R*** statistical system FAQ.

1.10. Выполнение команд из файла или перенаправление вывода в файл

Если команды были сохранены во внешнем файле, скажем '*command.R* ' в рабочем каталоге '*work*', то они могут быть выполнены в любое время в сеансе ***R*** командой:

*> source("commands.**R** ")*

Для Windows ***Source*** также доступен в меню File. Функция ***sink:***

> sink("record.lis")

отклонит весь последующий вывод консоли во внешний файл *‘record.lis’*. Команда

> sink()

восстановит вывод в консоли еще раз.

1.11. Сохранение данных и удаление объектов

Сущности, которые ***R*** создает и манипулирует, известны как объекты *(object)*. Они могут быть переменными, массивами чисел, символьными строками, функциями или более общими структурами, построенных из таких компонентов.

Во время сеанса ***R*** объекты создаются и хранятся по имени (мы обсуждаем этот процесс в следующем сеансе). Команда ***R:***

> objects()

(также как ***ls()*** может использоваться для вывода на экран имен (в основном) объектов, которые в настоящий момент хранятся в пределах ***R.*** Набор объектов, сохраненных в настоящий момент, называют рабочей областью *(workspace)*.

Для удаления объектов доступна команда *rm*:

> rm(x, y, z, ink, junk, temp, foo, bar)

Все объекты, создаваемые во время сеанса ***R,*** могут храниться постоянно в файле для использования в будущем сеансе ***R.*** В конце каждого сеанса ***R*** предоставляется возможность сохранить все в имеющиеся в данный момент объекты. Если подтвердить необходимость этого, то объекты записываются в файл, называемый *‘.RData’* в текущем каталоге, а строки команд, использованных в сеансе, сохраняются в файл ‘*.Rhistory*’.

При последующем запуске ***R*** рабочая область загружается из этого файла. Одновременно загружается присоединенная история команд.

Рекомендуется использовать отдельные рабочие каталоги для анализов, проводимых с ***R*** . Очень распространено использовать для анализа объекты с именами ***x*** и ***y***. Подобные имена часто значимы в контексте отдельного анализа, но может быть довольно трудно решить то, чем они отличаются, если несколько анализов было выполнено в одном и том же каталоге.

2. Простые манипуляции; числа и векторы

2.1. Вектора и присваивания

R оперирует именованными структурами данных (*data structures)*. Простейшая такая структура – это числовой вектор, который является отдельным объектом, состоящим из упорядоченного набора чисел. Чтобы создать вектор с именем ***x***, скажем, состоящий из пяти чисел, а именно, 10.4, 5.6, 3.1, 6.4 и 21.7, используют команду ***R:***

> x <- c (10.4, 5.6, 3.1, 6.4, 21.7)

Это - оператор присваивания, использующий функцию ***c(),*** которая в этом контексте может взять произвольное число аргументов вектора и значением которой является вектор, полученный путем объединения аргументов конец с концом.

Отдельное число, входящее в выражении, трактуется как вектор единичной длины.

Учтите, что оператор присваивания (‘ <-’), который состоит этих двух символов ‘<’ (“меньше чем”) и ‘-’ ("минус"), выполняется однонаправленно и 'указывает' на объект, получающий значение выражения. В большинстве случаев можно использовать оператор ‘=’ в качестве альтернативы.

Также можно сделать присвоение, используя функцию *assign()*. Эквивалентный выше приведенному способ присвоения выглядит как:

> assign ("x", c (10.4, 5.6, 3.1, 6.4, 21.7))

Обычный оператор <- может считаться синтаксическим сокращением для него.

Присвоения могут также быть сделаны в другом направлении, используя очевидное изменение в операторе присваивания. Таким образом, то же самое присвоение могло быть сделано, используя:

> c (10.4, 5.6, 3.1, 6.4, 21.7)-> x

Если выражение используется в качестве полной команды, значение печатается и теряется. Итак, если бы нам пришлось использовать команду:

> 1/x

то обратные величины пяти значений были бы напечатаны в терминале (а значение ***x***, конечно, не изменилось).

Дальнейшее присваивание:

> y <-c (x, 0, x)

создаст вектор y с 11 элементами, состоящими из двух копий ***x*** и нулем между ними.

2.2. Векторная арифметика

Векторы могут использоваться в арифметических выражениях, и в этом случае операции выполняются поэлементно. Векторы, используемы в одном выражении, не обязательно должны иметь одинаковую длину. Если длины отличаются, то результат выражения – это вектор с длиной самого длинного вектора, который находится в выражении. Короткие векторы в выражении используются повторно столько раз, сколько это необходимо (возможно не целое число раз), до тех пор, пока они не совпадут с длиной самого длинного вектора. В частности константа просто повторяется. Так, учитывая предыдущие присваивания, команда:

*> v <-2*x + y + 1*

создаст новый вектор ***v*** длины 11, составленный путем сложения элемент за элементом *2*x* повторенного 2.2 раза, ***y*** повторяется только раз, и 1 повторяется 11 раз.

Элементарными арифметическими операторами являются обычные +, -, *, / и ^ для возведения в степень. Дополнительно присутствуют все простые арифметические функции. *log, exp, sin, cos, tan, sqrt* и так далее, все они имеют свое обычное значение. *max* и *min* выбирают самые большие и наименьшие элементы вектора соответственно. *range* является функцией, значение которой - вектор длины два, а именно, *c(min(x), max(x)). length(x)* является числом элементов в ***x***, *sum(x)* дает сумму всех элементов ***x,*** и *prod(x)* их произведение.

Двумя статистическими функциями являются *mean(x),* которая вычисляет среднее выборки, что соответствует *sum(x)/length(x),* и *var (x),* которая дает

sum((x-mean(x))^2)/(length(x)-1)

или дисперсию выборки. Если параметром *var()* является *n-на-p* матрица, значение - матрица ковариации выборки *p-на-p*, полученная путем интерпретации строк как ***p*** независимых векторов-выборок.

sort(x) возвращает вектор того же размера как ***x*** с элементами, расположенными в возрастающем порядке; однако доступны и другие более гибкие средства сортировки (см. *order()*, или *sort.list(),* которые производят перестановку при сортировке).

Заметим, что *max* и *min* выбирают самое большое и наименьшее значение в их аргументах, даже если им дают несколько векторов. Параллельные функции максимума и минимума *pmax* и *pmin* возвращают вектор (длины, равной их самому длинному аргументу), который содержит в каждом своем элементе наибольшее (наименьшее) значение на этой позиции во всех входных векторах.

В большинстве случаев пользователю не важно, являются ли "числа" в числовых векторах целыми, реальными или даже комплексными. Внутренние расчеты осуществляются в реальных числах двойной точности, или комплексных числах двойной точности, если входные данные являются комплексными.

Для работы с комплексными числами надо явно предоставить мнимую часть. Таким образом:

sqrt (-17)

даст NaN и предупреждение, но

sqrt (-17+0i)

сделает вычисления как комплексных чисел.

2.3. Генерация регулярных последовательностей

У ***R*** есть много средств для генерации используемых последовательностей обычных чисел. Например, *1:30* является вектором *c(1, 2..., 29, 30).* У оператора двоеточия есть высокий приоритет в пределах выражения, таким образом, например *2*1:15* является вектором *c(2, 4..., 28, 30*). Введите *n <-10* и сравните последовательности *1:n-1* и *1: (n-1).*

Выражение *30:1* может использоваться для создания обратной последовательности.

Функция *seq()* является более общим средством для генерации последовательности. У нее имеется пять параметров, только некоторые из которых могут специфицироваться в любом вызове. Первые два параметра, если дано, специфицируют начало и конец последовательности, и если только эти два параметра, то результат аналогичен оператору двоеточия. Например, *seq(2,10)* дает такой же вектор как 2:10.

Аргументы для *seq()* и ко многим другим функциям ***R,*** могут также быть даны в именованной форме, когда порядок, в котором они появляются, не важен. Первые два аргумента можно назвать *from=value* и *to=value*; таким образом *seq(1,30), seq (from=1, to=30)* и *seq (to=30, from=1)* являются одинаковыми с 1:30. Следующие два аргумента для *seq()* можно назвать *by=value* и *length=value*, которые специфицируют размер шага и длину для последовательности соответственно. Если ни один из них не дан, то по умолчанию предполагается *by=1*.

Например:

> seq(-5, 5, by=.2) -> s3

генерирует вектор *c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0)*. Подобно этому:

> s4 <- seq(length=51, from=-5, by=.2)

генерируется аналогичный вектор.

Пятый аргумент можно назвать *along=vector*, который используется как единственный аргумент и создает последовательность *1, 2..., length(вектор)*, или пустую последовательность, если вектор пуст (такое тоже может быть).

Соответствующая функция *rep()*, которую можно использоваться для тиражирования объекта различными сложными способами. Самая простая форма:

> s5 <- rep(x, times=5)

которая поместит пять копий x от начала до конца в s5. Другая полезная версия

> s6 <- rep(x, each=5)

которая повторит каждый элемент ***x*** пять раз перед пересылкой в следующую.

2.4. Логические векторы

Так же как числовые векторы, ***R*** позволяет манипулирование логическими величинами. У элементов логического вектора могут быть значение TRUE, FALSE, и NA (для “не доступно”, см. ниже). Первые два часто сокращаются как ***T*** и ***F***, соответственно. Заметим, однако, что ***T*** и ***F*** - только переменные, которые установлены в TRUE и FALSE по умолчанию, но не зарезервированные слова и, следовательно, могут быть перезаписаны пользователем. Следовательно, следует всегда использовать TRUE и FALSE.

Логические векторы генерируются условиями. Например:

> temp <- x > 13

устанавливает *temp* как вектор одинаковой длины как ***x*** со значением FALSE, соответствующих тем элементам ***x***, где условие не соблюдается, и TRUE, где имеет место.

Логическими операторами являются <, <=, >,>=, == для точного равенства и *!=* для неравенства. Кроме того, если ***c1*** и ***c2*** - логические выражения, то ***c1&c2*** - их пересечение ("и"), ***c1|c2*** - их объединение ("или"), и ***!c1*** - отрицание ***c1***.

Логические векторы могут использоваться в обычной арифметике, в том случае они преобразуются в числовые векторы, FALSE равно ***0*** и TRUE равно ***1***. Однако есть ситуации, где логические векторы и их преобразованные числовые дубликаты не эквивалентны, например см. следующий подраздел.

2.5. Пропущенные значения

В некоторых случаях компоненты вектора не могут быть полностью известны. Когда элемент или значение “не доступно” или "отсутствует значение" в статистическом смысле, место в пределах вектора может быть зарезервировано,

присваивая ему специальное значение NA. Вообще любое действие с NA дает NA. Обоснование этого правила просто состоит в том, что, если спецификация действия является неполной, то результат не может быть известен и, следовательно, не доступен.

Функция *is.na(x)* дает логический вектор одинакового размера с ***x*** со значением TRUE, если и только если соответствующий элемент в ***x*** равен NA.

```
> z <- c(1:3,NA); ind <- is.na(z)
```

Заметим, что логическое выражение *x == NA* очень отличается от *is.na(x),* так как NA не действительно значение, а маркер для количества, которое не доступно. Таким образом, *x == NA* - вектор одинаковой длины с ***x,*** все значения которого равны NA, поскольку само логическое выражение является неполным и следовательно неразрешимым.

Заметим, что существует второй вид "пропущенного" значения, которое произведено числовым вычислением, так называемое значение «Не Число» - NaN. Пример:

```
> 0/0
```

или

```
> Inf – Inf
```

который оба дают NaN, так как результат не может быть определен заметно.

В итоге, *is.na(xx)* равно TRUE и для NA и для значения NaN. При дифференцировании их, *is.nan(xx)* равно TRUE только для NaN.

Отсутствующие значения иногда печатаются как <NA>, когда символьные векторы напечатаны без кавычек.

2.6. Векторы символов

Символьные количества и символьные векторы часто используются в ***R,*** например, как метки рисунка. Где необходимо они обозначены последовательностью символов, разграниченных символом двойной кавычки, например, *"x-значением", "Новая итерация заканчивается".*

Символьные строки вводятся, используя любые двойные (") или одинарные (’) кавычки, но напечатаны, используя двойные кавычки (или иногда без кавычек). Они используют *escape*-последовательности С-стиля, используя \, поскольку символ ESC, таким образом, \\вводится и печатается как \\, и в двойных кавычках, "вводится как \". Другие полезные escape-последовательности: ***\n*** - новая строка, ***\t*** – табуляция и ***\b*** - клавиша Backspace – смотри *?Quotes* для полного списка.

Символьные векторы могут быть связаны в вектор функцией c(); примеры их использования будут часто появляться.

Функция *paste()* берет произвольное число параметров и связывает их один за другим в символьные строки. Любые числа, данные среди параметров, принуждены в символьные строки очевидным способом, то есть, таким же образом они были бы таковыми при печати. Параметры по умолчанию разделены в результате одиночным знаком пробела, но это может быть изменено именованным аргументом *sep=string,* который изменяет их на строку, возможно пустую.

Например:

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

преобразует *labs* в символьный вектор

```
c("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Заметим отдельно, что рециркуляция коротких списков также имеет здесь место; таким образом, *c("X","Y")* повторен 5 раз для соответствия *последовательности 1:10*.

2.7. Векторы индексов; выбор и изменение подмножеств наборов данных

Подмножества элементов вектора могут быть выбраны путем добавления к имени вектора *индексного вектора* в квадратных скобках. Более широко у любого выражения, которое оценивает вектор, может быть подмножества его элементов, так же выбранных путем добавления индексного вектора в квадратных скобках сразу после выражения.

Такой индексный вектор может быть любым из четырех различных типов.

1. **Логический вектор**. В этом случае индексный вектор рециклично приводится к той же самой длине как вектор, из которого должны быть выбраны элементы. Значение, соответствующее TRUE в индексном векторе, выбрано, и те, которые соответствуют FALSE, опущены. Например:

   > y <-x [! is.na (x)]

создает (или воссоздает) объект ***y***, который будет содержать не несуществующие (только существующие) значения ***x*** в том же самом порядке. Заметим, что, если у ***x*** есть отсутствующие значения, то ***y*** будет короче, чем ***x***. Также:

> (x+1) [(! is.na (x)) & x> 0]-> z

создает объект ***z*** и помещает в него значение вектора ***x+1***, для которого соответствующее значение в ***x*** и не пропущено и положительное.

2. **Вектор положительных целых величин**. В этом случае значение в индексном векторе должно лежать в наборе *{1, 2..., length(x)}*. Соответствующие элементы вектора выбраны и связаны в этом порядке в результате. Индексный вектор может иметь любую длину, и результат имеет одинаковую длину с индексным вектором. Например, *x[6]* является шестой компонентой ***x*** и

   > x [1:10]

   выбирает первые 10 элементов ***x*** (предполагается, что *length(x)* не меньше, чем 10). Также:

   > c("x","y")[rep(c(1,2,2,1), times=4)]

   (по общему признанию вещь маловероятная), производит символьный вектор длины 16, состоящий из "*x*", "*y*", "*y*", "*x*" повторенных четыре раза.

3. **Вектор отрицательных целых величин.** Такой индексный вектор указывает значение, которое будет исключаться, а не включаться. Таким образом:

   > y <- x[-(1:5)]

   даст все ***y*** кроме первых пяти значений.

4. **Вектор символьных строк.** Эта возможность применяется там, где у объекта есть атрибут имен для идентификации его компонентов. В этом случае подвектор вектора имен может использоваться таким же образом в качестве положительных целых меток в пункте 2 далее выше:

   > fruit <- c(5, 10, 1, 20)
   > names(fruit) <- c("orange", "banana", "apple", "peach")
   > lunch <- fruit[c("apple","orange")]

Преимущество состоит в том, что алфавитно-цифровые имена часто легче запомнить, чем числовые индексы. Эта опция особенно полезна в соединении с фреймами данных, как увидим позже.

Также индексное выражение может появиться на приемном конце присвоения, когда операция присвоения выполняется только на этих элементах вектора. Выражение должно иметь вектор вида *[index_vector]*, поскольку наличие произвольного выражения вместо векторного имени не имеет здесь большого смысла.

Присвоенный вектор должен соответствовать длине индексного вектора, и в случае логического индексируют вектор, у него должна снова быть та же самая длина как вектор, который он индексирует.

Например:

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

заменяет пропущенные значения в ***x*** на нули и

```
> y[y < 0] <- -y[y < 0]
```

имеет такой же результат как:

```
> y <- abs(y)
```

2.8 Другие типы объектов

Векторы - самый важный тип объекта в ***R,*** но есть несколько других, которых мы определим более формально в последующих разделах.

- *matrices* (матрицы) или более широко *arrays (*массивы) - многомерные обобщения векторов. Фактически, они - векторы, которые могут быть индексированы двумя или больше индексами и будут напечатаны специальными способами. См. Главу 5 [Массивы и матрицы].
- *factor*s (факторы) обеспечивают компактные способы обработки категорических данных. См. Главу 4 [Факторы].
- *lists* (список) - общая форма вектора, в котором различные элементы могут не иметь одинаковый тип, и являются часто самостоятельно векторами или списками. Списки обеспечивают удобный путь к возврату результатов статистического вычисления. См. Раздел 6.1 [Списки].
- *data frames* (фреймы данных) - подобные матрице структуры, в которых столбцы могут иметь различные типы. Думайте о фреймах данных как о ‘матрице данных’ с одной строкой на отдельное наблюдение, но с (возможно) и числовыми и категориальными переменными. Много экспериментов лучше всего описываются фреймами данных: обработки категоричны, но отклик является числовым. См. Раздел 6.3 [Фреймы данных].
- *functions* (функции) - самостоятельные объекты в ***R,*** которые можно сохранить в рабочей области проекта. Это обеспечивает простой и удобный способ расширения ***R*** . См. Главу 10 [Написание собственных функций].

3. Объекты, их режимы и атрибуты

3.1. Внутренние атрибуты: режим и длина

Рабочие сущности ***R*** технически известны как объекты. Примерами могут быть векторы с численными (реальными) или комплексными величинами, векторы с логическими значениями и векторы строк символов. Они известны как "атомарные" структуры, так как их компоненты имеют одинаковый тип или режим *(mode)*, а именно, *numeric, complex, logical, character и raw.*

У векторов должен быть одинаковый режим для всех значений. Таким образом, любой данный вектор должен быть однозначно или логическим, числовым, комплексным, символьным или строковым (*logical, numeric, complex, character* или *raw)*. Единственное очевидное исключение к этому правилу - специальное "значение", обозначаемое как NA для отсутствующих значений, хотя реально есть несколько типов NA. Заметим, что вектор может быть пустым и иметь режим. Например, пустой вектор символьной строки обозначается как character(0) и пустой числовой вектор как *numeric(0).*

R также работает с объектами, называемыми списками (*list*), которые имеют тип список (*list).* Существуют упорядоченные последовательности объектов, которые индивидуально могут иметь любой тип. Списки (*list)* известны как "рекурсивные", а не атомарные структуры, так как их компоненты могут самостоятельно быть списками.

Другие рекурсивные структуры из этого типа - это функции и выражения (*function* и *expression).* Функции – это объекты, которые являются частью системы ***R*** наряду с аналогичными написанными пользователем функциями, которые в деталях обсуждаются позже. Выражения, как объекты, составляют самую сложную часть ***R,*** которая не будет обсуждаться в этом руководстве, кроме как косвенно при обсуждении формул (*formulae)*, используемых при моделировании ***R***.

Типом (***mode***) объекта мы обозначили основной тип его фундаментальных свойств. Это - особый случай "свойств" объекта. Другое свойство каждого объекта - своя длина. Можно использовать функции *mode(object)* и *length(object)*, чтобы узнать тип и длину любой определенной структуры.

Другие свойства объекта обычно получают посредством *attributes(object),* смотри Раздел 3.3 [Получение и установка атрибутов]. Из-за этого тип и длину также называют “внутренними атрибутами” объекта.

Например, если ***z*** - комплексный вектор длины 100, то в выражении *mode(z)* является символьной строкой "*complex*", и *length(z)* равна *100.*

R обслуживает изменения типа практически везде, где это имеет смысл сделать, и иногда там, где этого не так. Например, с:

```
> z <-0:9
```

можно ввести

```
> digits <- as.character(z)
```

после которого *digits* является символьным вектором *c("0", "1", "2"..., "9")*. Дальнейшее приведение, или изменение типа, восстанавливает числовой вектор снова:

```
> d <- as.integer(digits)
```

Теперь ***d*** и ***z*** одинаковы. Существует большое количество функций вида *as.something()* или для приведения от одного типа к другому, или для наделения объекта некоторым другим атрибутом, которым он, возможно, еще не обладает.

Читатель должен консультироваться с различными справочными файлами для ознакомления.

3.2. Изменяющаяся длина объекта

«Пустой» объект может все еще иметь тип. Например:

```
> e <- numeric()
```

делает ***e*** пустой векторной структурой типа числовой (*numeric)*. Так же *character()* является пустым символьным вектором, и так далее. Как только объект любого размера был создан, новые компоненты могут быть просто добавлены к нему, давая ему значение индексов вне его предыдущего диапазона. Таким образом:

```
> e[3] <- 17
```

теперь делает ***e*** вектором длины 3, (первые две компоненты которого равны NA). Это применяется к любой структуре вообще, если тип дополнительного компонента (ов) согласован с типом первого объекта.

Эта автоматическая настройка длин объекта часто используется для ввода, например, в функции *scan()* (см. Раздел 7.2 [Функция scan()]).

Наоборот требуется усечение размера объекта для выполнения присвоения. Следовательно, если *alpha* - объект длины 10, то

```
> alpha <- alpha[2 * 1:5]
```

делает его объектом длины 5, состоящим только из прежних компонентов с четным индексом. (Старые индексы не сохранены, конечно). Затем можно сохранить только первые три значения:

```
> length(alpha) <- 3
```

и вектор может быть расширен (путем пропущенных значений) аналогичным образом.

3.3. Получение и установка атрибутов

Функция *attributes(object)* возвращает список всех не внутренних атрибутов, в настоящий момент определенных для этого объекта. Можно использовать функцию *attr(object, name)* для выбора определенного атрибута. Эта функция редко используются, за исключением довольно особых обстоятельствах, когда некоторый новый атрибут создается для некоторой конкретной цели, например чтобы присоединить дату создания или оператор с объектом ***R.*** Понятие, однако, очень важно.

Некоторое внимание должно быть уделено, когда присваиваются или удаляются атрибуты как неотъемлемая часть системы объекта, используемой в ***R.***

Когда такое используется на левой стороне присвоения, то оно может использоваться или для присоединения нового атрибута с *object* или изменения существующего. Например:

```
> attr(z, "dim") <- c(10,10)
```

позволяет ***R*** обрабатывать ***z*** как будто он является матрицей *10-на-10*.

3.4. Класс объекта

У всех объектов в ***R*** есть класс *(class)*, определяемы при помощи функции *class*. Для простых векторов это - только тип, например, *"numeric", "logical", "character"* или *"list",* но *"matrix", "array", "factor"* и *"data.frame"* " являются другими возможными значениями.

Специальный атрибут, известный как *class* (*класс)* объекта, используется для учета объектно-ориентированного стиля программирования в ***R***. Например, если у объекта будет класс "*data.frame*", то он будет напечатан определенным способом,

функция *plot()* выведет на экран его графически определенным способом, и другие, так называемые универсальные функции, такие как ***summary(),*** будут реагировать на него как на параметр, способом применимым к его классу.

Чтобы удалить временно эффект класса, используйте функцию *unclass()*. Например, если у *winter* есть класс "*data.frame*" то:

```
> winter
```

напечатает его в форме фрейма данных, которая скорее походит на матрицу, тогда как:

```
> unclass(winter)
```

напечатает его как обычный список. Только в довольно специальных ситуациях следует использовать это средство, но каждый раз для достижения согласования идеи класса и универсальных функций.

Универсальные функции и классы будут обсуждены далее в Разделе 10.9 [Ориентация объекта], но только кратко.

4. Упорядоченные и неупорядоченные факторы

Фактор - векторный объект, используемый для спецификации дискретной классификации (группировки) компонентов других векторов одинаковой длины. ***R*** поддерживает как *упорядоченные*, так и *не упорядоченные* факторы. Хотя "реальное" применение факторов имеет место в формулах модели (см. Раздел 11.1.1 [Противоположности]), здесь рассмотрим на специальный пример.

4.1. Специальный пример

Предположим, например, у имеется выборка 30 налоговых деклараций из всех штатов и территорий Австралии, и их индивидуальное происхождение указывается символьным вектором абревиатуры штата как:

```
> state <- c( "tas", "sa", "qld", "nsw", "nsw", "nt", "wa", "wa",
              "qld", "vic", "nsw", "vic", "qld", "qld", "sa", "tas",
              "sa", "nt", "wa", "vic", "qld", "nsw", "nsw", "wa",
              "sa", "act", "nsw", "vic", "vic", "act")
```

Заметим, что в случае символьного вектора, «*sorted*» означает сортировку в алфавитном порядке.

Создаются факторы аналогичным образом с помощью функции *factor():*

```
> statef <- factor(state)
```

Функция *print()* обрабатывает факторы несколько иначе, чем другие объекты:

```
> statef
[1] tas sa qld nsw nsw nt wa wa qld vic nsw vic qld qld sa
[16] tas sa nt wa vic qld nsw nsw wa sa act nsw vic vic act
Levels: act nsw nt qld sa tas vic wa
```

Чтобы выяснить уровни фактора можно использовать функцию *level()*:

```
> levels(statef)
[1] "act" "nsw" "nt" "qld" "sa" "tas" "vic" "wa"
```

4.2. Функция tapply () и массивы с переменной длиной строк

Чтобы продолжить предыдущий пример, предположим, что у нас есть доходы от каждого налогоплательщика в другом векторе (в подходящих крупных денежных единицах):

```
> incomes <- c(60, 49, 40, 61, 64, 60, 59, 54, 62, 69, 70, 42, 56,
               61, 61, 61, 58, 51, 48, 65, 49, 49, 41, 48, 52, 46,
               59, 46, 58, 43)
```

Чтобы вычислить средний доход в выборке по каждому штату используем теперь специальную функцию *tapply()*:

```
> incmeans <- tapply(incomes, statef, mean)
```

дающей вектор средних с компонентами, маркированными уровнями:

```
act     nsw     nt      qld     sa      tas     vic     wa
44.500  57.333 55.500 53.600 55.000 60.500 56.000 52.250
```

Функция *tapply()* используется для применения здесь функции *mean()* к каждой группе компонентов первого параметра, здесь доходов, определенные уровнями второго компонента, здесь *statef*, как будто они были отдельными векторными структурами. Результат - структура той же самой длины как атрибут уровней фактора,

содержащего результаты. Читатель должен консультироваться с документом справки для большего количества деталей.

Предположи далее, что необходимо вычислить стандартные ошибки средних доходов штата. Для этого следует записать функцию ***R*** для вычисления стандартной ошибки для любого данного вектора. Так как существует встроенная функция *var()* для вычисления дисперсии выборки, то такая функция записывается в виде одной строки, задаваемая присвоением:

```
> stderr <- function(x) sqrt(var(x)/length(x))
```

Написание функций рассмотрим позже в Главе 10 [Написание собственных функций], и в этом случае было ненужным, поскольку ***R*** также имеет встроенную функцию *sd()*. После этого присвоения стандартные ошибки вычислены:

```
> incster <- tapply(incomes, statef, stderr)
```

и затем вычисленные величины:

```
> incster
```

act	nsw	nt	qld	sa	tas	vic	wa
1.5	4.3102	4.5	4.1061	2.7386	0.5	5.244	2.6575

В качестве примера можно вычислить обычные 95%-ые доверительные границы для средних доходов штата. Для этого можно использовать *tapply()* еще раз с функцией *length()*, чтобы найти размеры выборки, и функцию *qt()*, чтобы найти процентные точки соответствующих *t*-распределений. Также можно рассмотреть средства ***R*** для t-тестов.

Также можно применить функцию *tapply()* к более сложной индексация вектора на несколько категорий. Например, можно разделить налоговые счета как по штатам, так и по полу. Однако в этом простом примере (только один фактор) то, что происходит, можно представить следующим образом. Значение в векторе собрано в группы, соответствующие различным позициям в факторе. Затем функция применяется к каждой из этих групп отдельно. Результат - это вектор значений функции, маркированных согласно *levels* фактора.

Комбинация вектора и фактора меток - пример того, что иногда называют *неровными массивами*, так как размеры подкласса возможно не одинаковы. Когда размеры подкласса всегда одинаковы, то индексация может быть сделана неявно и намного более эффективно, как мы увидим в следующем разделе.

4.3. Упорядоченные факторы

Уровни факторов сохраняются в алфавитном порядке, или в том порядке, в котором они указывались к фактору, если они указывались явно.

Иногда у уровней будет естественное упорядочивание, которое мы записали и хотели использовать в статистическом анализе. Функция *ordered()* создает такие упорядоченные факторы, но, в противном, она идентична *factor*. В большинстве целей единственной разницей между упорядоченными и неупорядоченными факторами является то, что прежне напечатанное для упорядоченных уровней отличается от генерируемых для них в подгонке линейных моделей.

5. Массивы и матрицы

5.1. Массивы

Массив *(array)* можно рассмотреть как умножение преобразованного в нижний индекс набора вводов данных, например числовых. ***R*** позволяет простые средства для создания и обработки массивов, и их особый случай - матриц.

Размерностью вектора является вектор неотрицательных целых чисел. Если его длина равна ***k,*** то массив является ***k***-мерным, например, матрица является 2-мерным массивом. Размерности индексированы от единицы до значения, данного в векторе размерности.

Вектор может использоваться в ***R*** в качестве массива, только если у него имеется вектор размерности как его атрибут *dim*. Предположим, например, ***z*** - вектор из 1500 элементов.

> *dim(z) <- c(3,5,100)*

дает ему атрибут *dim*, который позволяет его обрабатывать как массив *3-на-5-на-100*.

Другие функции, такие как ***matrix() и array()*** доступны для более простых и более естественно выглядящих присвоений, как мы увидим в Разделе 5.4 [Функция array()].

Значение в векторе данных дает значение в массиве в том же самом порядке, как они произошли бы в ФОРТРАНЕ, который является “столбцом главного порядка” с первым нижним индексом, изменяющимся быстрее, и последним самым медленным нижним индексом.

Например, если вектор размерности для массива, скажем ***a***, является *c(3,4,2)* то есть 3 * 4 * 2 = 24 записи в ***a*** и векторе данных содержит их в порядке *[1,1,1], [2,1,1], ..., [2,4,2], [3,4,2]*.

Массивы могут быть одномерными: такие массивы обычно обрабатываются таким же образом как векторы (включая, печать), но исключения могут вызвать беспорядок.

5.2. Индексация массива. Подразделы массива

На отдельные элементы массива можно сослаться, давая имя массива, сопровождаемого нижними индексами в квадратных скобках, разделенных запятыми.

Более широко можно указать подразделы массива, давая последовательность векторов индексов вместо нижних индексов; однако, если какая-либо позиция индекса дана пустым индексным вектором, то берется полный спектр этого нижнего индекса.

Продолжая предыдущий пример, *a[2,,]* является массивом 4x2 с вектором размерности *c(4,2)* и вектором данных, содержащим значение:

c ([2,1,1], [2,2,1], [2,3,1], [2,4,1],
[2,1,2], [2,2,2], [2,3,2], [2,4,2])

в том порядке. ***a[,,]*** стоит для всего массива, который является таким же с исключенными нижними индексами полностью и использованием ***a*** отдельно.

Для любого массива, скажем ***Z,*** на вектор размерности можно сослаться явно как *dim(Z)* (по обе стороны от присвоения).

Кроме того, если имя массива дано только с одним нижним индексом, или индексируется вектором, то только используется соответствующее значение вектора данных; в этом случае вектор размерности игнорируется. Дело обстоит не так, однако,

если отдельный индекс не вектор, но он непосредственно массив, как мы затем обсуждаем.

5.3. Индекс матрицы

Включая индексный вектор в любой позиции нижнего индекса, матрица может использоваться с отдельной *матрицей индексов* в порядке либо присвоения вектора количеству нерегулярной коллекции элементов в массиве, либо в извлечении нерегулярной коллекции как вектора.

Пример матрицы ясно дает понять процесс. В случае вдвойне индексированного массива индексная матрица может состоять из двух столбцов и так много строк как требуется. Входы в индексной матрице - строка и индексы столбца для вдвойне индексированного массива. Предположим, например, что у нас есть массив ***X*** *4-на-5*, и мы хотим сделать следующее:

- извлечь элементы *X[1,3], X[2,2] и X[3,1]* как векторную структуру, и
- заменить эти записи в массиве ***X*** нулями.

В этом случае необходим массив нижнего индекса *3-на-2*, как в следующем примере.

```
> x <- array(1:20, dim=c(4,5)) # генерирует массив 4 на 5.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
> i <- array(c(1:3,3:1), dim=c(3,2))
> i                                  # i является индексным массивом 3 на 2
     [,1] [,2]
[1,]    1    3
[2,]    2    2
[3,]    3    1
> x[i]                               # Извлекает те элементы
[1]  9  6  3
> x[i] <- 0                          # Заменяет те элементы нулями.
> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
```

Отрицательные индексы не разрешены при индексировании матрицы. *NA* и нулевое значение позволены: строки в индексной матрице, содержащей нуль, игнорируются, и строки, содержащие *NA*, производят *NA* в результате.

Как менее тривиальный пример, предположим, что необходимо генерировать (не приведенную) матрицу проекта для блочной конструкции, определенной блоками

факторов (***b*** уровни) и варианты (***v*** уровни). Далее предположим, что в эксперименте есть ***n*** рисунков. Мы могли продолжить следующим образом:

```
> Xb <- matrix(0, n, b)
> Xv <- matrix(0, n, v)
> ib <- cbind(1:n, blocks)
> iv <- cbind(1:n, varieties)
> Xb[ib] <- 1
> Xv[iv] <- 1
> X <- cbind(Xb, Xv)
```

Для конструирования индексов матрицы, скажем ***N***, можно использовать:

```
> N <- crossprod(Xb, Xv)
```

Однако более простым способом производства этой матрицы является использование *table()*:

```
> N <- table(blocks, varieties)
```

Индексная матрица должна быть числовой: предоставленная любая другая форма матрицы (логическая или символьная) обрабатывается как индексный вектор.

5.4. Функция array()

Так же, как давая векторной структуре атрибут *dim*, массивы могут быть созданы из векторов функцией массива (array), у которой есть форма:

```
> Z <- array(data_vector, dim_vector)
```

Например, если вектор ***h*** содержит 24 или менее чисел, тогда команда:

```
> Z <- array(h, dim=c(3,4,2))
```

использовал бы ***h***, чтобы установить *3-на-4-на-2* массив ***Z***. Если размер ***h*** точно 24, результат выглядит так:

```
> Z <- h ; dim(Z) <- c(3,4,2)
```

Однако, если ***h*** короче, чем 24, его значения будут взяты циклически для дополнения до размера 24 (см. Раздел 5.4.1 [Правило рециркуляции]), но *dim(h) <-c (3,4,2)* сигнализировал бы ошибку о несоответствии длине. Как экстремальный, но типичный пример:

```
> Z <- array(0, c(3,4,2))
```

делает ***Z*** массивом всех нулей.

В этой месте *dim(Z)* обозначает вектор размерности *c(3,4,2)*, и ***Z[1:24]*** содержит вектора данных, как это было в ***h***, и ***Z[]*** с пустым нижним индексом или Z состоит без нижнего индекса для всего массива в качестве массива.

Массивы могут использоваться в арифметических выражениях, и результат является массивом, сформированным поэлементно операциями на векторах данных. Атрибуты *dim* операндов обычно должны быть одинаковыми, и они становятся вектором размерности результата. Так, если ***A, B*** и ***C*** являются все подобными массивами, то:

```
> D <-2*A*B + C + 1
```

делает ***D*** подобным массивом с его вектором данных, являющимся результатом данной поэлементно операции. Однако точное правило относительно смешанного массива и векторных вычислений нужно рассмотреть немного более тщательно.

5.4.1. Смешанный вектор и арифметика массива. Правило рециркуляции

Точное правило, влияющее поэлементно на смешанные вычисления с векторами и массивами, является более изощренными и точно описаны в ссылках. От опыта мы нашли следующее надежное руководство.

- Выражение просматривается слева направо.
- Любые короткие векторные операнды расширяются циклически их значениями, пока они не совпадет с размером любых других операндов.
- Если длинные и короткие вектора с массивами лишь пересчитываются, то все массивы должны иметь одинаковый атрибут *dim* или будет выдана ошибка.
- Любой векторный операнд более длинный, чем операнд матрицы или массива, генерирует ошибку.
- Если имеются структуры массива, и нет ошибок, или приведение к вектору было выполнено, то результат - структура массива с общим атрибутом *dim* ее операндов массива.

5.5. Внешнее произведение двух массивов

Важная работа на массивах – внешнее произведение. Если ***a*** и ***b*** - два числовых массива, их внешнее произведение - массив, вектор размерности которого получен, связывая их два вектора размерности (порядок важен), и чей вектор данных получен путем формирования всех возможных произведений элементов вектора ***a*** с соответствующим элементами вектора ***b***. Внешнее произведение выполняется специальным оператором %o%:

```
> ab <-%o % b
```

Альтернатива

```
> ab <- outer(a, b, "*")
```

Функция умножения может быть заменена произвольной функцией двух переменных. Например, если необходимо оценить функцию $f(x; y) = cos(y)/(1 + x^2)$ на регулярной сетке значения с ***x***-и ***y***-координатами, определенными векторами ***R*** ***x*** и ***y*** соответственно, можно продолжить следующим образом:

```
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

В особенности внешнее произведение двух обычных векторов - вдвойне преобразованный в нижний индекс массив (который является матрицей ранга самое большее 1). Заметьте, что оператор внешнего произведения, конечно, некоммутативен. Определение Ваших собственных функций ***R*** рассмотрят далее в Главе 10 [написание собственных функций].

5.6. Обобщенное транспонирование массива

Можно использовать функцию *aperm(a, perm)* для перестановки массива ***a***. Параметром *perm* должна быть перестановка целых чисел {1, …. , к}, где ***k*** является номером нижних индексов в ***a***. Результат функции - массив того же самого размера как ***a***, но со старой размерностью, вычисленной *perm[j]*, становящейся новой ***j-й*** размерностью. Самый простой способ понимания этой операции – это обобщение транспонирования для матриц. Действительно, если ***A*** - матрица, (то есть, вдвойне преобразованный в нижний индекс массив) тогда ***B*** вычисляется путем:

```
> B <-aperm (A, c (2,1))
```

и представляет собой транспонирование. Для этого особого случая доступна более простая функция ***t()***, таким образом, мы, возможно, использовали *B <-t (A)*.

5.7. Матричные инструменты

Как отмечено выше, матрица - это массив с двумя нижними индексами. Однако это такой важный особый случай, что нуждается в отдельном обсуждении. ***R*** содержит много операторов и функций, которые доступны только для матриц. Например, *t(X)* – функция транспонирующая матрицу, как отмечено выше. Функции *nrow (A)* и *ncol (A)* дают число строк и столбцов в матрице соответственно.

5.7.1. Умножение матриц

Оператор %*% используется для умножения матриц. Матрицы *n-на-1* или *1-на-n* могут, конечно, использоваться в качестве *n*-вектора, если это соответствует контексту. Наоборот, векторы, которые встречаются в выражениях умножения матриц, автоматически расширяются или на вектор строки или на вектор столбца, какой бы ни являлся мультипликативно когерентным, если возможный, (хотя это не всегда однозначно возможно, как мы видим позже).

Если, например, ***A*** и ***B*** - квадратные матрицы одинакового размера, то

```
> A * B
```

матрица поэлементно произведений и

```
> % * % B
```

матричное произведение. Если ***x*** - вектор, то

```
> x  %*%  %*%x
```

квадратная форма.

Функция *crossprod()* формирует "векторные произведения", значение, что *crossprod(X, y)* является таким же как *t(X) %*% y*, но выполняется более эффективно. Если второй параметр *crossprod()* опущен, то получаем то же, что в первом случае.

Значение *diag()* зависит от ее аргумента. *diag*(v), где ***v*** - вектор, дает диагональную матрицу с элементами вектора в качестве диагональных значений. С другой стороны *diag*(*M*), где ***M*** является матрицей, дает вектор основных диагональных значений ***M***. Это одинаковое соглашение для *diag*() в Matlab. Кроме того, не очень четко, если ***k*** является единственным числовым значением, то *diag*(k) является *k-на-k* единичной матрицей!

5.7.2. Линейные уравнения и инверсия

Решение линейных уравнений является инверсией умножения матриц. Когда после

```
> b <- A %*% x
```

только A и b даны, вектор ***x*** является решением этой системы линейных уравнений. В ***R***

```
> solve(A, b)
```

решает систему, возвращая ***x*** (с некоторой потерей точности). Заметим, что в линейной алгебре, формально $x = A^{-1}b$, где A^{-1} обозначает инверсию A, которая может быть вычислена:

```
solve(A)
```

но редко необходимо. В цифровой форме это является и неэффективным и потенциально нестабильным, чтобы вычислить *x <- solve(A) %*% b* вместо *solve(A,b)*.

Квадратичная форма $\boldsymbol{x^T A^{-1} x}$, которая используется в многомерных вычислениях, должна быть вычислена подобно *x %*% solve(A,x)*, вместо вычисления инверсии ***A***.

5.7.3. Собственные значения и собственные векторы

Функция *eigen(Sm)* вычисляет собственные значения и собственные векторы симметричной матрицы *Sm*. Результат этой функции - список двух компонентов, названных значением и векторами.

Присвоение:

> *ev <- eigen(Sm)*

присвоит этот список ***ev***. Затем *ev$val* - вектор собственных значений ***Sm*** и *ev$vec* - матрица соответствующих собственных векторов. Если бы мы только нуждались в собственных значениях, мы, возможно, использовали присвоение:

> *evals <- eigen(Sm)$values*

evals теперь содержит вектор собственных значений, и второй компонент отброшен. Если выражение

> *eigen(Sm)*

используется отдельно в качестве команды, эти два компонента напечатаны с их именами. Для больших матриц лучше избежать вычисления собственных векторов, если они не необходимы при использовании выражениях:

> *evals <- eigen(Sm, only.values = TRUE)$values*

5.7.4. Сингулярное разложение и определители

Функция *svd(M)* берет произвольный матричный параметр ***M*** и вычисляет сингулярное разложение ***M.*** Она является матрицей с ортонормированными столбцами ***U*** с одинаковым пробелом столбца как ***M***, вторая матрица ортонормированных столбцов ***V***, чей пробел столбца - пробел строки ***M*** и диагональная матрица положительных значений ***D*** так, что *M = U % * % D % * % t(V).* ***D*** фактически возвращен как вектор диагональных элементов. Результатом *svd(M)* является фактически список трех компонентов, названных ***d, u*** и ***v*** с очевидными значениями.

Если ***M*** является фактически квадратной, то не трудно видеть, что:

> *absdetM <- prod(svd(M)$d)*

вычисляет абсолютное значение определителя ***M***. Если бы это вычисление часто было необходимо со множеством матриц, то это могло бы быть определено как функция ***R***

> *absdet <- function(M) prod(svd(M)$d)*

после которого мы могли использовать *absdet()* лишь как другую функцию ***R.*** Как дальнейший тривиальный, но потенциально полезный пример, Вам могло бы понравиться рассматривать запись функции, скажем *tr()* для вычисления трассировки квадратной матрицы. [Подсказка: не следует использовать явный цикл. Смотрите снова на функцию *diag()*.]

У ***R*** есть встроенная функция *det*, чтобы вычислить определитель, включая знак, и другая *determinant,* чтобы дать знак и модуль (дополнительно в логарифмическом масштабе).

5.7.5. Подгонка методом наименьших квадратов и QR разложение

Функция *lsfit ()* возвращает список, дающий результаты процедуры подгонки методом наименьших квадратов. Такое присвоение как:

> *ans <- lsfit(X, y)*

дает результаты подгонки методом наименьших квадратов, где ***y*** - вектор наблюдений, и ***X*** является проектируемой матрицей. См. справочное руководство для большего количества деталей, и также для последующей функции *ls.diag()*, между прочим, для

диагностики регрессии. Заметим, что главный средний параметр автоматически включен и не должен быть включен явно как столбец ***X***. Далее отметьте, что всегда предпочтительней использовать *lm()* (см. Раздел 11.2 [Линейные модели]) к *lsfit()* для моделирования регрессии.

Другая тесно связанная функция – это *qr()* и ее последователи. Рассмотрим следующие присвоения:

```
> Xplus <- qr(X)
> b <- qr.coef(Xplus, y)
> fit <- qr.fitted(Xplus, y)
> res <- qr.resid(Xplus, y)
```

Они вычисляют ортогональную проекцию ***y*** на диапазон ***X*** в *fit*, проекция на ортогональное дополнение в *res* и векторе коэффициента для проекции в ***b***, то есть, ***b*** является по существу результатом оператора 'наклонной черты влево' Matlab.

Не предполагается, что ***X*** имеет полный ранг столбца. Избыточность будет обнаружена и удалена при ее выявлении.

Эта альтернатива - более старый, низкий уровень способа выполнения вычислений наименьших квадратов. Хотя все еще полезный в некоторых контекстах, он был бы теперь заменен статистическими функциями моделей, как будет обсуждено в Главе 11 [Статистические модели в ***R***].

5.8. Формирование разделенных матриц cbind () и rbind ()

Как мы уже видели неформально, матрицы могут быть созданы из других векторов и матриц функциями *cbind() и rbind()*. Например, *cbind()* формирует матрицы путем связывания матриц горизонтально, или по столбцам, а *rbind()* вертикально, или по строкам.

В присвоении:

```
> X <-cbind (arg_1, arg_2, arg_3...)
```

параметрами *cbind()* должны быть или векторы любой длины, или матрицы с одинаковым размером столбца, который является одинаковым числом строк. Результатом является матрица с присоединенными *arg_1, arg_2...*, которые формируют столбцы.

Если некоторыми из параметров *cbind()* являются векторами, то они могут быть короче, чем размер столбца любых существующих матриц, так как они циклически расширяются, чтобы соответствовать размеру столбца матрицы (или длине самого длинного вектора, если нет матриц).

Функция *rbind()* делает соответствующую работу для строк. В этом случае любой векторный параметр, возможно циклически расширенный, конечно взят в качестве векторов строки.

Предположим, что ***X1*** и ***X2*** имеет одинаковое число строк. Объединить их по столбцам в матрице ***X***, вместе с начальным столбцом, равным 1, можно использованием:

```
> X <-cbind (1, X1, X2)
```

У результата *rbind()* или *cbind()* всегда статус матрицы. Следовательно, *cbind(x)* и *rbind(x)* являются, возможно, самыми простым способом обработать вектор ***x*** как столбец или строку матрицы соответственно.

5.9. Функция связывания массивов *c()*

Нужно отметить, что, тогда как *cbind()* и *rbind()* являются функциями связывания, которые принимают во внимание атрибуты размерности *dim*, базовая функция ***c()*** этого не делает, а скорее очищает числовые объекты от всех атрибутов *dim* и *dimnames*. Иногда это полезно само по себе.

Официальный способ преобразовать массив обратно к простому векторному объекту состоит в использовании *as.vector():*

```
> vec <-as.vector (X)
```

Однако подобный результат может быть достигнут при использовании ***c()*** только с одним параметром, просто в качестве побочного эффекта:

```
> vec <-c (X)
```

Между ними есть незначительные различия, но, в конечном счете, выбор между ними - в значительной степени вопрос стиля (прежний предпочтительней).

5.10. Таблицы частот от факторов

Вспомним, что фактор определяет разделение на группы. Подобно этому пара факторов определяет два пути классификация кросса и так далее. Функция *table()* вычисляет таблицы частот от факторов равной длины. Если есть параметры ***k***-фактора, результат - ***k*** путем массива частот.

Предположим, например, что *statef* - фактор, дающий код состояния для каждой записи в векторе данных. Присвоение:

```
> statefr <- table(statef)
```

дает в *statefr* таблицу частот каждого состояния в выборке. Частоты упорядочены и маркированы атрибутом уровней фактора. Этот простой случай эквивалентен, но более удобен чем:

```
> statefr <- tapply(statef, statef, length)
```

Далее предположим, что *incomef* - фактор, дающий соответственно определенный “поступивший класс” для каждой записи в векторе данных, например, с функцией *cut()*:

```
> factor(cut(incomes, breaks = 35+10*(0:7))) -> incomef
```

Затем вычислим двухвходовую таблицу частот:

```
> table(incomef,statef)
                statef
incomef  act  nsw  nt  qld  sa  tas  vic  wa
(35,45]  1     1   0   1    0   0    1    0
(45,55]  1     1   1   1    2   0    1    3
(55,65]  0     3   1   3    2   2    2    1
(65,75]  0     1   0   0    0   0    1    0
```

Расширение более высокого пути таблицы частот непосредственно.

6. Списки и фреймы данных

6.1. Списки

Список (*list*) – это объект ***R,*** состоящий из упорядоченного набора объектов, известных как его компоненты.

Отсутствуют определенные требования одинаковости режима или типа, и, например, список может состоять из числового вектора, логического значения, матрицы, комплексного вектора, символьного массива, функции и так далее. Вот простой пример создания списка:

```
> Lst <- list(name="Fred", wife="Mary", no.children=3,
              child.ages=c(4,7,9))
```

Компоненты всегда нумеруются и могут всегда упоминаться как таковые. Таким образом, если *Lst* имя списка с четырьмя компонентами, они могут индивидуально упоминаться как *Lst[[1]], Lst [[2]], Lst [[3]]* и *Lst [[4]]*. Если, далее, *Lst [[4]]* преобразованный в нижний индекс массив вектора тогда *Lst [[4]] [1]* его первая запись.

Если *Lst* список, то функция *length(Lst)* дает число (верхнего уровня) компоненты, которые он имеет. Компоненты списков можно также назвать, и в этом случае на компоненту можно сослаться или, давая имя компоненты как символьной строки вместо числа в двойных квадратных скобках, или, более удобно, давая выражение вида:

```
> name$component_name
```

для той же самой вещи.

Это - очень полезное соглашение, поскольку оно облегчает получать правильную компоненту, если забыли число.

Так в простом примере, данном выше:

Lst$name - то же самое как *Lst [[1]]* и является строкой "*Фред",*

Lst$wife - то же самое как *Lst [[2]]* и является строкой "*Мэри",*

Lst$child.ages [1] является тем же самым как *Lst [[4]] [1]* и является числом 4.

Дополнительно, можно также использовать имена компонентов списка в двойных квадратных скобках, то есть, *Lst[["имя"]]* то же самое как *Lst$name*. Это особенно полезно, когда имя извлекаемой компоненты сохраняется в другой переменной как в:

```
> x <- "name"; Lst[[x]]
```

Очень важно различить *Lst[[1]]* от *Lst[1].‘ [[...]]’* является оператором, используемым для выбора отдельного элемента, тогда как *‘[...]’* общий оператор преобразования в нижний индекс. Таким образом, прежний - *первый объект в списке Lst*, и если это именованный список, то имя не включено. Последний является *подсписком списка* ***Lst****, состоящим только из первой записи. Если это - именованный список, имена переданы подсписку.*

Имена компонентов могут быть сокращены урезанием до минимального числа букв, необходимых для их идентификации единственным образом. Таким образом, *Lst$coefficients* может минимально специфицироваться как *Lst$coe* и *Lst$covariance* как *Lst$cov.*

Вектор имен - фактически просто атрибут списка как любой другой и может быть обработан как таковой. Другим структурам помимо списков можно, конечно, так же дать атрибут имен.

6.2. Построение и изменение списков

Новые списки могут быть сформированы из существующих объектов функцией *list()*. Присвоение вида:

> Lst <- list(name_1=object_1, ..., name_m=object_m)

устанавливает список *Lst* из ***m*** компонентов, используя *object 1, . . . , object m* для компонентов и дает им имена как указано именами параметра (которые можно свободно выбрать). Если эти имена опущены, то компоненты только нумеруются. Используемые при формировании компоненты копируются, и новый список не влияет на оригиналы.

Списки, как любой преобразованный в нижний индекс объект, могут быть расширены указанием дополнительных компонент. Например:

> Lst[5] <- list(matrix=Mat)

6.2.1. Конкатенация списков

При конкатенации функцией *c()* с аргументами в виде списков получается результат в виде объекта с режимом списка, компоненты которого состоят из последовательно объединенного списка параметров.

> list.ABC <- c(list.A, list.B, list.C)

Вспомним, что с векторными объектами в качестве аргументов, функция конкатенации так же объединяла все параметры в отдельную векторную структуру. В этом случае все другие атрибуты, такие как атрибут *dim*, отбрасываются.

6.3. Фреймы данных

Фрейм данных – это список с классом "*data.frame*". Есть ограничения на списки, которые могут быть превращены в фреймы данных, а именно,

- Компоненты должны быть векторами (числовыми, символьными или логическими), факторами, числовыми матрицами, списками или другими фреймами данных.
- Матрицы, списки, и фреймы данных предоставляют столько переменных для нового фрейма данных, сколько у них имеется столбцов, элементов или переменных, соответственно.
- Числовые и логические векторы, а также факторы включаются как таковые, и по умолчанию символьные векторы преобразуются в факторы, уровни которых - единственное значение, появляющееся в векторе.
- Векторные структуры, появляющиеся как переменные фрейма данных, должны иметь одинаковую длину, а матричные структуры должны иметь одинаковый размер строки.

Для многих целей фреймы данных можно расценить как матрицу со столбцами возможно отличающихся режимом и атрибутом. Он может быть выведен на экран в матричной форме, и его строки и столбцы извлекаются с использованием соглашения индексации матрицы.

6.3.1. Создание фреймов данных

Объекты, удовлетворяющие ограничениям, установленным для столбцов (компонентов) фрейма данных, могут использоваться для формирования с использованием функции *data.frame*:

> accountants <- data.frame(home=statef, loot=incomes, shot=incomef)

Список, компоненты которого соответствуют ограничениям фрейма данных, может быть *преобразован* во фрейм данных, используя функцию *as.data.frame().*

Самый простой способ создания фрейма данных с нуля состоит в использовании функции *read.table()*при считывании всего фрейма данных из внешнего файла. Это обсуждается далее в Главе 7 [Чтение данных из файла].

6. 3. 2. attach() и detach()

Нотация $, такая как *accountants$home,* для компонентов списка не всегда очень удобна. Полезное средство должно делать компоненты списка или фрейма данных временно видимыми в качестве переменных под их именем компонентов без потребности явно каждый раз заключать имя списка в кавычки.

Функция *attach*(() берет 'базу данных', такую как список или фрейм данных, как свой параметр. Таким образом, предположим, что *lentils* - фрейм данных с тремя переменными *lentils$u, lentils$v, lentils$w.* Присоединение:

> *attach(lentils)*

разместит фрейм данных в пути поиска на позиции 2 и обеспечит отсутствие переменных ***u, v*** или ***w*** в позиции 1; ***u, v*** или ***w*** доступны как переменные из фрейма данных в их собственной позиции. В этом месте такое присвоение как:

> *u <- v+w*

не заменяет компонент ***u*** фрейма данных, а скорее маскирует его с другой переменной ***u*** в рабочем каталоге в позиции 1 на пути поиска. Чтобы произвести постоянное изменение непосредственно во фрейме данных, самый простой путь состоит в обращении еще раз к нотации ***$***:

> *lentils$u <- v+w*

Однако новое значение компонента ***u*** не видимо, пока фрейм данных не отсоединен и присоединен снова.

Чтобы отсоединить фрейм данных, используйте функцию:

> *detach()*

Более точно этот оператор отсоединяет от пути поиска объект в настоящий момент в позиции 2. Таким образом, в существующем контексте переменные ***u, v*** и ***w*** больше не были бы видимы, кроме, как в соответствии с нотацией списка как *lentils$u* и так далее. Объекты в позициях, больше чем 2 на пути поиска, могут быть отсоединены, давая их число для отсоединения, но намного более безопасно всегда использовать имя, например *detach(lentils) or detach("lentils").*

> **Замечание:** В ***R*** списки и фреймы данных можно присоединить только в позиции 2 или выше, и то, что присоединено, является копией исходного объекта. Можно изменить присоединенное значение через присвоение, но исходный список или фрейм данных неизменен.

6.3.3. Работа с фреймами данных

Полезное соглашение, которое позволяет удобно работать со многими различными проблемами в том же самом рабочем каталоге является:

- собрать все переменные для любой четко определенной и отдельной проблемы во фрейме данных под соответственно информативным именем;
- при работе с проблемой присоедините соответствующий фрейм данных в позиции 2, и используйте рабочий каталог на уровне 1 для операционных величин и временных переменных;

- прежде, чем оставить проблему, добавьте любые переменные, которые необходимо сохранить для будущей ссылки на фрейм данных, используя форму присвоения ***$***, а затем *detach().*
- наконец удалите все нежелательные переменные из рабочего каталога и сохраните его столь же чистым из оставшихся временных переменных насколько возможно.

Таким образом, довольно просто работать со многими проблемами в одном и том же каталоге, у каждой из которых есть переменные, названные ***x, y*** и **z**, например.

6.3.4. Присоединение произвольных списков

attach() является общей функцией, которая позволяет не только присоединять каталоги и фреймы данных к пути поиска, но также другие классы объектов. В частности любой объект с режимом *"список"* может быть присоединен следующим образом:

```
> attach(any.old.list)
```

Что-либо, что было присоединено, может быть отсоединено *detach(),* номером позиции или, предпочтительно, по имени.

6.3.5. Управление путем поиска

Функция *search* показывает текущий путь поиска, и таким образом является очень полезным способом отслеживания, какие фреймы данных и списки (и пакеты) были присоединены и отсоединены. Первоначально получаем путь:

```
> search()
[1] ".GlobalEnv" "Autoloads" "package:base"
```

где *.GlobalEnv* является рабочей областью.

После присоединения *lentils* мы получаем:

```
> search()
[1] ".GlobalEnv" "lentils" "Autoloads" "package:base"
> ls(2)
[1] "u" "v" "w"
```

и как видим ***ls*** (или ***object***) может использоваться для определения контента любой позиции на пути поиска.

Наконец, мы отсоединяем фрейм данных и подтверждаем, что он был удален из пути поиска.

```
> detach("lentils")
> search()
[1] ".GlobalEnv" "Autoloads" "package:base"
```

7. Чтение данных из файлов

Большие объекты данных обычно читают как значение из внешних файлов, а не вводиться во время сеанса ***R*** с клавиатуры. Средства ввода ***R*** просты, и их требования ясно ограничены и даже довольно не гибки. Есть ясное предположение разработчиков ***R,*** что Вы в состоянии изменить свои входные файлы, используя другие инструменты, такие как редакторы файлов или Perl для согласования с требованиями ***R***. Обычно это очень просто.

Если переменные подлежат хранению, главным образом, во фреймах данных, то строго предлагается, что весь фрейм данных может быть считан непосредственно функцией *read.table()*. Есть также более примитивная функция ввода *scan()*, которую можно вызвать непосредственно.

Для получения дополнительной информации по импорту и экспорту данных в ***R*** см. справочник Импорта/Экспорта данных ***R*** .

7.1. Функция read.table()

Чтобы непосредственно считать весь фрейм данных из внешнего файла обычно используется специальная форма.

- Первая строка файла должно иметь *имя* для каждой переменной во фрейме данных.
- Каждая дополнительная строка файла в качестве своего первого элемента имеет *метку строки* и значение для каждой переменной.

Если файл имеет на один элемент меньше в его первой строке, чем в его второй, то это расположение, как предполагают, находится в силе. Так, первые несколько строк файла, подлежащего чтению в качестве фрейма данных, могут выглядеть следующим образом.

Форма входного файла с именами и метками строк:

	Price	Floor	Area	Rooms	Age	Cent.heat	
01	52.00	111.0	830	5		6.2	no
02	54.75	128.0	710	5		7.5	no
03	57.50	101.0	1000	5		4.2	no
04	57.50	131.0	690	6		8.8	no
05	59.75	93.0	900	5		1.9	yes
	...						

По умолчанию числовые элементы (кроме меток строки) считываются как числовые переменные, и нечисловые переменные, такие как *Cent.heat* в примере, как факторы. Это может быть изменено в случае необходимости.

Затем функцию *read.table()* можно использовать для непосредственного считывания фрейм данных:

```
> HousePrice <-read.table ("houses.data")
```

Часто желательно опустить включенные метки строк и использовать метки по умолчанию. В этом случае в файле можно опустить столбец метки строки как в следующем.

Входная форма файла без меток строк:

Price	Floor	Area	Rooms	Age	Cent.heat
52.00	111.0	830	5	6.2	no
54.75	128.0	710	5	7.5	no
57.50	101.0	1000	5	4.2	no
57.50	131.0	690	6	8.8	no
59.75	93.0	900	5	1.9	yes

Затем фрейм данных может быть считан как:

```
> HousePrice <-read.table ("houses.data", header=TRUE)
```

где опция *header=TRUE* указывает, что первая строка - строка заголовков, что следует из формы файла, и что отсутствуют явные метки строки.

7.2. Функция scan()

Предположим, что векторы данных имеют равную длину и должны быть считаны параллельно. Далее предположим, что есть три вектора, первый имеет символьный режим, а оставшиеся два – числовой режим, и файл - '*input.dat*'. Первый шаг состоит в использовании *scan()* для чтения в три вектора как списка, следующим образом:

```
> inp <- scan("input.dat", list("",0,0))
```

Второй параметр - фиктивная структура списка, которая устанавливает режим этих трех считываемых векторов. Результатом, сохраненным в *inp*, является список, компоненты которого - эти три прочитанных вектора. Для разделения пунктов данных на три отдельных вектора используются присвоения, подобные следующем:

```
> label <- inp[[1]]; x <- inp[[2]]; y <- inp[[3]]
```

Для большего удобства фиктивный список мог иметь именованные компоненты для последующего использования при получении доступа к чтению векторов. Например:

```
> inp <- scan("input.dat", list(id="", x=0, y=0))
```

Если желательно получить доступ к переменным отдельно, то их можно либо повторно присвоить переменным в рабочем фрейме:

```
> label <- inp$id; x <- inp$x; y <- inp$y
```

либо список может быть присоединен в позиции 2 из пути поиска (см. Раздел 6.3.4 [Присоединение произвольных списков]).

Если второй параметр - отдельное значение, а не список, считывается отдельный вектор, все компоненты которого должны иметь одинаковый режим с фиктивным значением.

```
> X <- matrix(scan("light.dat", 0), ncol=5, byrow=TRUE)
```

Есть более тщательно продуманные доступные средства ввода, и они детализированы в справочниках.

7.3. Доступ к встроенным наборам данных

Приблизительно 100 наборов данных предоставлены ***R*** (в наборах данных пакета), и другие доступны в пакетах (включая рекомендуемые пакеты, предоставленные ***R***). Посмотреть список наборов данных, доступных в настоящий момент для использования, можно:

```
data()
```

Наборы данных, предоставленные ***R,*** доступны непосредственно по имени. Однако много пакетов все еще используют общее соглашение, в котором также использовалась *data* для загрузки наборов данных в ***R,*** например:

data (infert)

и это может все еще использоваться со стандартными пакетами (как в этом примере). В большинстве случаев будет загружен объект ***R*** того же самого имени. Однако в некоторых случаях загружается несколько объектов, так см. онлайновую справку для объекта, чтобы увидеть ожидаемое.

7.3.1. Загрузка данных из других пакетов *R*

К данным доступа из определенного пакета используйте параметр пакета, например:

data(package="rpart")

data(Puromycin, package="datasets")

Если пакет был присоединен библиотекой, ее наборы данных автоматически включены в поиск.

Внесенные пользователем пакеты могут иметь богатый источник наборов данных.

7.4. Редактирование данных

Когда вызван фрейм данных или матрица, редактирование переводит отдельную подобную электронную таблицу среды в рабочее состояние для редактирования. Это полезно для создания небольших изменений после считывания набора данных.

Команда:

> xnew <- edit (xold)

позволит редактировать свой набор данных *xold,* и по завершении измененный объект присвоен *xnew*. Если хотите изменить исходный набор данных *xold*, то самый простой путь состоит в использовании *fix(xold),* что эквивалентно *xold <-edit (xold).*

Использовать:

> xnew <- edit (data.frame)

и ввести новые данные через интерфейс электронной таблицы.

8. Распределение вероятности

8.1. *R* как ряд статистических таблиц

Одним из удобных использований ***R*** состоит в обеспечении исчерпывающего набора статистических таблиц. Функции позволяют оценить кумулятивную функцию распределения $P(X \leq x)$, функцию плотности вероятности и функцию квантиля (для данного ***q***, наименьший ***x*** такое, что $P(X \leq x) > q$), и имитировать распределения.

Распределение	Имя ***R***	Дополнительные параметры
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
chi-squared	chisq	df, ncp
exponential	exp	rate
F	f	df1, df2, ncp
gamma	gamma	shape, scale
geometric	geom	prob
hypergeometric	hyper	m, n, k
log-normal	lnorm	meanlog, sdlog
logistic	logis	location, scale
negative	binomial	nbinom size, prob
normal	norm	mean, sd
Poisson	pois	lambda
signed rank	signrank n	
Student's t	t	df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

Снабдите префиксом имя, данное здесь ***'d'*** для плотности, ***'p'*** для CDF, ***'q'*** для функции квантиля и ***'r'*** для имитации (случайные отклонения). Первый параметр - ***x*** для *dxxx*, ***q*** для *pxxx*, ***p*** для *qxxx* и ***n*** для *rxxx* (за исключением *rhyper*, *rsignrank* и *rwilcox* для которого это - *nn*). Не во всех случаях в настоящий момент доступен параметр не центрированности ***ncp***: см. онлайновую справку для деталей.

У всех функций *pxxx* и *qxxx* имеются логические параметры *lower.tail* и *log.p*, а *dxxx* имеет логарифм. Это позволяет, например, получать совокупную (или "интегрированную") функцию риска $H(t) = -\log(1 - F(t))$ путем:

- pxxx(t, ..., lower.tail = FALSE, log.p = TRUE)

или более точные логнормальные правдоподобия *(dxxx(..., log = TRUE))* непосредственно.

Кроме того, есть функции *ptukey* и *qtukey* для распределения стьюдентизированного диапазона выборок из нормального распределения, и *dmultinom*

и *rmultinom* для распределения многочлена. Дальнейшие распределения доступны во внесенных пакетах, особенно ***SuppDists.***

Вот некоторые примеры.

```
> ## 2-tailed p-value for t distribution
> 2*pt(-2.43, df = 13)
> ## upper 1% point for an F(2,7) distribution
> qf(0.01, 2, 7, lower.tail = FALSE)
```

8.2. Исследование распределения набора данных

Для данного (одномерного) набора данных можно исследовать его распределение большим количеством способов. Самое простое исследовать числа. Две немного отличающихся сводки даны *summary* и *fivenum* и показывает числа с помощью *stem* (рисунок “основа и лист”).

```
> attach(faithful)
> summary(eruptions)
Min. 1st Qu. Median Mean 3rd Qu. Max.
1.600 2.163 4.000 3.488 4.454 5.100
> fivenum(eruptions)
[1] 1.6000 2.1585 4.0000 4.4585 5.1000
> stem(eruptions)
```

Десятичная точка находится на 1 разряде слева от |:

```
16 | 070355555588
18 | 000022233333335577777777888822335777888
20 | 00002223378800035778
22 | 0002335578023578
24 | 00228
26 | 23
28 | 080
30 | 7
32 | 2337
34 | 250077
36 | 0000823577
38 | 2333335582225577
40 | 000000335778888800223355557777 8
42 | 033355557788002333335555777 78
44 | 022223355577800000000233333577 78888
46 | 0000233357700000023578
48 | 00000022335800333
50 | 0370
```

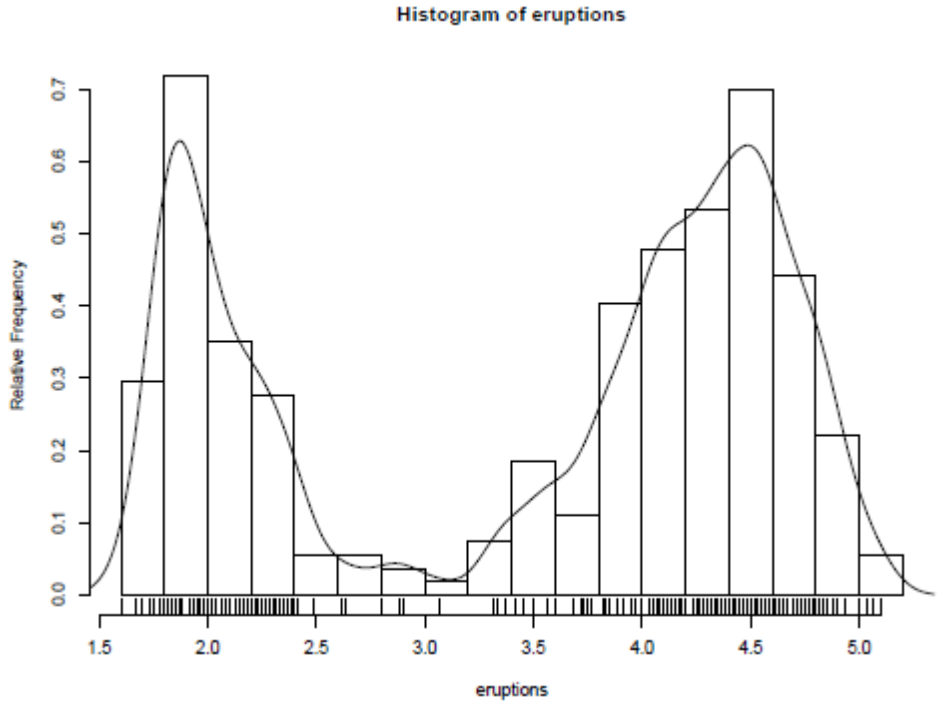
Рисунок основы-и-листа походит на гистограмму, и у ***R*** есть функция *hist* к гистограммам рисунка.

```
> hist(eruptions)
## make the bins smaller, make a plot of density
> hist(eruptions, seq(1.6, 5.2, 0.2), prob=TRUE)
```

> lines(density(eruptions, bw=0.1))

> rug(eruptions) # show the actual data points

Более изящные рисунки плотности могут быть сделаны с помощью *density*, и мы прибавили линию, продолженную плотностью в этом примере. Ширина полосы частот пропускной способности была выбрана эмпирическим, поскольку по умолчанию дает слишком много сглаживания (это обычно делает для "интересной" плотности). (Лучше автоматизированные методы выбора пропускной способности доступны, и в этой ширине полосы частот в качестве примера =, "SJ" дает хороший результат.)



Можно нарисовать эмпирическую кумулятивную функцию распределения путем использования функции *ecdf.*

> plot(ecdf(eruptions), do.points=FALSE, verticals=TRUE)

Это распределение очевидно далеко от любого стандартного распределения. Как о правом режиме, скажите прерывание дольше, чем 3 минуты? Давайте подгоним к нормальному распределению и наложим подогнанную CDF.

> long <- eruptions[eruptions > 3]

> plot(ecdf(long), do.points=FALSE, verticals=TRUE)

> x <- seq(3, 5.4, 0.01)

> lines(x, pnorm(x, mean=mean(long), sd=sqrt(var(long))), lty=3)

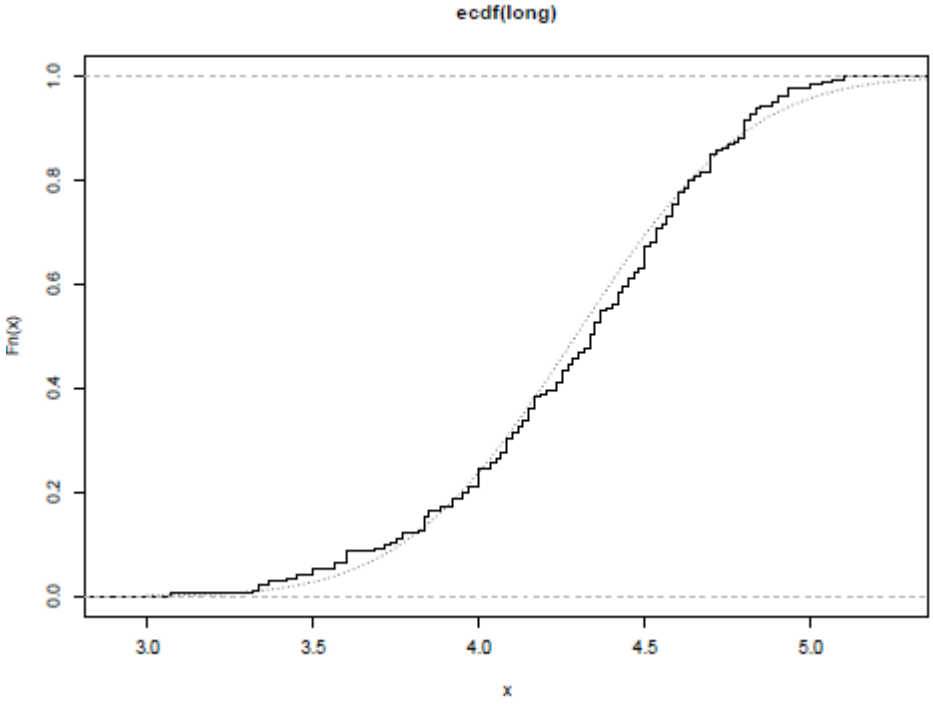
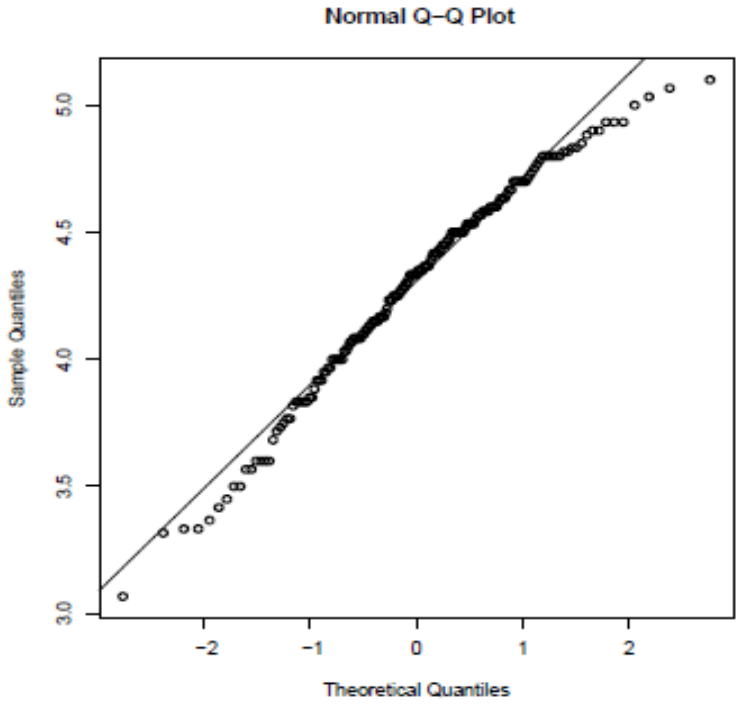


Рисунок квантиль-квантиль (Q-Q) может помочь нам проверить это более тщательно:

par(pty="s") # arrange for a square figure region

qqnorm(long); qqline(long)

которые показывают разумную подгонку, но более короткий правый хвост, чем можно было бы ожидать от нормального распределения. Давайте сравним его с некоторыми имитируемыми данными от ***t*** распределения:



x <- rt(250, df = 5)

qqnorm(x); qqline(x)

который обычно (если это будет случайная выборка) покажет более длинные хвосты, чем ожидается для нормального распределения. Мы можем сделать рисунок Q-Q против распределения генерации путем:

```
qqplot(qt(ppoints(250), df = 5), x, xlab = "Q-Q plot for t dsn")
qqline(x)
```

Наконец, можно провести более формальный тест согласования с нормальным распределением (или нет) ***R*** предоставляет тест Шапиро_Уилка:

```
> shapiro.test(long)
Shapiro-Wilk normality test
data: long
W = 0.9793, p-value = 0.01052
```

И тест Колмогора-Смирнова:

```
> ks.test(long, "pnorm", mean = mean(long), sd = sqrt(var(long)))
One-sample Kolmogorov-Smirnov test
data: long
D = 0.0661, p-value = 0.4284
alternative hypothesis: two.sided
```

(Заметим, что теория распределения не допустима здесь, поскольку мы оценили параметры нормального распределения от той же самой выборки.)

8.3. Тесты на одной и двух выборках

До сих пор мы сравнили отдельную выборку с нормальным распределением. В более общем сравнивают параметры двух выборок. Заметим, что в ***R,*** все "классические" тесты, включая используемых ниже, находятся в обычно загружаемых статистиках пакета.

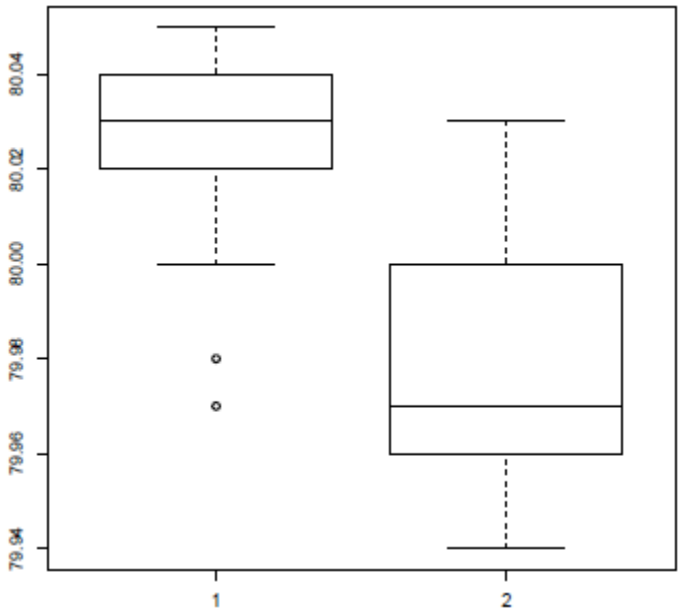
Рассмотрите следующие наборы данных на скрытом тепле плавления льда *(cal/gm)* от Райса (1995, p.490).

```
Method A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02
Method B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
```

Boxplots обеспечивают простое графическое сравнение двух выборок.

```
A <- scan()
79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97
80.05 80.03 80.02 80.00 80.02
B <- scan()
80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97
boxplot(A, B)
```

что указывает, что первая группа склонна давать более высокие результаты, чем вторая.



Чтобы проверить на равенство средних этих двух примеров, мы можем использовать непарный t-тест.

```
> t.test(A, B)
        Welch Two Sample t-test
data: A and B
t = 3.2499, df = 12.027, p-value = 0.00694
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
    0.01385526 0.07018320
sample estimates:
mean of x mean of y
    80.02077 79.97875
```

который действительно указывает на значительную разницу, предполагая нормальность. По умолчанию функция ***R*** не предполагает равенство дисперсий в двух выборках (в отличие от подобной функции в *S-Plus t.test*). Мы можем использовать тест ***F***, чтобы проверить на равенство в дисперсиях, при условии, что две выборки из нормальных совокупностей.

```
> var.test(A, B)
        F test to compare two variances
data: A and B
F = 0.5837, num df = 12, denom df = 7, p-value = 0.3938
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
    0.1251097 2.1052687
sample estimates:
ratio of variances
                0.5837405
```

который не приводит доказательства значительной разницы, и таким образом, мы можем использовать классический t-тест, который предполагает равенство дисперсий.

```
> t.test(A, B, var.equal=TRUE)
Two Sample t-test
data: A and B
t = 3.4722, df = 19, p-value = 0.002551
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
     0.01669058          0.06734788
sample estimates:
mean of x mean of y
     80.02077          79.97875
```

Все эти тесты предполагают нормальность двух выборок. Wilcoxon с двумя выборками (или Манн - Уитни) проверяют, только предполагает общее непрерывное распределение под нулевой гипотезой.

```
> wilcox.test(A, B)
Wilcoxon rank sum test with continuity correction
data: A and B
W = 89, p-value = 0.007497
alternative hypothesis: true location shift is not equal to 0
Warning message:
Cannot compute exact p-value with ties in: wilcox.test(A, B)
```

Предупреждение: существует несколько связей в каждой выборке, что предполагает строго, что эти данные от дискретного распределения (вероятно, из-за округления).

Есть несколько способов сравнить графически две выборки. Мы уже видели пару *boxplots*. Следующий пример:

```
> plot(ecdf(A), do.points=FALSE, verticals=TRUE, xlim=range(A, B))
> plot(ecdf(B), do.points=FALSE, verticals=TRUE, add=TRUE)
```

будет показано два эмпирических CDF, и *qqplot* выполнит рисунок Q-Q двух выборок. Тест Колмогороваа-Смирнова имеет максимальное вертикальное расстояние между двумя *ecdf*, предполагая общее непрерывное распределение:

```
> ks.test(A, B)
Two-sample Kolmogorov-Smirnov test
data: A and B
D = 0.5962, p-value = 0.05919
alternative hypothesis: two-sided
Warning message:
cannot compute correct p-values with ties in: ks.test(A, B)
```

9. Группировка, циклы и условное выполнение

9.1. Группирующие выражения

R - язык выражения в том смысле, что его единственный тип команды - функция или выражение, которое возвращает результат. Даже присвоение - выражение, результат которого присвоенное значение, и это может использоваться везде, где любое выражение может использоваться; в особенности при множественных присвоениях.

Команды могут группироваться в фигурных скобках, *{expr_1;...; expr_m},* когда значение группы - результат последнего выражения в оцененной группе. Так как такая группа - также выражение, то она может быть, например, включена в круглые скобки и использоваться как часть еще большего выражения и так далее.

9.2. Проверка утверждения

9.2.1. Условное выполнение: операторы if

В языке доступны условные конструкции в виде:

> if (expr_1) expr_2 else expr_3

где *expr 1* оценивается к единственному логическому значению, и результат всего выражения тогда очевиден.

Операторы "короткого цикла" ***&&*** и || часто используются в качестве части условия в операторе ***if***. Примем во внимание, что ***&*** и | применяются поэлементно к векторам, ***&&*** и || применяются к векторам длины один, и оценивают лишь их второй параметр в случае необходимости.

Есть векторная версия конструкция *if/else*, функция *ifelse.* Имеется форма *ifelse(condition, a, b*), которая возвращает вектор длины его самого длинного параметра, с элементами ***a[i],*** если *condition[i]* является истиной, иначе ***b[i].***

9.2.2. Повторное выполнение: for, loops, repeat и while

Также имеется конструкция цикла ***for***, у которой есть форма:

> for (name in expr_1) expr_2

где *name* - переменная цикла. *expr_1* является векторным выражением (часто последовательность как 1:20), и *expr_2* часто является сгруппированным выражением со своими подвыражениями, записанными с точки зрения фиктивного имени. *expr_2* неоднократно оценивается, поскольку *name* имеет значение в пределах векторного результата *expr_1*.

Как пример, предположим, что *ind* - вектор индикаторов класса, и мы хотим произвести отдельные рисунки ***y*** против ***x*** в пределах классов. Одна возможность здесь состоит в том, чтобы использовать *coplot(),* который произведет массив рисунков, соответствующих каждому уровню фактора. Другой способ сделать это - поместить все рисунки в один вывод следующими операторами:

```
> xc <- split(x, ind)
> yc <- split(y, ind)
> for (i in 1:length(yc)) {
plot(xc[[i]], yc[[i]])
abline(lsfit(xc[[i]], yc[[i]]))
}
```

Заметим, что функция *split()*, которая производит список векторов, полученных разделением большого вектора согласно классам, указанным фактором. Это полезная

функция, главным образом используемая в соединении с *boxplots*. См. справочное средство для получения дальнейшей информации.

Предупреждение: циклы *for()* используются в коде ***R*** намного менее часто, чем на языках компилирующего типа. Код, который получает ‘целый объект’ представление ‘целый объект’, вероятно, будет и более четким и быстрее в ***R*** .

Другие средства цикличного выполнения включают оператор:

```
> repeat expr
```

и оператор

```
> while (condition) expr
```

Оператор *break* может использоваться для завершения любого цикла, возможно неправильно. Это - единственный способ завершить зацикливание.

Оператор *next* может использоваться, чтобы прекратить один определенный цикл и перейти к "следующему".

Управляющие утверждения чаще всего используются в соединении с функциями, которые обсуждаются в Главе 10 [Написание собственных функций], и где имеется больше примеров.

10. Написание собственных функций

Из неформальных примеров видно, что язык ***R*** позволяет пользователю создавать объекты режима *функции*. Существуют реальные функции ***R,*** которые хранятся в специальной внутренней форме и могут использоваться в дальнейших выражениях и так далее. В процессе язык чрезвычайно усилился в мощности, удобстве и элегантности, и обучение написанию полезных функций являются одним из основных способов сделать использование ***R*** удобным и производительным.

Нужно подчеркнуть, что большинство функций, предоставленных как часть системы ***R,*** таких как *mean(), var(), postscript()* и так далее, сами написаны на ***R*** и, таким образом, существенно не отличается от написанных пользователем функций.

Функция определена присвоением вида:

```
> name <- function(arg_1, arg_2, ...) expression
```

expression это выражение ***R,*** (обычное групповое выражение), которое использует параметры *arg_i* для вычисления значения. Значение выражения – это значение, возвращаемое функцией.

Обычно в этом случае вызов функции затем обычно берет имя формулы *name(expr_1, expr_2...)* и может выполняться везде, где допустим вызов функции.

10.1. Простые примеры

Как первый пример, рассмотрим функцию, вычисляющую t-статистики двух выборок с показом “всех шагов”. Это - искусственный пример, конечно, так как есть другие, более простые способы достижения той же самой цели.

Функция определена следующим образом:

```
> twosam <- function(y1, y2) {
n1 <- length(y1); n2 <- length(y2)
yb1 <- mean(y1); yb2 <- mean(y2)
s1 <- var(y1); s2 <- var(y2)
s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
tst
}
```

С этой определенной функцией, можно выполнить t-тест для двух выборок, используя такой вызов:

```
> tstat <-twosam (data$male, data$female); tstat
```

В качестве второго примера рассмотрим функцию, непосредственно эмулирующую команду наклонной черты влево Matlab, которая возвращает коэффициенты ортогональной проекции вектора ***y*** на пространство столбца матрицы ***X.*** Это обычно называют оценкой коэффициентов регрессии методом наименьших квадратов. Это обычно делалось бы функцией ***qr()***; однако это иногда немного мудрено, чтобы использовать непосредственно, и проще иметь простую функцию как ниже для безопасного использования.

Таким образом, учитывая ***n-на-1*** вектор ***y*** и ***n-на-p*** матрицу ***X,*** то ***X y*** определены как $(X^TX)^-X^Ty$, где $(X^TX)^-$ является обобщенной инверсией ***X’X.***

```
> bslash <- function(X, y) {
X <- qr(X)
```

qr.coef(X, y)

}

После создания объекта, он может использоваться в таких операторах как:

> regcoeff <-bslash (Xmat, yvar)

и так далее.

Классическая функция ***R*** *lsfit()* делает это задание лучше. В свою очередь используются функции *qr()* и *qr.coef()* немного парадоксальным способом для выполнения своей части вычислений. Следовательно, существует, вероятно, некоторый смысл в простом выделении только этой части для использования функции в частом употреблении. Если так, то можно сделать это матричным бинарным оператором для более удобного использования.

10.2. Определение новых бинарных операторов

Если дать функции *bslash()* другое имя, а именно одной из форм:

%anything%

то это, возможно, использовалось бы в качестве *бинарного оператора* в выражениях, а не в форме функции. Предположим, например, мы выбираем ! для внутреннего символа. Определение функции тогда бы начиналось как:

> "%!%" <- function(X, y) { ... }

Заметим использование двойных кавычек. Затем функцию можно использовать как *X%!%y*. Символ самой наклонной черты влево не удобный выбор, поскольку он представляет специальные проблемы в этом контексте. Оператор умножения матриц ***%*%***, и внешний оператор матрицы произведения ***%o%*** являются другими примерами бинарных операторов, определенных таким образом.

10.3. Именованные параметры и умолчания

Как сначала отмечено в Разделе 2.3 [Генерация последовательности Коши], если параметры вызванным функциям переданы в форме *“name=object”*, то их можно передать в любом порядке. Кроме того, последовательность параметров может начинаться без имен, в позиционной форме, и указывать параметры, передаваемые по имени после позиционных параметров.

Таким образом, если существует функция *fun1*, определенная как:

> fun1 <- function(data, data.frame, graph, limit) {

[тело функции опущено]

}

то функция может быть вызвана несколькими эквивалентными способами, например:

> ans <- fun1(d, df, TRUE, 20)

> ans <- fun1(d, df, graph=TRUE, limit=20)

> ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)

Во многих случаях параметрам можно дать обычно соответствующие значения по умолчанию, тогда они могут быть опущены в целом от вызова, когда соответствуют умолчанию. Например, если *fun1* была определена как:

> fun1 <- function(data, data.frame, graph=TRUE, limit=20) { ... }

то можно было вызвать как:

> ans <- fun1(d, df)

что эквивалентно трем предыдущим случаям или как:

> ans <- fun1(d, df, limit=10)

в котором изменяется один из параметров на умолчания.

Важно отметить, что по умолчанию могут быть произвольными выражениями, даже вовлеченные другими аргументами той же самой функции; они не ограничены быть константами как здесь в нашем простом примере.

10.4. Параметр '...'

Другое частое требование состоит в предоставлении одной функции передавать установки аргументов другой. Например, много графических функций используют функцию *par(),* и функции подобную *plot(),* позволяя пользователю передавать графические параметры *par()*, чтобы управлять графическим выводом. См. Раздел 12.4.1 [Par() функция], для большего количества деталей о функции par(). Это может быть сделано включением дополнительного параметра функции буквально '...', которая может затем может быть передана. Пример схемы дан ниже.

```
fun1 <- function(data, data.frame, graph=TRUE, limit=20, ...) {
[пропущенный оператор]
if (graph)
    par(pch="*", ...)
[дальнейшие пропуски]
}
```

Реже функции необходимо сослаться на компонент '…'. Выражение *list(….)* оценивает все такие параметры и возвращает их в именованном списке, хотя …1, …2, и т.д. оценивает их один раз с возвращением "…n' для *n-го* не сопоставленного параметра.

10.5 Присвоения в пределах функций

Заметим, что любые обычные присвоения, сделанные в пределах функции, являются локальными и временными и теряются после выхода из функции. Таким образом, присвоение *X <-qr (X)* не влияет на значение аргумента в вызывающей программе.

Чтобы понять полностью правила, управляющие контекстом присвоений ***R,*** читателю необходимо ознакомится с понятием вычисления фрейма. Это несколько углубленное знание, хотя вряд ли сложное, однако тема в дальнейшем не будет здесь рассматриваться.

Если требуются глобальные и постоянные присвоения в пределах функции, то может быть использован или оператор "суперприсвоения" <<-, или функция *assign().* Подробнее смотри документацию. Пользователи S-Plus должны знать, что <<- имеет отличающуюся семантику в ***R*** . Эти вопросы обсуждаются далее в Разделе 10.7 [Контекст].

10.6. Более сложные примеры

10.6.1. Фактор эффективности при проектировании блоков

Более полный неинтересный пример функции, рассматривающий поиск эффективности при проектировании блоков. Некоторые аспекты этой проблемы были уже обсуждены в Разделе 5.3 [Индексные матрицы].

Блочная конструкция определена двумя факторами, скажем блоки (***b*** уровни) и варианты (***v*** уровни). Если ***R*** и ***K*** являются ***v-на-v*** и ***b-на-b*** репликациями и матрицами размера блока, соответственно, и ***N*** является ***b-на-v*** инцидентной матрицей, то коэффициенты полезного действия определены как собственные значения матрицы:

$E = I_v - R^{-1/2}N^TK^{-1}NR^{-1/2} = I_v - A^T A;$

где $A=K^{-1/2}NR^{-1/2}$. Один из путей написания функции дан ниже:

```
> bdeff <- function(blocks, varieties) {
      blocks <- as.factor(blocks)          # minor safety move
      b <- length(levels(blocks))
      varieties <- as.factor(varieties)    # minor safety move
      v <- length(levels(varieties))
      K <- as.vector(table(blocks))        # remove dim attr
      R <- as.vector(table(varieties))     # remove dim attr
      N <- table(blocks, varieties)
      A <- 1/sqrt(K) * N * rep(1/sqrt(R), rep(b, v))
      sv <- svd(A)
      list(eff=1 - sv$d^2, blockcv=sv$u, varietycv=sv$v)
}
```

В цифровой форме немного лучше работать с сингулярным разложением над этим случаем, а не подпрограммами собственного значения.

Результат функции - список, дающий не только эффективные факторы в качестве первого компонента, но также и канонические блочные и вариационные контрасты, так как иногда они дают дополнительную полезную качественную информацию.

10.6.2. Отбрасывание всех имен при печатании массива

Для целей печати больших матриц или массивов часто полезно напечатать их в форме плотного блока без имен массива или чисел. Удаление атрибута *dimnames* не даст нужного эффекта, напротив массиву нужно дать атрибут *dimnames*, состоящий из пустых строк. Например, чтобы напечатать матрицу ***X:***

```
> temp <- X
> dimnames(temp) <- list(rep("", nrow(X)), rep("", ncol(X)))
> temp; rm(temp)
```

что можно сделать более удобно, используя функцию *no.dimnames()*, показанную ниже, как "обертку" для достижения того же результата. Она также иллюстрирует, насколько короткими и полезными могут быть функции пользователя.

```
no.dimnames <- function(a) {
## Remove all dimension names from an array for compact printing.
d <- list()
l <- 0
for(i in dim(a)) {
d[[l <- l + 1]] <- rep("", i)
}
dimnames(a) <- d
a
}
```

С таким определением функции массив может быть напечатан в плотном формате путем использования:

```
> no.dimnames(X)
```

Это полезно для больших целочисленных массивов, в которых образец представляет реальный интерес, а не значение.

10.6.3. Рекурсивное числовое интегрирование

Функции могут быть рекурсивными, и могут самостоятельно определить функции в пределах себя. Заметим, однако, что такие функции, или действительно переменные, не наследованы вызванными функциями в более высоких фреймах оценки, как если они были бы на пути поиска.

Пример ниже показывает наивный способ выполнить одномерное числовое интегрирование. Подынтегральное выражение оценено в конечных точках диапазона и в середине. Если в целом правило трапеции дает ответ достаточно близкий для обеих частей, то последний возвращается в качестве значения. Иначе процесс рекурсивно применен к каждой части. Результат - адаптивный интеграционный процесс, который концентрирует функциональные оценки в областях, где подынтегральное выражение наиболее удалено от линейного. Есть, однако, тяжелые издержки, и функция способна конкурировать с другими алгоритмами, когда подынтегральное выражение является и гладким, и очень трудным для оценки.

Пример дан частично в качестве небольшой головоломки в ***R*** программировании.

```
area <- function(f, a, b, eps = 1.0e-06, lim = 10) {
    fun1 <- function(f, a, b, fa, fb, a0, eps, lim, fun) {
        ## function 'fun1' is only visible inside 'area'
        d <- (a + b)/2
        h <- (b - a)/4
        fd <- f(d)
        a1 <- h * (fa + fd)
        a2 <- h * (fd + fb)
        if(abs(a0 - a1 - a2) < eps || lim == 0)
            return(a1 + a2)
        else {
            return(fun(f, a, d, fa, fd, a1, eps, lim - 1, fun) +
            fun(f, d, b, fd, fb, a2, eps, lim - 1, fun))
        }
    }
fa <- f(a)
fb <- f(b)
a0 <- ((fa + fb) * (b - a))/2
fun1(f, a, b, fa, fb, a0, eps, lim, fun1)
```

10.7. Область действия

Обсуждение в этом разделе является несколько более схематичным, чем в других частях этого документа. Однако детализируется одно из существенных различий между S-Plus и ***R*** .

Символы в теле функции могут быть разделены на три класса: формальные параметры, локальные переменные и свободные переменные. Формальные параметры функции это те, которые возникли в списке параметров функции. Их значение

определяется процессом связывания фактических аргументов функции с формальными аргументами. Локальные переменные это те, значение которых определено оценкой выражений в теле функций. Переменные, которые не являются формальными параметрами или локальными переменными, называют свободными переменными. Свободные переменные становятся локальными переменными, если им они присваиваются. Рассмотрим следующее определение функции.

```
f <- function(x) {
y <- 2*x
print(x)
print(y)
print(z)
}
```

В этой функции ***x*** - формальный параметр, ***y*** - локальная переменная и ***z*** - свободная переменная.

В ***R*** связывание свободной переменной разрешается сначала путем поиска в области, в которой создавалась функция. Это называют лексическим контекстом. Сначала определим функцию *cube*.

```
cube <- function(n) {
sq <- function() n*n
n*sq()
}
```

Переменная ***n*** в функции *sq* не является аргументом этой функции. Поэтому - это свободная переменная, и следует использовать правила области действия, чтобы установить значение, которое должно быть ей назначено. Согласно статической области действия (S-Plus) значение определяется глобальной переменной с именем ***n***. В области действия (***R***) она является параметром функции *cube*, так как это активное связывание переменной ***n*** в момент определения функция *sq*. Разность между оценкой в ***R*** и оценкой в S-Plus в том, что S-Plus ищет глобальную переменную с именем ***n***, а ***R*** сначала ищет переменную с именем ***n*** в окружении, созданном после определения *cube*.

```
## сначала вычислим в S
S> cube(2)
Error in sq(): Object "n" not found
Dumped
S> n <- 3
S> cube(2)
[1] 18
## затем та же функция, вычисленная в R
R > cube(2)
[1] 8
```

Также лексический контекст может использоваться для *изменения состояния* функциям. В следующем примере покажем, как можно использовать ***R*** для имитации банковского счета. У функционирующего банковского счета должны быть баланс или итог, функция для вывода средств, функция для формирования депозитов и функция определения текущего баланса. Это достигается путем создания трех функций в внутри счета (*account)*, а затем возвращения списка, содержащего их. Когда счет (*account)* заводится, то он принимает числовой аргумент *total* и возвращает список, содержащий

три функции. Поскольку эти функции определены в среде, которая содержит *total*, у них будет доступ к ее значению.

Специальный оператор присваивания `<<-` используется для изменения значения, связанного с *total*. Этот оператор делает обратный проход в окружающие среды, которые содержит символ *total,* и когда он находит такую среду, то заменяет значение в той среде значением с правой стороны. Если среда глобального или верхнего уровня достигнута, не находя символа *total,* то создается переменная и там присваивается. Для большинства пользователей <<- создает глобальную переменную и присваивает ей значение правой стороны. Лишь когда <<- использовался в функции, которая вернула в качестве значения другой функции, то возникает специальный режим, описанный здесь.

```
open.account <- function(total) {
    list(
        deposit = function(amount) {
            if(amount <= 0)
                stop("Deposits must be positive!\n")
            total <<- total + amount
            cat(amount, "deposited. Your balance is", total, "\n\n")
        },
        withdraw = function(amount) {
            if(amount > total)
                stop("You don't have that much money!\n")
            total <<- total - amount
            cat(amount, "withdrawn. Your balance is", total, "\n\n")
        },
        balance = function() {
            cat("Your balance is", total, "\n\n")
        }
    )
}
ross <- open.account(100)
robert <- open.account(200)

ross$withdraw(30)
ross$balance()
robert$balance()

ross$deposit(50)
ross$balance()
ross$withdraw(500)
```

10.8. Настройка окружения

Пользователи могут настроить свое окружение несколькими различными способами. существует системный файл, и у каждого каталога может быть свой

собственный специальный файл инициализации. Наконец, можно использовать специальные функции *.First* и *.Last*.

Расположение системного файла инициализации берется из значения переменной окружения *R_PROFILE*. Если эта переменная сброшена, то используется файл *'Rprofile.site'* в подкаталоге *'etc'* домашней директории ***R***. Этот файл должен содержать команды, которые необходимо выполнять каждый раз при запуске ***R*** в Вашей системе. Второй, персональный файл профиля с именем *'.Rprofile*, может быть помещен в любой каталог. Если ***R*** вызван в этом каталоге, то файл будет считан из него. Этот файл дает отдельный пользовательский контроль над рабочей областью и учитывает различные процедуры запуска в различных рабочих каталогах. Если файл *'.Rprofile'* не найден в каталоге запуска, то ***R*** ищет файл *'.Rprofile'* в корневом каталоге пользователя и использует его (если он существует). Если установлена переменная среды *R_PROFILE_USER*, то файл, на который она указывает, используется вместо файлов *'.Rprofile'*.

У любой функции с именем *.First()* в любом из этих двух файлов профиля, либо в образе *'.RData'* имеется особый статус. Она автоматически выполняется в начале сеанса ***R*** и может использоваться для инициализации среды. Например, определение в примере ниже изменяет приглашение системы на ***$*** и устанавливает разные другие полезные вещи, которые затем могут считаться само собой разумеющимся в остальной части сеанса.

Таким образом, последовательность, в которой выполняются файлы, это *'Rprofile.site'*, *'Rprofile'*, *'.RData'*, а затем *.First()*. Определение в последующих файлах замаскирует определения в более ранних файлах.

```
> .First <- function() {
      options(prompt="$ ", continue="+\t")        # $ как приглашение
      options(digits=5, length=999)        # формат чисел и область печати
      x11()                                       # графический драйвер
      par(pch = "+")                              # plotting character
      source(file.path(Sys.getenv("HOME"), "R", "mystuff.R "))
                                                  # собственные функции
      library(MASS)                               # присоединить пакет
}
```

Аналогично, если определена функция *.Last()*, то она (обычно) выполняется в самом конце сеанса. Пример дан ниже.

```
> .Last <- function() {
      graphics.off()                    # небольшая мера безопасности.
      cat(paste(date(),"\nAdios\n"))    # Время обеда?
}
```

10.9. Классы, универсальные функции и объектно-ориентированное программирование

Класс объекта определяет, как он будет обработан *универсальными* функциями. Наоборот, универсальная функция выполняет действия и обработку своих аргументов в зависимости от *конкретного класса самого аргумента*. Если у аргумента отсутствует атрибут класса, или он имеет класс, не удовлетворяющий формату общей функции, всегда предусмотрено действие по умолчанию.

Поясним это примером. Механизм класса предлагает пользователю средство по проектированию и написанию универсальных функций для определенных целей. Среди ряда универсальных функций имеется *plot()* для вывода на экран графических объектов, *summary()* для суммирования анализов различных типов, и *anova()* для сравнения статистических моделей.

Количество универсальных функций, которые могут обработать класс определенным образом, может быть довольно большим. Например, функции, которые могут быть применены к некоторым объектам класса *"data.frame",* включают:

```
[ [[<- any as.matrix
[<- mean plot summary
```

Текущий полный список может быть получен использованием функции *methods():*

```
> methods(class="data.frame")
```

Наоборот, число классов, обрабатываемых универсальной функцией, также может быть довольно большим. Например, у функции *plot()* есть метод по умолчанию и разновидности для объектов классов *"data.frame", "density", "factor"* и многое другое. Полный список может быть получен снова при использовании функции *method()*:

```
> methods(plot)
```

Для многих универсальных функций тело функции довольно короткое, например:

```
> coef
function (object, ...)
UseMethod("coef")
```

Присутствие *UseMethod* указывает, что это универсальная функция. Чтобы увидеть, какой метод доступен, следует использовать *method():*

```
> methods(coef)
[1] coef.aov*        coef.Arima*   coef.default*   coef.listof*
[5] coef.nls*        coef.summary.nls*
      Non-visible functions are asterisked
```

В этом примере существует шесть методов, ни один из них нельзя увидеть, набрав его имя. Можно прочесть их, введя:

```
> getAnywhere("coef.aov")
A single object matching 'coef.aov' was found
It was found in the following places
      registered S3 method for coef from namespace stats
      namespace:stats
with value
function (object, ...)
{
      z <- object$coef
      z[!is.na(z)]
}
> getS3method("coef", "aov")
function (object, ...)
```

```
{
    z <- object$coef
    z[!is.na(z)]
}
```

Читатель отсылается к «Определению языка R» для более полного обсуждения этого механизма.

11. Статистические модели в *R*

Этот раздел предполагает, что у читателя есть некоторые познания в статистической методологии, в особенности в регрессионном анализе и дисперсионном анализе. Позже сделаем некоторые более честолюбивые предположения, а именно, что что-то известно об общей линейной модели и нелинейной регрессии.

Требования для подгонки статистической модели достаточно хорошо определены для разработки универсального, применимого для широкого спектра задач инструментария.

R обеспечивает набор взаимосвязанных инструментов, который делает очень простой подгонку статистических моделей. Как упоминалось во введении, по умолчанию отображается минимальный набор результатов, и нужно запрашивать подробности при обращении к функциям вывода.

11.1 Определение статистических моделей; формулы

Шаблон для статистической модели - линейная регрессионная модель с независимыми, гомоскедастичными ошибками:

$$y_i = \sum_{j=0}^{p} \beta_j x_{ij} + e_i, \qquad e_i \sim \mathrm{NID}(0, \sigma^2), \qquad i = 1, \ldots, n$$

В матричном виде можно записать:

$$y = X\beta + e$$

где ***y*** - вектор отклика, ***X*** *матрица модели или матрица проекта* и имеет столбцы x_0; x_1 …. ; x_p определяющих переменных. Очень часто $\boldsymbol{x_0}$ будет столбцом, дающий параметр смещения.

Примеры

Прежде чем дать формальное определение, несколько примеров помогут составить общее представление.

Предположим, что ***y, x, x0, x1, x2***... числовые переменные, ***X*** матрица и ***A, B, C***... являются факторами. Ниже следующие формулы задают статистические модели, справа даны описания моделей.

y ~ x

y ~ 1 + x

Обе подразумевают одинаковую простую линейную регрессионную модель ***y*** на ***x***. У первой есть неявный параметр смещения, а у второй - явный.

y ~ 0 + x

y ~-1 + x

y ~ x - 1

Простая линейная регрессия ***y*** на ***x*** через источник (то есть, без параметра смещения).

log (y) ~ x1 + x2

Множественная регрессия преобразованной переменной *log(y)* на ***x1*** и ***x2*** (с неявным параметром смещения).

y ~ poly (x, 2)

y ~ 1 + x + I(x^2)

Параболическая регрессия ***y*** на ***x*** степени 2. Первая форма использует ортогональные полиномы, вторая использует явную степень, как основание.

y ~ X + poly (x, 2)

Множественная регрессия ***y*** с модельной матрицей, состоящей из матрицы ***X***, включая параметр полинома ***x*** степени 2.

y ~ A

Модель дисперсионного анализа одиночной классификации ***y*** с классами, определенными A.

y ~ A+ x

Модель ковариационного анализа одиночной классификации ***y*** с классами, определенными ***A***, и с ковариантом ***x***.

*y ~ A*B*

y ~ + B + A:B

y ~ B %in % A

y ~ A/B

Модель двух факторного дисперсионного анализа ***y*** по ***A*** и ***B***. Первые две специфицируют одинаковую кросс классификацию, а вторые две специфицируют одинаковую вложенную классификацию.

В абстрактных понятиях все четыре специфицируют одинаковое подмножество моделей.

y ~ (A+ B + C) ^2

*y ~ A*B*C - A:B:C*

Трех факторный эксперимент, но с моделью, содержащей основные эффекты и факторы попарного взаимодействия. Обе формулы специфицируют одинаковую модель.

*y ~ A * x*

y ~ A/x

y ~ A / (1 + x) - 1

Изолированные модели простой линейной регрессии ***y*** на ***x*** в пределах уровней заданных в ***A*** различными метками. В последнем виде производит четко столько вычислений различных отсекаемых отрезков и коэффициентов наклона, сколько имеется уровней ***A.***

*y ~ A*B + Error(C)*

Эксперимент с двумя факторами воздействия ***A*** и ***B,*** и стратифицированной ошибкой, определяемой фактором ***C***. Например, разделить отображение эксперимента на участки (и, следовательно, части рисунка), определяемые фактором ***C***.

Оператор **~** используется для определения формулы модели в ***R*** . Форма для простой линейной модели:

response ~ op_1 term_1 op_2 term_2 op_3 term_3 ...

где:

response - вектор или матрица (или оценка выражения к вектору или матрице), определяющая переменную (ые) отклика.

op_i - оператор, или “+” или “-“, подразумевая включение или исключение параметра в модели (первое является дополнительным).

term_i также является либо:

- векторным или матричным выражением, или 1, либо
- фактор, либо
- выражением формулы, состоящей из факторов, векторов или матриц, соединенных операторами формулы.
  Во всех случаях каждый параметр определяет набор столбцов либо для добавления к матрице модели, либо для удаления из матрицы модели. 1 устанавливается для столбца смещения и по умолчанию включена в матрицу модели, если явно не удалена.

Операторы формулы подобны нотации Уилкинсона и Роджерса, используемой такими программами как Glim и Genstat. Одно неизбежное изменение то, что оператор '.' становится ‘:’ так как точка является допустимым символом имени в ***R.***

В итоге ниже получена нотация (основано на Chambers & Hastie, 1992, p.29):

Y ~ M Y смоделирован как M.

M_1 + M_2 Включают M_1 и M_2.

M_1 - M_2 Включают M_1 и исключают параметр M_2.

M_1: M_2 Тензорное произведение M_1 и M_2. Если оба параметра - факторы, то фактор "подклассов".

M_1 %in % M_2
Подобно M_1:M_2, но с различным синтаксисом.

*M_1 * M_2* *M_1 + M_2 + M_1:M_2.*

M_1 / M_2 *M_1 + M_2 %in % M_1.*

M^n Все параметры в *M* вместе со "взаимодействиями" до порядка n

I(M) Изолированное ***M.*** Внутри ***M*** все операторы имеют свое обычное арифметическое значение, и этот параметр появляется в матрице модели.

Заметим, что в круглых скобках, которые обычно включают аргументы функции, у всех операторов есть свое нормальное арифметическое значение. Функция ***I()*** является зеркальным отображением, используемым для придания определенности параметрам в формулах модели, используя арифметические операторы.

В частности заметим, что формулы модели описывают столбцы матрицы модели, определение подразумевающихся параметров. Дело обстоит не так в других контекстах, например в определении нелинейных моделей.

11.1.1. Противопоставления

Мы нуждаемся, по крайней мере, в некоторой идее, как формулы модели описывают столбцы матрицы модели. Это просто для непрерывных переменных, поскольку каждая обеспечивает один столбец матрицы модели (и свободный член задает столбец из единиц, если он включен в модель).

Что относительно ***k***-уровневого фактора ***A***? Ответ отличается для неупорядоченных и упорядоченных факторов. Для неупорядоченных факторов ***k-1*** столбец генерируются для показатели второго, ..., ***k-го*** уровней фактора. (Таким образом, применяемая параметризация создает на каждом уровне такой же контраст

откликов, что и на первом.) Для упорядоченных факторов, ***k-1*** столбцы являются ортогональными полиномами по основанию ***1, ..., k***, исключая параметры константы.

Хотя ответ уже дан, но это еще не все. Во-первых, если смещение опущено в модели, которая содержит факторный параметр, первое, такой параметр закодирован в ***k*** столбцов, дающих индикаторы для всех уровней. Во-вторых, поведение в целом может быть изменено установкой опций для противоположностей. По умолчанию в ***R*** установлено:

options(contrasts = c("contr.treatment", "contr.poly"))

Главная причина этого упоминания состоит в том, что у ***R*** и ***S*** есть различия по умолчанию для неупорядоченных факторов, ***S*** использует противоположности Helmert. Так, если необходимо сравнить результаты с таковыми из учебника или статьи, которая использовала S-Plus, то следует установить:

options(contrasts = c("contr.helmert", "contr.poly"))

Это - преднамеренная разность, поскольку обработка противоположностей (по умолчанию ***R***) будет легче для понимания новичками.

Мы все еще не закончили, поскольку схема противопоставления для использования может быть установлена для каждого параметра в модели, используя противоположности функций и ***C.***

Мы еще не рассмотрели параметры взаимодействия: они генерируют произведения столбцов, представленных для компонентов этих параметров.

Хотя сложные особенности и сохраняют некоторую маргинальность, формулы модели в ***R*** будут обычно генерировать модели, которые ожидал бы опытный статистик, при условии, что принцип малых приращений сохранен. Подгонка, например, модели взаимодействия, не включающая соответствующих главных эффектов, в целом приведет к неожиданным результатам, и предназначена только для экспертов.

11.2. Линейные модели

Основная функция для подгонки обычным многоуровневым моделям является ***lm(),*** а усовершенствованный вариант вызова выглядит следующим образом:

> fitted.model <- lm(formula, data = data.frame)

Например:

> fm2 <- lm(y ~ x1 + x2, data = production)

будет соответствовать подгонке множественной регрессионной модели ***y*** на *x1* и *x2* (с неявным параметром смещения).

Важный (но технически дополнительный) параметр *data = production* указывает, что любые переменные, необходимые для создания модели, должны быть в первую очередь из *фрейма данных* ***production***. *Это не зависит от того, был ли фрейм данных* ***production*** *присоединен к пути поиска или нет.*

11.3. Универсальные функции для извлечения информации о модели

Значением *lm()* является подогнанный объект *model*; технически это список результатов класса "*lm*". В этом случае информация о подогнанной модели может быть выведена на экран, извлечена, графически изображена и так далее при использовании универсальных функций, которые относятся к объектам класса *"lm"*. Они включают:

add1	*deviance*	*formula*	*predict*	*step*
alias	*drop1*	*kappa*	*print*	*summary*

anova	*effects*	*labels*	*proj*	*vcov*
coef	*family*	*plot*	*residuals*	

Краткое описание наиболее часто используемых функций дано ниже.

anova(object_1, object_2)

Сравните подмодель с внешней моделью и произведите таблицу дисперсионного анализа.

coef(object)

Извлеките коэффициент регрессии (матрицу).

Длинная форма: *coefficients(object)*

deviance(object)

Сумма квадратов остатков, взвешенная если возможно.

formula(object)

Извлеките формулу модели.

plot(object)

Произведите четыре рисунка, показав остатки, подогнанное значение и некоторую диагностику.

predict(object, newdata=data.frame)

Предоставленному фрейму данных нужно было специфицировать переменные с теми же самыми метками как оригинал. Значение - вектор или матрица ожидаемых значений, соответствующих значениям определенных переменных в *data.frame.*

print(object)

Напечатайте краткую версию объекта. Чаще всего используется неявно.

residuals(object)

Извлеките (матрицу) остатков, взвешенных если возможно.

Краткая форма: *resid (объект).*

step(object)

Выберите подходящую модель, добавляя или отбрасывая параметры и сохраняя иерархии. Возвращается модель с наименьшим значением AIC (информационный критерий), обнаруженным в пошаговом поиске.

summary(object)

Напечатайте общую сводку результатов регрессионного анализа.

vcov(object)

Возвращает матрицу ковариации дисперсии основных параметров подогнанного объекта модели.

11.4. Дисперсионный анализ и сравнение модели

Функция подгонки модели *aov(формула, data=data.frame)* работает на самом простом уровне очень похожим способом как функция *lm()*, и большинство

универсальных функций, перечисленных в таблице в Разделе 11.3 [Универсальные функции для извлечения информацию о модели].

Нужно отметить, что дополнительно *aov()* позволяет анализировать модели с множественными слоями ошибок, такими как рисунками разделенных экспериментов, или сбалансированные неполноблочные планы с восстановлением межблочной информации. Формула модели:

response ~ mean.formula + Error(strata.formula)

указывает на многослойный эксперимент с ошибками слоев определенными *strata.formula*. В самом простом случае *strata.formula* - просто фактор, когда определяет два слоя эксперимента, а именно, между и внутри уровней фактора.

Например, со всеми определенными переменными факторами, формула модели выглядит следующим образом:

*> fm <- aov(yield ~ v + n*p*k + Error(farms/blocks), data=farm.data)*

и обычно используется для описания эксперимента со средней модели *v + n*p*k* и тремя слоями ошибок, а именно: "между farms", "внутри farms, между bloks" и "в рамках blocks".

11.4.1. Таблицы ANOVA

Заметим также, что анализ дисперсионных таблиц (или таблицы) выполняется для последовательности подогнанных моделей. Суммы квадратов показывают уменьшение суммы квадратов остатков в результате включения *конкретного* параметра в модель в *определенное* место последовательности. Следовательно, только для ортогональных экспериментов порядок включения будет несущественным.

Для многослойного эксперимента начинают процедуру проекцией отклика на слои ошибки, по порядку, и затем подгоняют модельное среднее для каждой проекции. Более подробную информацию см. Chambers & Hastie (1992).

Более гибкую альтернативу полной таблице ANOVA по умолчанию, для сравнения двух или более моделей можно непосредственно использовать функцию *anova()*.

> anova(fitted.model.1, fitted.model.2, ...)

В результате отображается таблица различий ANOVA между подогнанными моделями, при подгонке по порядку. Подогнанные модели сравниваются, конечно, как правило, в иерархической последовательности. Это не дает информацию, отличную от способа по умолчанию, а просто проще для понимания и контроля.

11.5. Обновление подогнанных моделей

Функция *update()* в значительной степени функция для удобства, которая подгоняет модель с отличиями от ранее подогнанной обычно только несколькими дополнительными или удаленными параметрами. Ее синтаксис:

> new.model <- update(old.model, new.formula)

В *new.formula* специальное имя, состоящее из точки '.', которая используется для обозначения “соответствующей части старой формулы модели”. Например,

> fm05 <- lm(y ~ x1 + x2 + x3 + x4 + x5, data = production)

> fm6 <- update(fm05, . ~ . + x6)

> smf6 <- update(fm6, sqrt(.) ~ .)

подгоняет множественную регрессию с пятью случайным переменным (по-видимому) из фрейма данных *production*, подгоняет дополнительную модель, включая шестой

регрессор, и подгоняет разновидность модели, где к отклику применили преобразование квадратного корня.

Заметим особенно, что, если *data* = параметр специфицируется в исходном вызове функции подгонки модели, то эта информация передана через подогнанный объект модели в *update()* и его союзники.

Имя '.' может также использоваться в других контекстах, но с немного отличающимся значением. Например

> fmfull <- lm(y ~ . , data = production)

подогнал бы модели откликом ***y*** и переменными регрессоров все другие переменные во фреймах данных *production.*

Другие функции для исследования дополнительных последовательностей моделей являются *add1(), drop1()* и *andstep()*. Имена их дают указание на цель их использования, но для полного изложения см. онлайновую справку.

11.6. Обобщенные линейные модели

Обобщенное линейное моделирование это разработка линейных моделей для учета чистым и простым способом, как не-нормальных распределений отклика, так и линеаризующих преобразований. Обобщенная линейная модель может быть описана в терминах следующей последовательности предположений:

- Есть отклик ***y*** и воздействующие переменные x_1, x_2 …., чье значение влияет на распределение отклика.
- Воздействующие переменные влияют на распределение ***y*** *только через единственную линейную функцию*. Эту линейную функцию называют *линейным предиктором*, и обычно записывают как:

  $$\eta = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p,$$

  следовательно, X_i не имеет никакого влияния на распределение ***y*** если и только если $\beta i = 0$.

- Распределение ***y*** имеет форму

  $$f_Y(y;\mu,\varphi) = \exp\left[\frac{A}{\varphi}\{y\lambda(\mu) - \gamma(\lambda(\mu))\} + \tau(y,\varphi)\right]$$

  то, где ϕ коэффициент масштабирования (возможно известный), и является постоянным для всех наблюдений, ***A*** представляет собой предшествующий вес, предположительно известный, но возможно меняющийся в зависимости от наблюдений, и μ является средним ***y***. Таким образом, предполагается, что распределение *y* определено его средним, а также, возможно, коэффициентом масштабирования.

- Среднее μ, гладкая обратимая функция линейного предиктора:

  $$\mu = m(\eta), \qquad \eta = m^{-1}(\mu) = \ell(\mu)$$

  и эта обратная функция $l(\mu)$ называется функцией ссылки.

Эти предположения достаточно неточны, чтобы охватить широкий класс моделей, полезных в статистической практике, но достаточно жесткие для разработки, по крайней мере, приблизительной объединенной методологии оценки и вывода. Читатель отсылается в любую из текущих ссылочных работ по предмету для полного изложения, например, McCullagh & Nelder (1989) или Добсон (1990).

11.6.1. Семейства

Класс обобщенных линейных моделей, обработанных средствами, предоставленными в ***R,*** включает *gaussian, binomial, poisson, inverse gaussian* и *gamma response* распределения, а также модели квазиправдоподобия, где распределение отклика явно не указывается. В последнем случае функция дисперсии должна специфицироваться как функция среднего, но в других случаях эта функция подразумевается распределением отклика.

Каждое распределение отклика допускает, что множество функций ссылки соединяет среднее с линейным предиктором. Автоматически доступные распределения показаны в следующей таблице:

Имя семейства	Функции связи
binomial	*logit, probit, log, cloglog*
gaussian	*identity, log, inverse*
Gamma	*identity, inverse, log*
inverse.gaussian	*1/mu^2, identity, inverse, log*
poisson	*identity, log, sqrt*
quasi	*logit, probit, cloglog, identity, inverse, log, 1/mu^2, sqrt*

Комбинация распределения отклика, функции ссылки и различные другие необходимые сведения для выполнения задач моделирования, называются семейством обобщенной линейной модели.

11.6.2. Функция glm()

Поскольку распределение отклика определяется воздействующими переменными посредством только одной линейной функции, тот же механизм, который был использован для линейных моделей, может быть использован для задания линейной части обобщенной модели. В семействе это задается различными способами.

Функция ***R*** для подгонки обобщенной линейной модели называется *glm()* и имеет вид:

```
> fitted.model <- glm(formula, family=family.generator, data=data.frame)
```

Единственная новая функция - ***family.generator***, которая является инструментом для описания семейства. Это - имя функции, которая генерирует список функций и выражений, которые вместе определяют и управляют моделью и процессом оценки. Хотя это может казаться немного усложненным на первый взгляд, использование довольно просто.

Имена стандартных, предоставленных генераторов семейства даны под "Фамилией" в таблице в Разделе 11.6.1 [семейства]. Где есть выбор ссылок, имя ссылки может также быть предоставлено фамилией в круглых скобках в качестве параметра. В случае квази семейства функция дисперсии может также специфицироваться таким образом.

Некоторые примеры ясно дают понять процесс.

Гауссовское семейство

Вызов такой как

```
> fm <- glm(y ~ x1 + x2, family = gaussian, data = sales)
```

достигает того же самого результата как

```
> fm <- lm(y ~ x1+x2, data=sales)
```

но намного менее эффективно. Обратим внимание, поскольку гауссовскому семейству автоматически не предоставляют выбор ссылок, то и не допускается никаких параметров. Если задача требует гауссовского семейства с нестандартной ссылкой, это может обычно достигаться через *квази* семейство, как увидим позже.

Биномиальное семейство

Рассмотрите небольшой искусственный пример от Silvey (1970).

На острове Kalythos в Эгейском море местные жители страдают от врожденной болезни глаз, симптомы которой становятся более заметны с возрастом. Выборки мужчин-островитянин различных возрастов были проверены на слепоту и результаты записаны. Эти данные показаны ниже:

Возраст:	*20*	*35*	*45*	*55*	*70*
Количество испытаний:	*50*	*50*	*50*	*50*	*50*
Количество слепых:	*6*	*17*	*26*	*37*	*44*

Проблема, по нашему мнению, заключается в том, чтобы подогнать как логит, так и пробит модели этих данных, и определить для каждой модели LD50, то есть возраст, в котором вероятность слепоты для мужского населения составляет 50%.

Если y это число слепых в возрасте x и n число проверенных, обе модели имеют вид y ~ B(n, F(β_0 + β_1 x)), где для пробит случая F(z) = Phi(z) является стандартной нормальной функцией распределения, а в логит случае (по умолчанию), F(z) = e^z/(1+e^z). В обоих случаях LD50 является LD50 = - beta_0/beta_1, то есть той точки, в которой аргумент функции распределения равен нулю.

Первый шаг - это ввод данных, в виде таблицы данных

```
> kalythos <- data.frame(x = c(20,35,45,55,70), n = rep(50,5), y = c(6,17,26,37,44))
```

Чтобы подогнать binomial модель с использованием *glm()* есть три варианта ввода отклика:

- Если отклик это вектор, то предполагается что в нем двоичные данные и он должен содержать только 0/1.
- Если отклик это двух-колоночная матрица предполагается, что первая колонка содержит число успешных исходов, а второй содержит число неудач.
- Если отклик является фактором, его первый уровень - воспринимается как "проигрыш" (0) и все другие уровни как "успех" (1).

Здесь нам нужно второе из этих соглашений, поэтому мы добавляем матрицу к нашей таблице данных:

```
> kalythos$Ymat <- cbind(kalythos$y, kalythos$n - kalythos$y)
```

Чтобы подогнать модели мы используем

```
> fmp <- glm(Ymat ~ x, family = binomial(link=probit), data = kalythos)
> fml <- glm(Ymat ~ x, family = binomial, data = kalythos)
```

Поскольку logit связь стоит по умолчанию, этот параметр может быть пропущен во втором вызове. Чтобы увидеть результаты каждой подгонки мы могли бы использовать

```
> summary(fmp)
> summary(fml)
```

Обе модели соответствуют хорошо (даже слишком). Чтобы найти LD50 оценки мы можем использовать простую функцию:

```
> ld50 <- function(b) -b[1]/b[2]
> ldp <- ld50(coef(fmp)); ldl <- ld50(coef(fml)); c(ldp, ldl)
```

Фактическая оценка по этим данным дает 43,663 лет и 43,601 лет соответственно.

Модели Пуассона

С семейством Пуассона ссылка по умолчанию - log, и практически главное использование этого семейства должно подогнать суррогатному Пуассону log - линейные модели к данным частоты, фактическое распределение которых часто - многочлен. Это - большая и важная тема, которую мы не будем затрагивать далее здесь. Это даже является главной частью использования негауссовских обобщенных моделей повсюду.

Иногда истинно данные Пуассона возникают практически, и в прошлом они часто анализировались как гауссовские данные или после преобразования log или квадратного корня. Как корректная альтернатива последнему, обобщенную модель Пуассона можно подогнать как в следующем примере:

```
> fmod <- glm(y ~ A + B + x, family = poisson(link=sqrt),
data = worm.counts)
```

Модели квазиправдоподобия

Для всех семейств дисперсия отклика будет зависеть от среднего и будет иметь масштабный коэффициент как множитель. Форма зависимости дисперсии на среднем - характеристика распределения отклика; например для распределения Пуассона VAR[y] = μ.

Для оценки квазиправдоподобия и вывода точное распределение отклика не специфицируется, а скорее только функция ссылки и форма функции дисперсии, поскольку они зависят от среднего. Хотя оценки квазиправдоподобия формально используют идентичные методы распределения Гаусса, это семейство обеспечивает способ подгонки гауссовских моделей нестандартными функциями ссылки или функциями дисперсии, случайно.

Например, рассмотрите подгонку нелинейной регрессии:

$$y = \theta_1 z_1/(z_2-\theta_2) + e$$

что можно записать альтернативным способом:

$$y = 1/(\beta_1 x_1 + \beta_{2x2}) + e$$

Предполагая соответствующий фрейм данных можно подогнать эту нелинейную регрессию как:

```
> nlfit <- glm(y ~ x1 + x2 - 1,
family = quasi(link=inverse, variance=constant),
data = biochem)
```

11.7. Нелинейные наименьшие квадраты и модели наибольшего правдоподобия

Определенного вида нелинейные модели можно подогнать Обобщенными Линейными Моделями (*glm*). Но в большинстве случаев следует подходить к проблеме подбора нелинейных кривых как к одной из нелинейных оптимизаций. Нелинейные подпрограммы оптимизации ***R*** - *optim(), nlm()* и *nlminb(),* которые обеспечивают (и превосходят) функциональность S-плюсовых *ms()* и *nlminb().* Ищется значение параметра, которое минимизирует некоторый индекс отклонения-от- цели, и делается

это путем многократного испытания различных значений параметров. В отличие от линейной регрессии, например, нет никакой гарантии, что процедура будет сходиться к удовлетворительным оценкам. Все методы требуют исходных предположений о начальном значении параметра, и сходимость может критически зависеть от качества начальных значений.

11.7.1. Наименьшие квадраты

Один из способов подгонки нелинейной модели состоит в минимизации суммы квадратных ошибок (SSE) или остатков. Этот метод имеет смысл, если наблюдаемые ошибки, возможно, правдоподобно имеют нормальное распределение.

Вот пример от Бэйтса & Ватт (1988). Данные:

```
> x <- c(0.02, 0.02, 0.06, 0.06, 0.11, 0.11, 0.22, 0.22, 0.56, 0.56,
       1.10, 1.10)
> y <- c(76, 47, 97, 107, 123, 139, 159, 152, 191, 201, 207, 200)
```

Критерий подгонки, подлежащий минимизации:

```
> fn <- function(p) sum((y - (p[1] * x)/(p[2] + x))^2)
```

Для подгонки необходимы первоначальные оценки параметров. Одним из способов найти начальные значения состоит в рисовании данных, угадывания некоторых значений параметра, и наложения кривой модели, используя это значение.

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 200 * xfit/(0.1 + xfit)
> lines(spline(xfit, yfit))
```

Можно сделать лучше, но эти стартовые значения 200 и 0.1 кажутся адекватными. Теперь сделаем подгонку:

```
> out <- nlm(fn, p = c(200, 0.1), hessian = TRUE)
```

После подгонки *out$minimum* равно SSE, и *out$estimate* является оценкой параметров методом наименьших квадратов. Чтобы получить оценку приблизительных стандартных ошибок (SE), мы делаем:

```
> out <- nlm(fn, p = c(200, 0.1), hessian = TRUE)
```

2 в строке выше представляют число параметров. 95%-ый доверительный интервал был бы оценкой параметра $\pm$ 1.96 SE. Мы можем наложить подбор методом наименьших квадратов на новый рисунок:

```
> plot(x, y)
> xfit <- seq(.02, 1.1, .05)
> yfit <- 212.68384222 * xfit/(0.06412146 + xfit)
> lines(spline(xfit, yfit))
```

Стандартный пакет *stats* предоставляют намного более обширные средства для подгонки нелинейных моделей наименьшими квадратами. Модель, которой мы только что подогнали, является моделью Michaelis-Menten, таким образом, можно использовать:

```
> df <- data.frame(x=x, y=y)
> fit <- nls(y ~ SSmicmen(x, Vm, K), df)
> fit
Nonlinear regression model
```

```
      model:      y ~ SSmicmen(x, Vm, K)
      data:       df
      Vm          K
212.68370711      0.06412123
residual sum-of-squares: 1195.449
> summary(fit)
Formula: y ~ SSmicmen(x, Vm, K)
Parameters:
   Estimate     Std. Error   t value     Pr(>|t|)
Vm 2.127e+02    6.947e+00    30.615      3.24e-11
K  6.412e-02    8.281e-03    7.743       1.57e-05
Residual standard error: 10.93 on 10 degrees of freedom
Correlation of Parameter Estimates:
     Vm
K 0.7651
```

11.7.2. Метод максимального правдоподобия

Максимальное правдоподобие - метод подгонки нелинейной модели, который применяется, даже если ошибки не нормальны. Метод находит значение параметра, которое максимизирует логарифмическое правдоподобия, или что эквивалентно, которое минимизируют отрицательное логарифмическое правдоподобие. Вот пример от Добсона (1990), стр 108-111. Этот пример подгоняет логистической модели к данным отклика дозы, которые ясно могли также быть подогнаны *glm()*. Данные таковы:

```
> x <- c(1.6907, 1.7242, 1.7552, 1.7842, 1.8113,
1.8369, 1.8610, 1.8839)
> y <- c( 6, 13, 18, 28, 52, 53, 61, 60)
> n <- c(59, 60, 62, 56, 63, 59, 62, 60)
```

Отрицательная логарифмическая функция правдоподобия, подлежащая минимизации, равна:

```
> fn <- function(p)
sum( - (y*(p[1]+p[2]*x) - n*log(1+exp(p[1]+p[2]*x))
+ log(choose(n, y)) ))
```

Возьмем разумное начальное значение для подгонки:

```
> out <- nlm(fn, p = c(-50,20), hessian = TRUE)
```

После подгонки *out$minimum* равно отрицательному логарифмическому правдоподобию, и *out$estimate* является наибольшим правдоподобием оценки параметров. Чтобы получить приблизительную оценку SE, выполним:

```
> sqrt(diag(solve(out$hessian)))
```

95% доверительный интервал получим при SE $\pm$ 1.96

11.8. Некоторые нестандартные модели

Завершается эта глава кратким упоминанием о некоторых других средствах, доступных в ***R,*** для специальной регрессии и проблем анализа данных.

- **Смешанные модели.** Рекомендуемый пакет *nlme* обеспечивает функции *lme()* и *nlme()* для линейных и нелинейных моделей смешанных

эффектов, который является линейными и нелинейными регрессиями, в которых некоторые из коэффициентов соответствуют случайным эффектам. Эти функции интенсивно используют формулы для спецификации модели.

- **Локально подогнанные регрессии.** Функция *loess()* подгоняет непараметрическую регрессию использованием локально взвешенной регрессии. Такие регрессии полезны для выделения тренда в зашумленных данных или для снижения объема данных для обзора большого набора данных.
  Функция *loess()* находится в стандартном пакете *stats* вместе с кодом для прогноза следящей регрессии.
- **Устойчивая (робастная) регрессия**. Есть несколько функций для подгонки моделей, устойчивых к влиянию экстремальных выбросов в данных. Функция *lqs* в рекомендуемом пакете MASS обеспечивает современные алгоритмы для чрезвычайно устойчивой подгонки. Менее устойчивые, но статистически более эффективные методы, доступны в пакетах, например функция *rlm* в пакете MASS.
- **Аддитивные модели.** Этот метод стремится создавать функцию регрессии из аддитивных функций сглаживания детерминированных переменных, как правило, по одной для каждой независимой переменной. Функции *avas* и *ace* в пакете *acepack* и функциях *bruto* и *mars* в пакете *mda* обеспечивают некоторые примеры этих методов в пользовательских пакетах ***R***. Расширением является Обобщено Аддитивная Модель, реализованная в пользовательских пакетах *gam и mgcv*.
- **Древовидные модели**. Вместо поиска явно глобальной линейной модели для прогноза или интерпретации, древовидные модели стремятся рекурсивно разбить данные в критических точках независимых переменных для раздела данных, в конечном счете, на группы, которые являются настолько однородны, насколько возможно внутри группы, и настолько неоднородными, насколько возможно между группами. Результаты часто приводят к пониманию, к которому другие методы анализа данных не предоставляют.
  Модели еще специфицируются в обычной форме линейной модели. Функция подгонки называется *tree()*, но много других универсальных функций, таких как *plot() и text()*, хорошо адаптированных к отображению результатов древовидной модели в графической виде. Древовидные модели доступны в ***R*** через пользовательские пакеты *rpart* и *tree*.

12. Графические процедуры

Графические средства - важный и чрезвычайно универсальный компонент среды ***R.*** Возможно использование средств для вывода на экран широкого спектра статистических графиков, а также создать полностью новые типы графиков.

Графические средства могут использоваться как в интерактивном, так и в пакетном режимах, но в большинстве случаев интерактивное использование более продуктивно. Интерактивное использование также просто, потому что во время запуска ***R*** инициализируется графический драйвер устройства, который открывает специальное графическое окно для отображения интерактивной графики. Хотя это делается автоматически, полезно знать, что используется команда *X11()* под UNIX, *windows()* под Windows и *quartz()* под OS X. Новое устройство всегда может быть открыто с помощью *dev.new()*.

При запуске драйвера устройств могут использоваться команды рисования ***R*** для производства различных графических отображений и создания полностью новых типов изображений.

Команды рисования разделены на три основных группы:

- **Высокоуровневые** функции рисования создают новый рисунок на графическом устройстве, возможно с осями, метками, заголовками и так далее.
- **Низкоуровневые** функции рисования добавляют дополнительную информацию к существующему рисунку, такие как дополнительные точки, линии и метки.
- **Интерактивные** графические функции позволяют в интерактивном режиме добавить, либо извлечь информацию о существующем рисунке, используя графический манипулятор, например, мышь.

Кроме того, ***R*** поддерживает список графических параметров, которыми можно управлять для настройки рисунков.

Этот справочник описывает только то, что известно как 'базовая' графика. Отдельная графическая подсистема в пакете *grid* сосуществует с базовой - она более мощная, но ее труднее использовать. Есть рекомендуемый пакет *lattice*, который построен на *grid* и имеет инструменты для получения составных графиков наподобие тем, которые имеются в *Trellis* системы ***S***.

12.1. Высокоуровневые команды рисования

Высокоуровневые функции рисования разработаны для генерации полного рисунка данных, переданных функции в качестве параметров. При необходимости автоматически генерируются оси, метки и заголовки (если не указано иначе). Высокоуровневые команды рисования всегда запускают новый рисунок, стирая текущий рисунок в случае необходимости.

12.1.1. Функция plot()

Одна из наиболее часто используемых функций рисования в ***R*** - функция *plot()*. Это универсальная функция: тип произведенного рисунка зависит от типа или *класса* первого параметра.

plot(x, y)

plot(xy) Если ***x*** и ***y*** - векторы, то *plot(**x, y**)* выводит корреллограмму ***y*** по ***x***. Тот же самый эффект может быть получен, предоставляя один параметр (вторая форма) или как список, содержащий два элемента ***x*** и ***y*** или как матрицу из двух колонок.

plot(x)	Если ***x*** - временной ряд, то выводит график временного ряда. Если ***x*** – вектор чисел, то выводится рисунок значений в векторе по его индексу в векторе. Если ***x*** - комплексный вектор, то выводится график мнимой части против действительных частей элементов вектора.
plot(f)	
plot(f, y)	***f*** - факторный объект, ***y*** - числовой вектор. Первая форма генерирует рисунок бара ***f***; вторая форма производит свечи ***y*** для каждого уровня ***f***.
plot(df)	
plot(~ expr)	
plot(y ~ expr)	***df*** - фрейм данных, ***y*** - любой объект, *expr* - список имен объектов, разделенных '+' (например, a + b + c). Первые две формулы выводят графики распределения переменных во фрейме данных (первая формула) или многих именованных объектов (вторая формула). Третья формула рисует ***y*** против каждого объекта, указанному в *expr*.

12.1.2. Отображение многомерных данных

R обеспечивает две очень полезных функции для представления многомерных данных. Если ***X*** числовая матрица или фрейм данных, то команда:

```
> pairs(X)
```

производит попарные рисунки рассеивания переменных, определенных столбцами ***X***, то есть, каждый столбец ***X*** графически изображен против любого столбца ***X*** и результирующие *n (n - 1)* графиков расположены в виде матрицы с постоянным масштабом рисунка по строкам и столбцам матрицы.

Рассмотрение трех или четырех переменных *coplot* может быть более поучительным. Если ***a*** и ***b*** - числовые векторы, а ***c*** - числовой векторный или факторный объект (одинаковой длины), то команда:

```
> coplot(a ~ b | c)
```

производит набор рисунков рассеивания ***a*** против ***b*** для данного значения ***c***. Если ***c*** - фактор, то это просто означает, что ***a*** графически изображается против ***b*** для каждого уровня ***c***. Когда ***c*** является числовым, то он разделен на ряд условных интервалов и для каждого интервала ***a*** графически изображается против ***b*** для значения ***c*** в пределах интервала. Числом и позицией интервалов можно управлять путем *given.values = параметр,* а для *coplot()* - функция *co.intervals()* полезна для выбора интервалов. Также можно использовать две переменные в команде, как:

```
> coplot(a ~ b | c + d)
```

которая производит рисунки рассеивания ***a*** против ***b*** для каждого объединенного условного интервала ***c*** и ***d***.

Функции *coplot()* и *pairs()* обе берут параметр *panel*=, который можно использовать для настройки типа рисунка, который появляется в каждой из панелей. По умолчанию применяется *points()* для вывода графика рассеивания, но путем указания некоторых других низкоуровневых функций графики двух векторов ***x*** и ***y*** как значение panel =, можно произвести любой тип рисунка, который пожелаете. Примером полезной функции *panel* для *coplots* является *panel.smooth()*.

12.1.3. Графический вывод

Другие высокоуровневые функции графики производят различные типы рисунков. Некоторые примеры:

qqnorm(x)

qqline(x)

qqplot(x, y) Графики сравнения-распределения. Первый вариант рисует численный вектор ***x*** в сравнении с ожидаемым Нормальным распределением квантилей (график значений перцентилей, относящихся к долевым оценкам), а второй добавляет прямую линию к такому графику, проводя ее через распределение данных и квартили. Третий вариант выводит квантили ***x*** по ***y*** для сравнения их распределений.

hist(x)

hist(x, nclass=n)

hist(x, breaks=b, ...) Производит гистограмму числового вектора ***x***. Обычно выбирается разумное количество интервалов группировки, но можно дать рекомендацию путем *nclass = параметр*. Кроме этого, точки разбиения можно точно задать путем *breaks = параметр*. Если дан параметр *probability=TRUE*, бары представляют относительные частоты, разделенные на ширину колонки вместо расчетной.

dotchart(x, ...) Создает точечную диаграмму данных в ***x***. В точечной диаграмме ось ***Y*** дает маркирование данных в ***x***, и ось ***X*** дает свое значение. Например, это позволяет легкий визуальный выбор всех вводов данных со значением, находящимся в указанных диапазонах.

image(x, y, z, ...)

contour(x, y, z, ...)

persp(x, y, z, ...) Графики трех переменных. График *image* рисует прямоугольную сетку, используя различные цвета для представления значения ***z***, график *contour* рисует горизонтали для представления значение ***z***, и график *persp* рисует 3D поверхность.

12.1.4. Параметры для высокоуровневых графических функций

Есть ряд параметров, которые можно передать для высокоуровневых функций графики, такие как:

add=TRUE Заставляет функцию действовать в качестве низкоуровневой функции графики, накладывая рисунок на текущий рисунок (только для некоторых функций).

axes=FALSE Подавляет генерацию осей - полезно для добавления собственных осей функцией *axes()*. По умолчанию, *axes=TRUE* означает включение осей.

log="x"

log="y"

log="xy" Приводит ***x, y*** или обе оси к логарифмическому масштабу. Срабатывает не для всех видов графиков.

type= *type=параметр* управляет типом производимого графика, а именно:

type="p" Рисует отдельные точки (по умолчанию)

type="l" Рисует линии

type="b" Рисует точки, соединенные линиями

type="o" Рисует точки, перекрытые линиями

type="h" Рисует вертикальные линии от точки до нулевой оси (*высокая плотность*)

type="s"

type="S" Ступенчатая функция рисования. В первом случае верх вертикальной линии определяется точкой, во втором – низ.

type="n" Вообще не рисует. Однако оси все еще рисуются (по умолчанию), и система координат установлена согласно данным. Идеал для рисования последующими низкоуровневыми функциями графики.

xlab=string

ylab=string Метки для осей ***x*** и ***y***. Используйте эти параметры для изменения меток по умолчанию, обычно имен объектов, используемых в вызове высокоуровневой функции рисования.

main=string Заглавие рисунка крупным шрифтом в верхней части рисунка.

sub=string Подзаголовок меньшим шрифтом чуть ниже оси ***x***.

12.2. Низкоуровневые команды рисования

Иногда высокоуровневые функции рисования не дают требуемый вид графика. В этом случае можно использовать низкоуровневые команды рисования, чтобы прибавить дополнительную информацию (такую как точки, строки или текст) к текущему графику.

Некоторые из наиболее полезных низкоуровневых графических функций:

points(x, y)

lines(x, y) Прибавляет точки или связывающие линии к текущему графику. Для *plot() type = параметр* также передает этим функциям (и по умолчанию к "***p***" для *points()* и "***l***" для *lines()*).

text(x, y, labels, ...) Добавление текста к рисунку в точках, указанных ***x,y***. Обычно *labels* - целочисленный или символьный вектор, в этом случае *labels[i]* графически изображен в точке (*x [i], y [i]*). По умолчанию *1:length (x)*.

Замечание: Эта функция часто используется в последовательности:

plot(x, y, type="n"); text(x, y, names)

Графический параметр *type = "n"* подавляет точки, но устанавливает оси, а функция *text()*, предоставляет специальные символы, которые заданы именами в символьном векторе для каждой из точек.

abline(a, b)

abline(h=y)

abline(v=x)

abline(lm.obj) Прибавляет строку наклона ***b*** и смещение ***a*** к текущему рисунку. *h=y* может использоваться для указания ***y***-координаты высоты горизонтальной линии на рисунке, и *v=x* так же для ***x***-координат для вертикальных строк. Также *lm.obj* может быть списком с компонентом коэффициентов длины 2 (такой как результат функции подгонки модели), которые взяты в качестве смещения и наклона именно в таком порядке.

polygon(x, y, ...) Рисует многоугольник, определенный упорядоченными вершинами *(x,y)*, и (дополнительно) затеняет его штриховкой, или закрашивает его, если графическое устройство позволяет закраску рисунков.

legend(x, y, legend, ...) Прибавляет легенду текущему рисунку в указанной позиции. Начертания символов, стили линий, цвета и т.д., определяются метками в символьном векторе описания. По крайней мере, один аргумент ***v*** (вектор такой же длины как описание) с соответствующими значениями единиц измерения должен также быть задан следующим образом:

- *legend(, fill=v)* цвет заполнения прямоугольника
- *legend(, cov=v)* цвет точек или линий
- *legend(, lty=v)* стиль линии
- *legend(, lwd=v)* ширина линий
- *legend(, pch=v)* рисует символы (вектор символов)

title(main, sub) Прибавляет сверху основной заголовок текущего рисунка большим шрифтом, и (дополнительно) подзаголовок *sub* в нижней части меньшим шрифтом.

axis(side, ...) Прибавляет ось к текущему рисунку на стороне, данной первым параметром (1 - 4, рассчитывая по часовой стрелке от нижней части.) Другие параметры управляют расположением оси внутри или рядом с рисунком, и позиций отметок и меток. Полезно для добавления пользовательских осей после вызова *plot()* с аргументом *axes=FALSE*.

Низкоуровневые функции рисования обычно запрашивают некоторую информацию расположения (например, координаты ***x*** и ***y***) для определения положения новых элементов рисунка. Координаты даны с точки зрения пользовательских координат, которые определены предыдущей высокоуровневой командой графики и выбираются на основе предоставленной информации.

Если параметры ***x*** и ***y*** обязательны, то достаточно предоставить один аргумент из списка с элементами по имени ***x*** и ***y***. Матрица с такими же двумя столбцами - также допустимый ввод. Таким образом, функции, такие как *locator()* (см. ниже) могут использоваться для определения позиции на рисунке в интерактивном режиме.

12.2.1 Математическая аннотация

В некоторых случаях полезно прибавить математические символы и формулы к рисунку. Это может быть достигнуто в ***R*** не символьной строкой в любом из *text, mtext, axis* или *title*, а описанием выражения. Например, следующий код рисует формулу биномиальной функции распределения:

```
> text(x, y, expression(paste(bgroup("(", atop(n, x), ")"), p^x, q^{n-x})))
```

Более подробную информацию, включая полное перечисление доступных возможностей, можно получить в ***R,*** используя команды:

```
> help(plotmath)
> example(plotmath)
> demo(plotmath)
```

12.2.2 Векторные шрифты Херши

Можно специфицировать векторные шрифты Херши для рендеринга текста, когда используются функций контура и текста. Есть три причины использования шрифтов Херши:

- Шрифты Херши могут дать лучший результат особенно на мониторе для повернутого и/или мелкого текста.
- Шрифты Херши предоставляют некоторые символы, которые, возможно, не доступны в стандартных шрифтах. В частности есть знаки Зодиака, картографические символы и астрономические символы.
- Шрифты Херши обеспечивают кириллические и японские (Кана и Кандзи) символы.

Более подробную информации, включая таблицы символов Херши, можно получить в ***R*** путем использования команд:

```
> help(Hershey)
> demo(Hershey)
> help(Japanese)
> demo(Japanese)
```

12.3. Интерактивная графика

R также обеспечивает функции, которые позволяют пользователям извлекать или прибавлять информацию к рисунку с помощью мыши. Самой простой из них является функция *locator()*:

locator(n, type)

Ожидает пользовательский выбор расположения на текущем рисунке, используя левую кнопку мыши. Это продолжается до тех пор, пока не будет выбрано ***n*** (по умолчанию 512) точек, или нажата другая кнопка мыши. Параметр *type* применим для вывода отмеченных точек и действует также, как высокоуровневые команды графики; по умолчанию: вывод отсутствует. *locator()* возвращает расположения точек, выбранных как список с двумя компонентами ***x*** и ***y***.

locator() обычно вызывают без параметров. Это особенно полезно для интерактивного режима выбора позиции для графических элементов, таких как легенды или метки, когда трудно вычислить заранее, куда надо поместить графику. Например, чтобы поместить некоторый информативный текст около отдаленной точки, команда

> text(locator(1), "Outlier", adj=0)

может быть полезной. *locator()* будет игнорироваться, если текущее устройство, такое как *postscript* не будет поддерживать интерактивное указание.

identify(x, y, labels)

Позволяет пользователю выделить любую из точек, определенных ***x*** и ***y*** (использующий левую кнопку мыши) рисованием соответствующий компонент меток поблизости (или индекс точки, если метки отсутствуют). Возвращает индексы выбранных точек при нажатой другой кнопке.

Иногда желательно идентифицировать определенные точки на рисунке, а не их позиции. Например, необходимо выбрать некоторое интересуемое наблюдение на графическом дисплее, а затем определенным способом управлять этим наблюдением. Для получения значения координат *(x; y)* в двух числовых векторах ***x*** и ***y*** можно использовать функцию *identify()* следующим образом:

> plot(x, y)

> identify(x, y)

Функция *identify()*непосредственно не выполняет рисования, а просто позволяет пользователю перемещать указатель мыши и щелкать левой кнопкой мыши около точки. Если точка существует около указателя мыши, то она будет отмечена с ее индексом (то есть, ее позицией в векторах ***x/y***) графически изображенной поблизости. Кроме этого, можно использовать некоторую информативную строку (такую как имя выбора) как выделение путем использования параметра *labels* для *identify()*, или отключить маркировку в целом параметром *plot = FALSE*. Когда процесс завершен (см. выше), *identify()* возвращает индексы выбранных точек; можно использовать эти индексы для извлечения выбранных точек из исходных векторов ***x*** и **y.**

12.4. Использование графических параметров

Создавая графику, особенно для выступлений или публикаций, по умолчанию ***R*** не всегда выводит именно то, что требуется. Однако можно настроить практически каждый аспект показа, используя графические параметры. ***R*** поддерживает список из большого числа графических параметров, которые управляют вещами, такими как стиль линии, цвета, расположение рисунка и текстовое выравнивание среди многих других. У каждого графического параметра есть имя (такое как *'col'* для цвета) и значение (номер цвета, например.)

Отдельный список графических параметров сохраняется для каждого активного элемента, и у каждого устройства есть набор параметров по умолчанию при инициализации. Графические параметры могут быть установлены двумя способами: либо постоянно, влияя на все графические функции, которые получают доступ к текущему устройству; или временно, влияя только на отдельный вызов графической функции.

12.4.1. Постоянные изменения: функция par()

Функция *par()* используется для доступа и изменения списка графических параметров для текущего графического устройства.

par() Без параметров, возвращает список всех графических параметров и их значения для текущего устройства.

par(c("col", "lty"))

Если аргумент символьный вектор, то возвращает только именованные графические параметры (снова, как список.)

par(col=4, lty=2)

С поименованными параметрами (или одним общим списком аргументов) устанавливает значение именованных графических параметров, и возвращает исходные значения параметров как список.

При установке графических параметров функцией *par()* значения параметров изменяются постоянно, в том смысле, что на все будущие вызовы графических функций (на текущем устройстве) будет влиять новое значение. Можно думать об установленных графических параметрах как об установке значений "по умолчанию" для параметров, которые будут использоваться всеми графическими функциями, если потом не задаются альтернативные значения.

Заметим, что вызовы *par()* ***всегда*** влияют на глобальное значение графических параметров, даже при *par()* вызове изнутри функции. Это часто нежелательное поведение - обычно желательно установить некоторые графические параметры, сделать некоторые рисунки, а затем восстановить исходные значения, чтобы не влиять на сеанс пользователя ***R.*** Можно восстановить начальные значения при внесении изменений, сохраняя результат *par()*, и восстановить первоначальные значения по завершению рисования.

```
> oldpar <- par(col=4, lty=2)
```

. . . команды рисования . . .

```
> par(oldpar)
```

Чтобы восстановить все установленные графические параметры следует использовать:

```
> oldpar <- par(no.readonly=TRUE)
```

. . . команды рисования . . .

```
> par(oldpar)
```

12.4.2. Временные изменения: параметры для графических функций

Графические параметры можно также передать (почти) любой графической функции как именованные параметры. Это действует так же, как передача параметров функции *par()*, за исключением того, что изменения действуют только во время вызова функции. Например:

```
> plot(x, y, pch="+")
```

производит рисунок рассеивания, используя знак "плюс" в качестве символа рисования, не изменяя символ рисования по умолчанию для будущих рисунков.

К сожалению, это не реализуется полностью последовательно, и иногда необходимо установить и сбросить графические параметры, используя *par()*.

12.5. Список графических параметров

Следующие разделы детализируют многие из широко используемых графических параметров. В справочной документации по ***R*** сведения для функция *par()* предоставлены кратко; здесь дан несколько более подробный вариант.

Графические параметры будут представлены в следующей форме:

name=value

Описание воздействия параметра. *name* является именем параметра, то есть, имя параметра для использования в вызовах

par() или графических функциях. *value* - типичное значение, которое можно использовать при задании параметров.

Заметим, что оси **не** графические параметры, а параметр нескольких методов *plot*: см. *xaxt* и *yaxt*.

12.5.1. Графические элементы

Рисунки ***R*** составлены из точек, линий, текста и многоугольников (заполненных областей). Графические параметры существуют для управления рисованием этих графических элементов, а именно:

pch="+" Символ, который будет использоваться для рисования точек. По умолчанию меняется в зависимости от графических драйверов, но, как правило, круг. Графически изображенные точки имеют тенденцию появляться немного выше или ниже соответствующей позиции, если Вы не используете "." в качестве символа рисования, который производит центрируемые точки.

pch=4 Если *pch* равно целому числу между 0 и 25, то производится специальный символ печати. Чтобы увидеть символ, используйте команду

> legend(locator(1), as.character(0:25), pch = 0:25)

Символы с 21 до 25 могут копировать более ранние символы, но могут быть окрашены по-разному: см. справку по *points* и ее примеры.

Кроме того, *pch* может быть символом или числом в диапазоне 32:255 для представления символа в текущем шрифте.

lty=2 Типы линии. Альтернативные стили линии не поддерживаются всеми графическими устройствами (и различаются у разных устройств), но тип 1 – это всегда сплошная линия, тип 0 линии всегда невидим, и линия 2 и больше являются штрих-пунктирными или пунктирными линиями, или некоторой комбинацией обеих.

lwd=2 Ширина линии. Требуемая ширина линий кратна "стандартной" толщине линии. Влияет на линии оси, а так же на проведенные *lines()*, и т.д. Не все устройства поддерживают это, и у некоторых есть ограничения на ширину, которой можно пользоваться.

col=2 Цвета, которые будут использоваться для точек, линий, текста, заполнений области и изображений. Число от текущей палитры (см.*?palette*) или названия цвета.

col.axis

col.lab

col.main

col.sub Цвет, который будет использоваться для аннотации осей и меток ***x*** и ***y***, заголовка и подзаголовки, соответственно.

font=2 Целое число, которое определяет шрифт, используемый для текста. Если возможно, драйверы устройства организуют это так, чтобы 1 соответствовал простому тексту, 2 полужирному стилю, 3 курсиву, 4 полужирному курсиву и 5 символьному шрифту (включающему греческие буквы).

font.axis

font.lab

font.main

font.sub Шрифт, который будет использоваться для аннотации осей и меток ***x*** и ***y*** заголовка и подзаголовков, соответственно.

adj=-0.1 Выравнивание текста относительно позиции рисования. 0 означает выравнивание влево, 1 означает выравнивание вправо и 0.5 означает выравнивание по центру горизонтально по позиции рисования. Фактическое значение - соотношение текста, который появляется налево от позиции рисования, таким образом, значение -0.1 оставляет между текстом и позицией вывода пробел 10% от ширины текста.

cex=1.5 Растяжение символов. Значение - требуемый размер символов (включая выводимые символы) относительно размера текста по умолчанию.

cex.axis

cex.lab

cex.main

cex.sub Растяжение символов, которое будет использоваться для аннотации осей и меток ***x*** и ***y*** заголовков и подзаголовков, соответственно.

12.5.2. Оси и метки

Многие высокоуровневые рисунки ***R*** имеют оси, но можно создать оси самостоятельно с помощью низкоуровневой графической функции *axis()*. Оси имеют три основные компоненты: линию оси (стиль линии под контролем графического параметра *lty*), шкалу (которая обозначает деление линии оси единицами измерения) и метки шкалы (которые обозначают единицы измерения.) Эти компоненты можно настроить следующими параметрами графики.

lab=c(5, 7, 12)

Первые два числа - требуемое число интервалов на осях ***x*** и ***y***, соответственно. Третье число - требуемая длина меток оси в символах (включая десятичную точку.) Выбор слишком малого значения для этого параметра может привести к тому, что все метки масштаба округлятся к одному числу!

las=1 Ориентация меток оси. 0 всегда означает параллельно оси, 1 всегда означает горизонтально и 2 всегда означает перпендикулярно к оси.

mgp=c(3, 1, 0) Позиции компонентов оси. Первый компонент - расстояние от метки оси до позиции оси в текстовых строках. Второй компонент - расстояние до меток шкалы, и заключительный компонент - расстояние от позиции оси до линии оси (обычно нуль). Положительные числа измеряют вне рисунка, отрицательные числа - внутри.

tck=0.01 Длина меток шкалы, как доля размера области рисования. Если *tck* является небольшим (меньше чем 0.5), то метки на осях ***x*** и ***y***

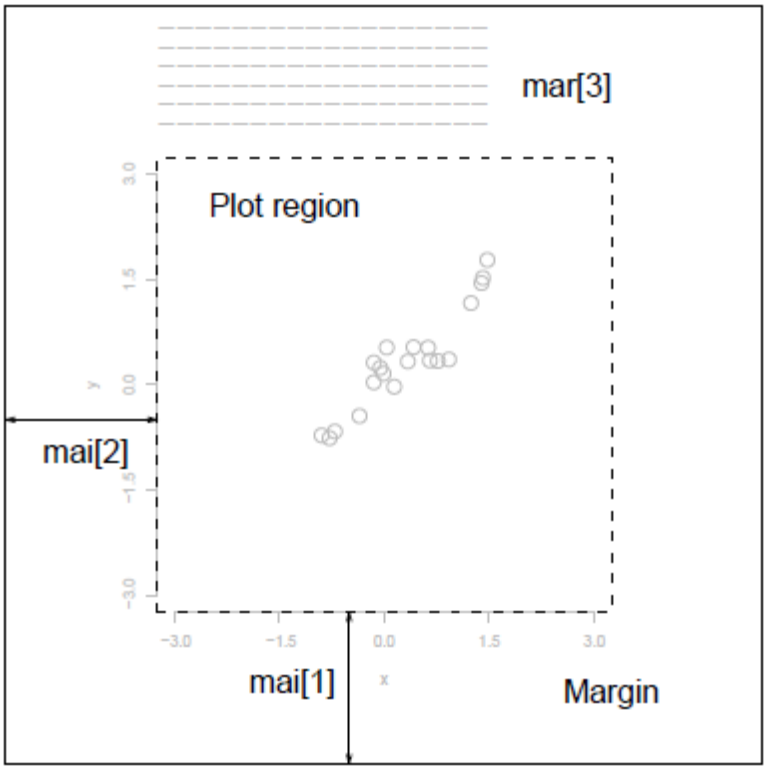
принудительно имеют одинаковые размеры. Значение 1 дает сетку линий. Отрицательные величины дают метки вне области рисования. Используйте *tck=0.01* и *mgp=c (1,-1.5,0)* для внутренних меток шкалы.

xaxs="r"

yaxs="i" Стили для осей ***x*** и ***y***, соответственно. Со стилями ***"i"*** (внутренний) и ***"r"*** (по умолчанию) метки шкалы всегда находятся в пределах диапазона данных, однако стиль ***"r"*** оставляет небольшое пространство на краях. (***S*** имеет другие стили, не реализованные в ***R***).

12.5.3. Поля рисунка

Отдельный рисунок в ***R*** известен как *figure* и включает область рисунка, окруженную полями (возможно содержащий метки оси, заголовки, и т.д.) и (обычно) ограничен осями непосредственно.



Графические параметры, контролирующие формат фигуры включают:

mai=c(1, 0.5, 0.5, 0) Ширина поля внизу, слева, вверху и справа, соответственно, измеренная в дюймах

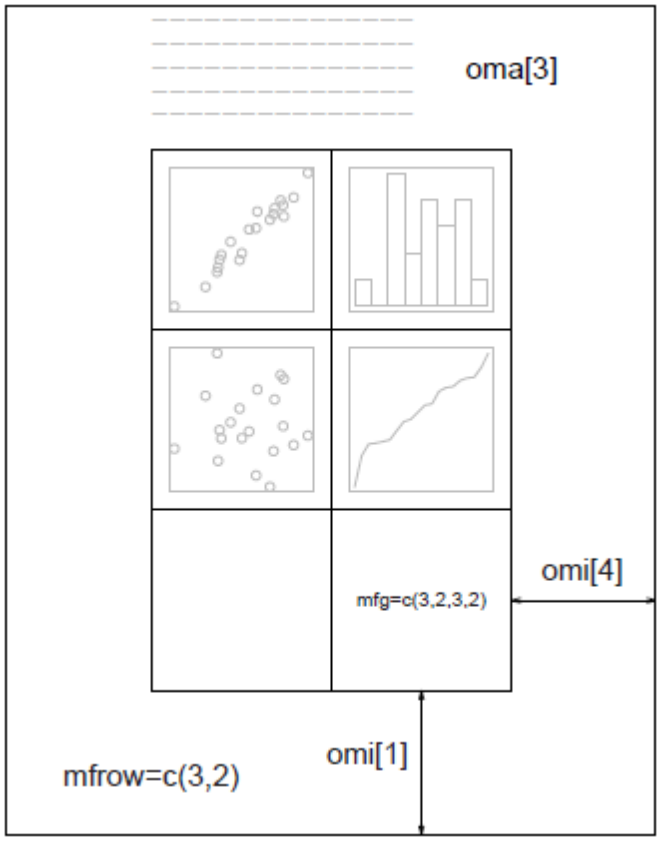
mar=c(4, 2, 2, 1) Подобно *mai*, за исключение того, что единицей измерения являются строки текста.

mar и *mai* эквивалентны в том смысле, что настройка одного изменяет значение другого. Значения по умолчанию, выбранные для этого параметра, являются часто слишком большими; правое поле редко необходимо, и не нужно верхнее поле, если заголовок не используется. Нижние и левые поля должны быть достаточно большими, чтобы разместить ось и метки масштаба. Кроме того, умолчания выбраны без учета размера поверхности устройства: например, используя *postscript()* драйвер с *height=4* параметром приведет к рисунку, который является приблизительно 50%-ым полем,

если *mar* или *mai* не установлены явно. Когда выводятся составные фигуры (см. ниже), то поля уменьшаются, однако это, возможно, не достаточно, когда много рисунков совместно разделяют одну страницу.

12.5.4. Окружение составных фигур

R позволяет создать массив рисунков ***n на m*** на отдельной странице. У каждого рисунка есть свои собственные поля, и массив рисунков дополнительно окружен внешним полем, как показано в следующем рисунке.



Графические параметры, связанные с составными фигурами это:

mfcol=c(3, 2)

mfrow=c(2, 4) Устанавливает размер массива составных фигур. Первое значение - это количество строк, а второй - число столбцов. Единственная разница между этими двумя параметрами, что установление mfcol выводит фигуры по колонкам; mfrow заполняет массив построчно.

Расположение на *figure* можно было создать путем установки *mfrow=c(3,2);* фигура отобразится на странице после вывода всех четырех рисунков.

Установка любого из этих показателей может сократить базовый размер символов и текста (контролируется *par("cex")* и *pointsize* устройства вывода). В формате с ровно двумя строками и столбцами базовой размер уменьшается на коэффициент 0.83: если есть три или больше строки или столбца, коэффициент уменьшения равен 0.66.

mfg=c(2, 2, 3, 2) Позиция текущей фигуры в составном фигурном окружении. Первые два номера это строка и столбец текущей фигуры; последние два числа это строки и столбцы

в составном массиве фигур. Используйте этот параметр для перехода между фигурами в массиве. Вы даже можете использовать отличающиеся от истинных значений значения для последних двух аргументов, для отображения разно размерных фигур на одной и той же странице.

fig=c(4, 9, 1, 4)/10 Позиция текущей фигуры на странице. Значения параметров это позиции левого, правого, верхнего и нижнего края соответственно, в процентах от размера страницы, измеряемого от нижнего левого углу. Значению примера соответствует фигура в нижнем правом углу страницы. Используйте этот параметр для произвольного позиционирования фигур на странице. Если вы хотите добавить фигуру к текущей странице, используйте *new=TRUE* (в отличие от S).

oma=c(2, 0, 3, 0) omi=c(0, 0, 0.8, 0)

Размер внешних полей. Как *mar* и *mai*, первое измеряется в текстовых строках, а второе - в дюймах, начиная с нижнего поля, и выполняется по часовой стрелке.

Внешнее поле особенно полезно для заметок на полях и т.д. Текст может быть добавлен к внешнему полю функцией *mtext()* с аргументом *outer=TRUE*. По умолчанию не существует внешних полей, именно поэтому вы должны создать их непосредственно используя *oma* или *omi*.

Более сложные схемы составных фигур могут быть изготовлены функциями *split.screen()* и *layout()*, а также используя пакеты *grid* и *lattice*.

12.6. Устройства вывода

R может генерировать графику (разного уровней качества) на почти любом типе дисплея или печатающего устройства. Но сначала ***R*** должен быть проинформирован об устройстве, с которым он имеет дело. Это делается путем запуска драйвера устройства. Цель драйвера устройства состоит в преобразовании графических инструкций из ***R*** (“чертить линию”, например) в форму, которую может понять определенное устройство.

Драйверы устройства стартуют путем вызова функцию драйвера устройства. Существует единственная функция для каждого устройства графического вывода: введите *help(Devices)*) для получения полного списка. Например, ввод команды:

```
> postscript()
```

отправляет весь будущий графический вывод на принтер в формате PostScript. Некоторые широко используемые драйверы устройства:

X11()	Для использования с X11 оконной системой Unix-подобных ОС
windows()	Используется в Windows
quartz()	Используется в OS X
postscript()	Для печати на принтерах PostScript, или создания графических файлов PostScript.
pdf()	Производит файл PDF, который также может быть включен в другие файлы PDF.

png()	Производит двоичный файл PNG. (Доступно не всегда. См. справку).
jpeg()	Производит двоичный файл JPEG, наиболее подходящий для изображений. (Доступно не всегда. См. справку).

При завершении работы с устройством следует убедиться в завершении драйвера устройства помощью команды:

> dev.off()

Она гарантирует, что устройство заканчивается чисто; например, в случае устройств на бумажном носителе гарантируется завершение всех страниц и их отправка на принтер. (Это произойдет автоматически при нормальном конце сеанса).

12.6.1. PostScript диаграммы для типографии

Добавляя параметр с именем файла в функцию драйвера вывода *postscript()*, можно сохранить графику в формате PostScript в произвольный файл. Рисунок будет иметь альбомную ориентацию, если только не приводится параметр *horizontal=FALSE*, и вы можете контролировать размер изображения параметрами *width* и *height* (график будет масштабирован с учетом этих размеров.) Например, команда:

> postscript("file.ps", horizontal=FALSE, height=5, pointsize=10)

подготовит файл, содержащий код PostScript для рисунка пять дюймов высотой, возможно для включения в документ. Важно отметить, что, если файл, названный в команде, уже существует, то он будет перезаписан. Дело обстоит так, даже если файл только создавался ранее в этом же самом сеансе ***R.***

Обычно PostScript вывод включают как рисунок в другой документ. Это работает лучше всего, когда используют инкапсулированный PostScript: ***R*** всегда делает совместимый графический вывод, но только в случае задания параметра *onefile=FALSE*. Происходит это из-за S-совместимости: в этом случае действительно достигается то, что результаты будут на одной странице (что является условием EPSF спецификации). Таким образом, для получения графика для последующего включения используют что-то вроде:

> postscript("plot1.eps", horizontal=FALSE, onefile=FALSE,
height=8, width=6, pointsize=10)

12.6.2. Несколько графических устройств одновременно

В профессиональном использовании ***R*** часто полезно иметь несколько графических устройств в использовании одновременно. Конечно, только одно графическое устройство может принять графические команды в любой момент, и это известно как текущее устройство. Когда открыто много устройств, то они формируют пронумерованную последовательность с именами, определяющими тип устройства в любой позиции.

Основные команды, используемые для работы многими устройствами, и их значениями, следующие:

X11() [UNIX]
windows()
win.printer()
win.metafile()
[Windows]
quartz() [OS X]
postscript()

pdf()
png()
jpeg()
tiff()
bitmap()

Каждое новый вызов функции драйвера устройства вывода открывает новое графическое устройство, что расширяет на одну позицию список устройств вывода. Это устройство становится текущим устройством вывода, и графический вывод направляется на него. (Некоторые платформы могут иметь дополнительные устройства).

dev.list()	Возвращает количество и наименование всех активных устройств вывода. Устройство в позиции 1 списка всегда является *null* устройством, которое вообще не принимает графических команд.
dev.next()	
dev.prev()	Возвращает номер и название графического устройства, последующего или предыдущего соответственно.
dev.set(which=k)	Может быть использована для изменения текущего графического устройства на устройство в позиции номер ***k*** списка устройств вывода. Возвращает номер и метку устройства.
dev.off(k)	Закрыть графическое устройство в позиции ***k*** списка устройств вывода. Для некоторых устройств, таких, как устройства *postscript*, либо немедленно распечатывается файл или корректно завершать файл для последующей печати, в зависимости от того как было инициировано устройство.
dev.copy(device, ..., which=k) dev.print(device, ..., which=k)	Делает копию устройства ***k***. Здесь устройство - это функция драйвера устройства, такое как *postscript*, в случае необходимости с дополнительными аргументами, назначенными посредством `...'. *dev.print* аналогично, но копируемое устройство немедленно закрывается, так что завершающие, такие как печать твердой копии, действия немедленно выполняться.
graphics.off()	Закрыть все графические устройства в этом списке, кроме *null* устройства.

12.7. Динамическая графика

У ***R*** нет встроенных возможностей динамической или интерактивной графики, например, «вращения облака точек» или "рисования кистью" (выделение в интерактивном режиме) точки. Однако, обширные динамические графические средства доступны в системе GGobi Swayne, Cook и Buja, доступной по адресу:

http://www.ggobi.org/

и к ним можно получить доступ из ***R*** через пакет *rggobi*, описаным в *http://www.ggobi.org/ rggobi.*

Кроме того, пакет *rgl* обеспечивает способы взаимодействовать с 3-D рисунками, например поверхностями.

13. Пакеты

Все функции ***R*** и наборы данных хранятся в пакетах. Только, когда пакет загружен, доступно его содержание. Это сделано и для эффективности (полный список взял бы больше памяти и займет больше времени, чем поиск в подмножестве), и разработчикам программы помощи, которые защищены от коллизии имен с другим кодом. Процесс разработки пакетов описан в Разделе, "Создание пакетов" в Написании расширений ***R*** . Здесь, мы опишем их с точки зрения пользователя.

Чтобы увидеть установленные пакеты на Вашей инсталляции, введите команду:

```
> library()
```

без параметров. Загрузить определенный пакет (например, пакет начальной загрузки, содержащий функции от Davison & Hinkley (1997)), используют команду:

```
> library(boot)
```

Пользователи, соединенные с Интернетом, могут использовать функции *install.packages()* и *update.packages()* (доступно через меню Packages в Windows и GUI, см. Раздел "Пакеты установки" в Установке и Администрировании ***R***) установить и обновить пакеты.

Чтобы узнать загруженные в настоящий момент пакеты, следует ввести:

```
> search()
```

для отображения список поиска. Некоторые пакеты могут быть загружены, но не доступные в списке поиска (см. Раздел 13.3 [Пространства имен]): они будут включены в список, выводимый по:

```
> loadedNamespaces ()
```

Чтобы увидеть список всех доступных тем справки в установленном пакете, следует использовать:

```
> help.start ()
```

для запуска системы справочной информации HTML, а затем перейти на список пакетов в разделе «Ссылки».

13.1. Стандартные пакеты

Стандартные (или *base*) пакеты считаются частью исходного кода ***R***. Они содержат основные функции, которые позволяют ***R*** работать, и наборы данных и стандартные статистические и графические функции, которые описаны в этом справочнике. Они должны быть автоматически доступными в любой установке ***R***. См. Раздел "***R*** пакеты" в FAQ ***R*** для полного списка.

13.2. Сторонние пакеты и CRAN

Есть тысячи сторонних пакетов для ***R,*** записанных многими различными авторами. Некоторые из этих пакетов - это реализация специализированных статистических методов, другие, предоставляют доступ к данным или аппаратным средствам, а третьи призваны дополнять руководства. Некоторые из них (рекомендуемые пакеты) распространяются с каждой бинарной дистрибуцией ***R.*** Большинство доступно для скачивания с CRAN (*http://CRAN.R -project.org/* и его зеркала) и других репозитариев, таких как Биопроводник (http://www.bioconductor.org/) и Omegahat (http://www.omegahat.org/). FAQ ***R*** содержит список пакетов CRAN, актуальных во время публикации, но набор доступных пакетов меняется очень часто.

13.3. Пространства имен

У пакетов могут быть пространства имен, и в настоящий момент все *base* и рекомендуемые пакеты имеют пакет наборов данных. Пространства имен делают три вещи: они позволяют писателю пакета скрывать функции и данные, которые предназначаются только для внутреннего пользования, они препятствуют тому, чтобы функции ломались, когда пользователь (или автор другого пакета) выбирает имя, которое пересекается с другим в пакете, и они предусматривают способ обратиться к объекту в пределах определенного пакета.

Например, ***t()*** является функцией транспонирования в ***R,*** но пользователи могли бы определить свою собственную функцию, названную *t*. Пространства имен препятствуют тому, чтобы определение пользователя имело приоритет, и сломало каждую функцию, которая пытается транспонировать матрицу.

Есть два оператора, которые работают с пространствами имен. Оператор двойного двоеточия ***::*** выбирает определения из определенного пространства имен. В примере выше, функция транспонирования всегда будет доступна как *base :: t*, потому что это определено в пакете *base*. Таким способом можно получить только функции, которые экспортируются из пакета.

Оператор тройного двоеточия ***:::*** может быть замечен в нескольких местах в коде ***R***: он действует как оператор двойного двоеточия, но кроме этого предоставляет доступ к скрытым объектам. Пользователи, более вероятно, будут использовать функцию *getAnywhere(),* которая ищет сразу в нескольких пакетах.

Пакеты являются часто взаимозависимыми, и загрузка одного пакета может привести к автоматической загрузке других. Операторы двоеточия, описанные выше, также вызовут автоматическую загрузку присоединенного пакета. При автоматической загрузке пакетов с пространством имен они не добавляются к списку поиска.

13.4. Пакеты для анализа временных рядов

Оригинал текста: http://cran.r-project.org/web/views/TimeSeries.html

Остальные ссылки можно взять в оригинале.

Большая функциональность базовой поставки ***R*** полезна для временных рядов в особенности в пакете статистик. Он дополнен многими пакетами на CRAN, которые кратко изложены ниже. Есть также значительное перекрытие между инструментами для временного ряда и в задачах Эконометрики и Финансов. Пакеты в этом виде могут быть примерно структурированы в следующие темы. Если Вы думаете, что некоторый пакет пропущен в списке, пожалуйста, сообщите нам.

13.4.1. Основные пакеты - Basics

Инфраструктура: Основные пакеты ***R*** содержит базовую инфраструктуру для представления и анализа данных временного ряда. Фундаментальный класс - ***"ts"***, который может представить расположенный с равными интервалами временной ряд (использующий числовые отметки времени). Следовательно, он является особенно подходящим для ежегодных, ежемесячных, ежеквартальных данных и т.д.

Моделирование: Методы для анализа и моделирования временного ряда включают модели ARIMA в *arima(),* AR(p) и VAR(p) модели в *ar(),* структурные модели в *StructTS(),* визуализация через *plot(),* (частные) автокорреляционные функции *acf()* и *pacf(),* классическое разложение в *decompose(),* разложение STL в *stl(),* скользящее среднее и авторегрессивные линейные фильтры в *filter(),* и основной прогноз Holt-Winters в *HoltWinters().*

13.4.2. Время и даты -Times and Dates

Класс *"**ts**"* может только иметь дело с числовыми отметками времени, но еще много классов доступно для хранения информации времени/даты и вычислений с ними. Для краткого обзора смотри Справочную службу R: Класс Даты и Времени в ***R*** Гэбором Гротендиком и Томасом Пецолдтом в Новостях ***R*** 4 (1), 29-32.

Классы "*yearmon*" и "*yearqtr*" из ***zoo*** учитывают более удобное вычисление с ежемесячными и ежеквартальными наблюдениями, соответственно.

Класс "*Date*" из основного пакета является основным классом для дат в ежедневных данных. Даты внутренне хранятся как число дней с 01.01.1970.

Пакет *chron* обеспечивает классы для *dates(), hours()* и даты/времени (интрадэй). Отсутствует поддержка часовых поясов и летнего времени. Внутренне объекты "*chron*" – это дробные дни с 01.01.1970.

Классы "POSIXct" и "POSIXlt" реализуют POSIX - стандарт (интрадэй) информации для даты/времени, и также поддерживают часовые пояса и летнее время. Однако, вычисления часового пояса требуют некоторого внимания и могут быть системно зависимыми. Внутренне объекты "POSIXct" - число секунд, начиная с 01.01.1970 GMT 0:00:00. Пакет *lubridate* обеспечивает функции, которые облегчают определенные вычисления POSIXbased.

Класс "*timeDate*" обеспечен в *timeDate* пакете (ранее: fCalendar). Он нацелен на финансовую информацию времени/даты и соглашения с часовыми поясами и временем перехода на летнее время через новое понятие "финансовых центров". Внутренне, вся информация хранится в "POSIXct" и делает все вычисления только в GMT. Также включены календарные функции, например, информацию о выходных и праздничных днях для различных фондовых бирж.

tis пакет обеспечивает "***ti***" класс для информации времени/даты.

"*mondate*" класс от пакета *mondate* облегчает вычисления с датами с точки зрения месяцев.

rtv пакет учитывает представление, манипулирование и визуализацию переменных случайного времени.

TSAgg обеспечивает функции для агрегации неполных данных временного ряда.

tempdisagg пакет включает методы для временного разукрупнения и интерполяции низкочастотного временного ряда к ряду более высокой частоты.

13.4.3. Классы временных рядов - Time Series Classes

Как упомянуто выше, "***ts***" - основной класс для расположенного с равными интервалами временного ряда, использующего числовые отметки времени.

Пакет ***zoo*** обеспечивает инфраструктуру для временного ряда с регулярно и нерегулярно расположенными интервалами, используя произвольные классы для отметок времени (то есть, позволяя все классы из предыдущего раздела). Он разработан так, чтобы по мере возможности не противоречить "***ts***". Приведение «***из***» и «***в***» "***zoo***" доступно для всех других классов, упомянутых в этом разделе.

Пакет ***xts*** основан на ***zoo*** и обеспечивает универсальную обработку различных основанных на времени классов данных ***R***.

Различные пакеты реализуют неправильный временной ряд, основанный на отметках времени "POSIXct", предназначенных специально для финансовых применений. Они включают "*irts*" от *tseries* и "*fts*" от *fts*.

Класс *"timeSeries*" в *timeSeries* (ранее: fSeries) реализует временной ряд с "*timeDate*" отметками времени.

Класс "*tis*" в *tis* реализует временной ряд с "*ti*" отметками времени.

Пакет *tframe* содержит инфраструктуру для установки периодов времени в различных форматах.

13.4.4. Прогноз и одномерное моделирование -Forecasting and Univariate Modeling

Пакет forecast обеспечивает класс и методы для одномерных прогнозов временного ряда, и обеспечивает много функций, реализовывая различные модели прогноза, включая все из пакета статистик.

Экспоненциальное сглаживание: *HoltWinters()* в статистиках предоставляет некоторым базовым моделям частную оптимизацию, *ets()* из пакета *forecast* обеспечивает больший набор моделей и средств с полной оптимизацией.

Авторегрессивные модели: *ar()* в статистиках (с выбором модели), *FitAR* для подмножества моделей AR, и *pear* для периодических авторегрессивных моделей временного ряда.

Модели ARIMA:

- *arima()* в статистиках является основной функцией для ARIMA, SARIMA, ARIMAX, и подмножества модели ARIMA. Это улучшено в пакете прогноза наряду с *auto.arima()* для автоматического выбора порядка.
- *arma()* в пакете *tseries* обеспечиваются различные алгоритмы для моделей ARMA и ее подмножеств;
- *FitARMA* реализует быстрый алгоритм MLE для моделей ARMA. Некоторые услуги для дробных дифференцированных моделей ARFIMA предоставлены в пакете *fracdiff*. *afmtools* проводит оценку, диагностику и прогноз для моделей ARFIMA. *armaFit()* из пакета *fArma* является интерфейсом для ARIMA и моделей ARFIMA. Пакет *gsarima* содержит функции для имитации обобщенной модели временного ряда SARIMA. Пакет *mar1s* обрабатывает мультипликативный AR(1) с сезонными процессами.

Модели GARCH:

- *garch()* из *tseries* подгоняет основным моделям GARCH, *garchFit()* из *fGarch* реализует модели ARIMA с широким классом инноваций GARCH.
- *bayesGARCH* оценивает Байесовскую модель GARCH(1,1) с ***t*** инновациями.
- *gogarch* реализует модель Обобщенной Ортогональной GARCH (GOGARCH).
- Проект R-Forge *rgarch* стремится обеспечивать гибкое и богатое моделирование GARCH и тестовую среду, включая одномерные и многомерные пакеты GARCH. На его веб-странице есть обширная информация и примеры.

Разное: *ltsa* содержит методы для линейного анализа временных рядов, *dlm* для Байесового анализа динамических линейных моделей, *timsac* для анализа временных рядов и управления, *BootPR* для исправления смещения прогноза и интервалов прогноза загрузки для авторегрессивного временного ряда.

13.4.5. Ресэмплирование - Resampling

Бутстрэппинг: пакет *boot* обеспечивает функцию *tsboot()* для бутстрэпинга временного ряда, включая блочный бутстрэпинг с несколькими разновидностями.

tsbootstrap() от *tseries* обеспечивает быстрый стационарный и блочный бутстрэпинг. Максимальная энтропия бутстрэпинга для временного ряда доступна в *meboot.*

13.4.6. Декомпозиция и фильтрация - Decomposition and Filtering

Фильтры: *filters()* в статистиках обеспечивает авторегрессивную и скользящего среднего линейную фильтрацию многих временных рядов одной переменной. Пакет *robfilter* обеспечивает несколько устойчивых фильтров временного ряда, в то время как *mFilter* включает разные фильтры временного ряда, полезные для сглаживания и извлечения тренда и циклических компонентов.

Разложение: классическое разложение обеспечено через *decomposition(),* более усовершенствованное и гибкое разложение - доступно путем использования *stl(),* оба из основного пакета статистик.

Методы вейвлета: пакет вейвлетов включает вычислительные фильтры вейвлета, преобразования вейвлета и анализы мультиразрешения. Методы вейвлета для анализа временных рядов, основанного на Персивале и Уолдене (2000), даны в *wmtsa*. Дальнейшие методы вейвлета могут быть найдены в пакетах *rwt, waveslim* и *wavethresh.*

Разное:

signalextraction для экстракции сигнала в реальном времени (прямой подход фильтра). *bspec* для Байесового вывода на дискретном спектре мощности временного ряда. kza обеспечивает Колмогоровские-Zurbenko Адаптивные Фильтры включая обнаружение разрыва, спектральный анализ, вейвлеты и Фурье Трансформа KZ.

quantspec включает методы вычисления и рисования периодограммы Лапласа для одномерного временного ряда.

Rssa обеспечивает быструю реализацию Сингулярного Анализа Спектра для разложения временного ряда.

13.4.7. Стационарность, единичный корень и коинтеграция - Stationarity, Unit Roots, and Cointegration

Стационарность и единичные корни: *tseries* обеспечивает различные тесты стационарности и единичного корня включая Расширенный dickey-fuller, Phillips-Perron, и KPSS. Альтернативные реализации тестов ADF и KPSS находятся в urca пакете, который также включает дальнейшие методы, такие как Эллиот-Ротэнберг-Сток, Шмидт-Филлипс и тесты Зивот-Эндрюса. *fUnitRoots* пакет также обеспечивает тест Маккиннона.

CADFtest обеспечивает реализации тестов и стандартного ADF и увеличенного ковариантом ADF (CADF).

Коинтеграция: двух шаговый метод Энгл-Грейнджера с тестом коинтеграции Филлипса-Улиэриса реализован в *tseries* и *urca*. Последний дополнительно содержит функции для трассировки Джохэнсена и тестов максимальной лямбды.

13.4.8. Нелинейный анализ временных рядов -Nonlinear Time Series Analysis

Нелинейная авторегрессия: Различные виды нелинейной авторегрессии доступны в *tsDyn,* включая аддитивную AR, нейронные сети, SETAR и модели LSTAR. *bentcableAR* реализует авторегрессию Bent-Cable. BAYSTAR обеспечивает Байесов анализ пороговых авторегрессивных моделей.

Проект TISEAN обеспечил алгоритмы для анализа временных рядов из теории нелинейных динамических систем. *RTisean* обеспечивает интерфейс ***R*** для алгоритмов, и *tseriesChaos* обеспечивает реализацию ***R*** алгоритмов.

Тесты: Различные тесты на нелинейность обеспечены в fNonlinear.

13.4.9. Модели динамических регрессий - Dynamic Regression Models

Динамические линейные модели: удобный интерфейс для подгонки динамической модели регрессии с использованием OLS, что доступно в *dynlm*; улучшенный подход, который также работает с другими функциями регрессии и большим количеством классов временного ряда, реализован в *dyn*. *tslars* пакет применяет динамическую переменную процедуру отбора, используя расширение алгоритма LARS. Для более усовершенствованной подгонки уравнений динамических систем можно использовать *dse*. Для подгонки Гауссовских линейных моделей в пространстве состояний можно использовать *dlm* (через наибольшее правдоподобие, фильтрацию/сглаживание Кальмана и методы Bayesian).

Для подгонки моделей с изменяющимися во времени параметрами можно использовать *tpr* пакет.

Нелинейные модели с распределенным лагом обрабатываются через *dlnm* пакет.

13.4.10. Модели многомерных временных рядов - Multivariate Time Series Models

Векторная авторегрессивная модель (VAR) обеспечена через *ar()* в основном пакете статистик, включая выбор порядка через AIC. Эти модели ограничены стационарностью. Возможно, нестационарные модели VAR приспособлены в пакете *mAr*, который также позволяет модели VAR с основным компонентом пространства.

Более тщательно продуманные модели обеспечены в *vars* пакета, *estVARXls()* в *dse*, и Байесовский подход доступен в MSBVAR. *fastVAR* использует быстрые реализации, чтобы оценить модели VAR (возможно с экзогенными вводами и прореженными матрицами коэффициента).

Модели VARIMA и модели в пространстве состояний обеспечены в *dse* пакете. *EvalEst* облегчает эксперименты Монте-Карло, чтобы оценить присоединенные методы оценки.

Векторные модели коррекции ошибок доступны через пакеты *urca* и *vars*, включая версии со структурными ограничениями.

Факторный анализ временного ряда обеспечен в *tsfa*.

Многомерные модели в пространстве состояний реализованы в пакете *FKF* (Быстрый фильтр Кальмана). Он обеспечивает относительно гибкие модели в пространстве состояний через *fkf()* комментарий: параметрам пространства состояний позволяют быть изменяющимися во времени, и смещения включены в оба уравнения. Альтернативная реализация обеспечена пакетом *KFAS*, который обеспечивает быстрый многомерный фильтр Кальмана, более сглаживание, имитация сглаживания и прогноза. Еще одна реализация дана в *dlm* пакете, который также содержит инструменты для того, чтобы преобразовать другие многомерные модели в форму пространства состояний.

MARSS подгоняет ограниченные и неограниченные многомерные авторегрессивные модели в пространстве состояний, используя алгоритм EM. Все четыре пакета предполагают некоррелированность параметров наблюдений и ошибок состояний.

Частично наблюдаемые процессы Маркова - обобщение обычных линейных многомерных моделей в пространстве состояний, позволяя негауссовы и нелинейные модели. Они реализованы в пакете *pomp*.

13.4.11. Модели непрерывного времени - Continuous time models

Авторегрессивное моделирование непрерывного времени обеспечено в cts.

Оценка модели ARMA непрерывного времени и имитация доступна в ctarma.

13.4.12. Исходные временные ряды - Time Series Data

Data from Makridakis, Wheelwright and Hyndman (1998) Forecasting: methods and applications are provided in the fma package.

Data from Hyndman, Koehler, Ord and Snyder (2008) Forecasting with exponential smoothing are in the expsmooth package.

Data from the M-competition and M3-competition are provided in the Mcomp package.

Data from Tsay (2005) Analysis of financial time series are in the FinTS package, along with some functions and script files required to work some of the examples.

TSdbi provides a common interface to time series databases.

fame provides an interface for FAME time series databases

VhayuR provides an interface to he Vhayu Velocity time series database

AER and Ecdat both contain many data sets (including time series data) from many econometrics text books

13.4.13. Разное - Miscellaneous

dtw: Динамические алгоритмы деформирования времени для вычислений и рисования попарное выравнивание между временным рядом.

fractal: Фрактальное моделирование временного ряда и анализ.

fractalrock: Генерация фрактального временного ряда с ненормальным распределением приращений.

GeneCycle и GeneNet: временной ряд микромассива и сетевой анализ.

pastecs: Регулирование, разложение и анализ пространственно-временного ряда.

ptw: Параметрическое время, деформируясь.

sde: Имитация и вывод для стохастических дифференциальных уравнений.

tiger: определение и визуализация временно разрешенных групп типичных разностей (ошибки) между двумя временными рядами.

wavethresh: Локально стационарные модели вейвлета для нестационарного временного ряда (включая функции оценки, рисования и имитации для изменяющихся во времени спектров).

13.5. Перечень пакетов для анализа временных рядов:

• **AER**	• **afmtools**	• **bayesGARCH**
• **BAYSTAR**	• bentcableAR	• boot
• **BootPR**	• bspec	• CADFtest
• **chron**	• ctarma	• cts
• **deseasonalize**	• dlm	• dlnm

• **dse**	• dtw	• dyn
• **dynlm**	• Ecdat	• ensembleBMA
• **EvalEst**	• expsmooth	• fame
• **fArma**	• fastVAR	• fGarch
• **FinTS**	• FitAR	• FitARMA
• **FKF**	• fma	• fNonlinear
• **forecast** (core)	• fracdiff	• fractal
• **fractalrock**	• fts	• fUnitRoots
• **GeneCycle**	• GeneNet	• gogarch
• **gsarima**	• hydroGOF	• hydroTSM
• **Interpol.T**	• its	• KFAS
• **kza**	• ltsa	• lubridate
• **mAr**	• mar1s	• MARSS
• **Mcomp**	• meboot	• mFilter
• **mondate**	• MSBVAR	• paleoTS
• **pastecs**	• pear	• pomp
• **ptw**	• quantspec	• RMAWGEN
• **robfilter**	• RSEIS	• Rssa
• **RTisean**	• rtv	• rwt
• **sde**	• season	• signalextraction
• **tempdisagg**	• tiger	• timeDate
• **timeSeries**	• timsac	• tis
• **tpr**	• TSAgg	• TSdbi
• **tsDyn**	• tseries (core)	• tseriesChaos

• **tsfa**	• tslars	• tsModel
• **urca**	• vars	• VhayuR
• **wavelets**	• waveslim	• wavethresh
• **wmtsa**	• wq	• xts
• **zoo** (core)		

Приложение A. Примерный сеанс

Следующий сеанс предназначен для демонстрации некоторых функций среды ***R*** через их применение. Много функций системы будут незнакомыми и озадачивающими сначала, но это замешательство скоро исчезнет.

Запустите ***R*** соответственно для Вашей платформы (см. Приложение B [Вызов ***R***]).

Программа ***R*** начинается с баннером.

(В пределах кода ***R*** во избегании путаницы приглашение слева не будет показываться).

help.start()

Стартует HTML интерфейс к системе интерактивной помощи (с помощью браузера имеющегося на вашей машине). Вы должны вкратце изучить возможности этой системы с помощью мыши.

Сверните окно с помощью и переходите к следующей части.

x <- rnorm(50)

y <- rnorm(x)

Создает два псевдослучайные нормальных векторов с ***x*** и ***y*** - координатами.

plot(x, y)	Рисует точки на плоскости. Автоматически появится окно графического вывода.
ls()	Показывает ***R*** объекты, которые в настоящее время находятся в рабочем пространстве R.
rm(x, y)	Удаляет не нужные объекты. (Очистка).
x <- 1:20	Создает x = (1, 2, ..., 20).
w <- 1 + sqrt(x)/2	"Весовой" вектор стандартных отклонений.
*dummy <- data.frame(x=x, y= x + rnorm(x)*w) dummy*	Сделать таблицу данных из двух столбцов ***x*** и ***y*** и вывести ее.
fm <- lm(y ~ x, data=dummy) summary(fm)	Подогнать простую линейную регрессию ***y*** по ***x*** и вывести результат анализа.
fm1 <- lm(y ~ x, data=dummy, weight=1/w^2) summary(fm1)	Поскольку мы знаем стандартные отклонения, мы можем подогнать взвешенную регрессию.
attach(dummy)	Сделать столбцы в таблице данных видимыми в качестве переменных.
lrf <- lowess(x, y)	Вычислить функцию непараметрической локальной регрессии.
plot(x, y)	Стандартный рисунок точек.
lines(x, lrf$y)	Добавить в него локальную регрессию.
abline(0, 1, lty=3)	Истинная линия регрессии: (отсекающий отрезок 0, наклон 1).
abline(coef(fm))	Невзвешенная линия регрессии.

abline(coef(fm1), col = "red")

Взвешенная линия регрессии.

detach() Удалить таблицу данных из пути поиска.

plot(fitted(fm), resid(fm), xlab="Fitted values", ylab="Residuals", main="Residuals vs Fitted")

Стандартный диагностический график регрессии для проверки на гетероскедастичность. Вы ее видите?

qqnorm(resid(fm), main="Residuals Rankit Plot")

График нормальных значений для проверки асимметрии, эксцесса и выпадающих значений. (Не очень полезен в данном случае).

rm(fm, fm1, lrf, x, dummy) Снова очистка.

В следующем разделе будут проанализированы данные из классического эксперимента Michaelson и Morley для измерения скорости света. Этот набор данных доступен в объекте *morley*, и мы считаем ее, чтобы проиллюстрировать функцию *read.table*.

filepath <- system.file("data", "morley.tab" , package="datasets") filepath

Записать путь к текущему файлу данных.

file.show(filepath) Не обязательно: посмотреть на файл.

mm <- read.table(filepath) mm

Читает данные Michaelson и Morley, как таблицу данных, и выводит их. Здесь пять, надлежащим образом закодированных, экспериментов (столбец Expt), каждый имеет 20 прогонов (столбец Run), а sl является измеренной скоростью света.

mm$Expt <- factor(mm$Expt) mm$Run <- factor(mm$Run)

Трансформация Expt и Run в факторы.

attach(mm) Сделать таблицу данных видной в позиции 3 (по умолчанию).

plot(Expt, Speed, main="Speed of Light Data", xlab="Experiment No.")

Сравнение пяти экспериментов простыми коробочными диаграммами.

fm <- aov(Speed ~ Run + Expt, data=mm) summary(fm)

Анализировать, как рандомизированный блок, с "runs" и "experiments" в качестве факторов.

fm0 <- update(fm, . ~ . - Run) anova(fm0, fm)

Подогнать к субмодель опуская "runs", и сравнить с помощью формального дисперсионного анализа.

detach() rm(fm, fm0) Почистим перед дальнейшей работой.

Сейчас посмотрим на некоторые дальнейшие графические возможности: контурные графики и графики изображений.

x <- seq(-pi, pi, len=50) y <- x

x это вектор из 50 равномерно распределенных значений в интервале *[-pi\, pi]*. ***y*** такой же вектор.

f <- outer(x, y, function(x, y) cos(y)/(1 + x^2))

f - это квадратная матрица, причем строки и столбцы индексированы как ***x*** и ***y***, соответственно, состоит из значений функции *cos(y)/(1 + x^2).*

oldpar <- par(no.readonly = TRUE) par(pty="s")

Сохраняем параметры графического вывода и устанавливаем "квадратный" регион вывода графики.

contour(x, y, f) contour(x, y, f, nlevels=15, add=TRUE) Делаем контурную карту ***f;*** добавляем побольше линий для детальности изображения.

fa <- (f-t(f))/2

fa - это "асимметричная часть" ***f***. (***t()*** - это транспонирование). *contour(x, y, fa, nlevels=15)*

Делаем контурный рисунок, ...

par(oldpar)

... и восстанавливаем старые графические параметры.

image(x, y, f) image(x, y, fa)

Делаем пару высоко четких графиков-изображений, (из которых можно получить, если нужно, распечатку)

rm (x, y, f, fa)

... и очищаем перед дальнейшей работой.

R имеет комплексную арифметику:

*th <- seq(-pi, pi, len=100) z <- exp(1i*th)*

1i используется для комплексного числа ***i.***

par(pty="s")

plot(z, type="l")

Вывод комплексных аргументов означает график мнимой части числа по реальной. Это должен быть круг.

*w <- rnorm(100) + rnorm(100)*1i*

Предположим, нам нужны точки в пределах единичной окружности. Один из способов, это получить комплексные числа со стандартно нормально распределенной реальной и мнимой частями ...

w <- ifelse(Mod(w) > 1, 1/w, w)

... и отобразить все выходящие за пределы круга на их обратное значение.

plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+",xlab="x", ylab="y")

lines(z)

Все точки внутри единичной окружности, но распределение не является равномерным.

*w <- sqrt(runif(100))*exp(2*pi*runif(100)*1i)*

plot(w, xlim=c(-1,1), ylim=c(-1,1), pch="+", xlab="x", ylab="y")

lines(z)

Второй метод используется равномерное распределение. Точки должны теперь выглядеть более равномерно распределенными по всему кругу.

rm(th, w, z)

Снова уборка.

q()

Выход из ***R*** программы. Будет задан вопрос, хотите ли сохранить рабочее пространство ***R,*** и для такой учебной сессии как эта, это наверное не понадобится.

Приложения B. Вызов *R*

Пользователи ***R*** на Windows или Mac OS X сначала должны прочесть соответствующий для ОС раздел, но также поддерживается использование командной строки.

B.1. Вызов *R* из командной строки

Работая в командной строке под UNIX или Windows, команда '***R***' может использоваться как для запуска основной программы ***R*** в форме:

R *[опции] [<infile] [> outfile],*

так и через интерфейс «R CMD», как обертка к различным инструментам ***R*** (например, для обработки файлов в формате документации ***R*** или управления дополнительными пакетами), которые не предназначены для "непосредственного" вызова.

В командной строке Windows *rterm.exe* предпочтительнее ***R.***

Следует убедиться, что или переменная окружения *TMPDIR* сброшена, или она указывает на допустимое место для создания временных файлов и каталогов.

Большинство опций управляет началом и окончанием сеанса ***R.*** Механизм запуска состоит в следующем (см. также онлайновую справку для темы 'Запуск' для получения дополнительной информации, и раздел ниже для некоторых специфичных для Windows деталей).

- Пока не задан *'--no-environ'* ***R*** ищет файлы пользователя и системы для обработки установок переменных окружения. Имя системного файла указано в переменной окружения *R_ENVIRON*; если она сброшена, то используется *'R_HOME/etc/renviron.site'* (если он существует). Пользовательский файл с настройками ищется как *R_ENVIRON_USER*, если он установлена; иначе разыскиваются файлы *'.Renviron'* в текущем или в корневом каталоге пользователя (в этом порядке). Эти файлы должны содержать строки вида *'name=value'*. См. справку "Запуск" для точного описания. Переменные, которые можно установить, включают *R_PAPERSIZE* (размер бумаги по умолчанию), *R_PRINTCMD* (команда печати по умолчанию) и *R_LIBS* (указывает список деревьев библиотеки ***R*** для поиска дополнительных пакетов).
- Затем ***R*** ищет системный загрузочный профиль, если был задан параметр командной строки *'--no-site-file*'. Имя этого файла взято из значения переменной окружения *R_PROFILE*. Если эта переменная сброшена, то используется по умолчанию *'R_HOME/etc/rprofile.site'*, если он существует.
- Затем, если *'--no-init-file'* был дан, ***R*** ищет профиль пользователя и определяет его источник. Имя этого файла взято из переменной окружения *V_PROFILE_USER;* если сброшено, то разыскивается файл с именем *'.Rprofile'* в текущем каталоге или в корневом каталоге пользователя (в этом порядке).
- Также загружается сохраненная рабочая область из файла *'.RData'* в текущем каталоге, если такой существует (пока не указано *'--no-restore'* или *'--no-restore-data'*).
- Наконец, если функция *.First()* существует, то она выполняется. Эта функция (так же как *.Last()*, которая выполняется в конце сеанса ***R)*** может быть определена в соответствующих профилях запуска, или находиться в *'.RData'.*

Кроме того, есть опции для управления памятью, доступная процессу ***R*** (см. онлайновую справку для темы 'Память' для получения дополнительной информации). Пользователям не следует их обычно использовать, если они не пытаются ограничить объем памяти, используемый ***R.***

R принимает следующие параметры командной строки.

‘*--help*’

‘*-h*’

Печатает короткое сообщение помощи и успешно выходит.

‘*--version*’

Печатает информацию о версии и успешно выходит.

‘*--encoding=enc*’

Укажите кодирование, которое будет использоваться для ввода с консоли или *stdin*. Это должно быть кодированием, известным *iconv*: см. страницу справки. (*--encoding=enc*’ также принимается.) Входные данные перекодируются и исполняются в локали *R* и необходимо представить в последующем кодировании (итак, нельзя перекодировать греческий текст во французскую локаль, пока локаль использует кодировку UTF-8).

‘*RHOME*’

Напечатайте путь к "корневому каталогу" ***R*** к стандартному выводу и выходу успешно. Кроме скрипта оболочки и страницы справочника, установка ***R*** помещает все (исполнимые программы, пакеты, и т.д.) в этот каталог.

‘*--save*’

‘*--no-save*’

Управляет, должны ли наборы данных быть сохранены или нет в конце сеанса ***R.*** Если не указано в интерактивном сеансе, пользователя спросят относительно требуемого режима при окончании с *q()*; в не интерактивном использовании один из них должен указываться или подразумеваться некоторой другой опцией (см. ниже).

‘*--no-environ*’

Не читать какой-либо пользовательский файл для установки среды

‘*--no-site-file*’

Не читать профиль с сайта при старте.

‘*--no-init-file*’

Не читать профиль пользователя при старте.

‘*--restore*’

‘*--no-restore*’

‘*--no-restore-data*’

Управляет, должен ли быть сохраненный образ восстановлен при старте или нет (файл ‘*.RData*’ в каталоге, где ***R*** был запущен). По умолчанию должен восстанавливаться. ‘*--no-restore*’ подразумевает все указанные опции ‘*--no-restore-**’.

‘*--no-restore-history*’

Управление, должен ли файл истории быть восстановлен при старте или нет. Обычно файл *'.Rhistory'* находится в каталоге, из которого ***R*** был запущен, но может быть установлен переменной окружения *R_HISTFILE* По умолчанию следует восстановить.

'*--no-rconsole'*

(Только Windows). Блокирует загрузку файла *'rconsole'* при старте.

'*--vanilla'*

Комбинирует *'--no-save', '--no-environ', '--no-site-file', '--no-init-file'* и *'--no-restore'*. Под Windows также включает *'--no-rconsole'*.

'*-f file'*

'*--file=file'*

(не *rgui.exe*). Берут ввод из файла: '-' означающий *stdin*. Подразумевают *'--no-save'*, пока не установлено *'--save'*.

'*-e expression'*

(не *rgui.exe*). Используется *expression* как входную строку. Может использоваться одна или более опций ***'-e'***, но не вместе с ***'-f'*** или *'--file'*. Подразумевают *'--no-save'*, пока не установлено *'--save'*. Есть предел 10 000 байтов на полной длине выражений, используемых таким образом. Выражения, содержащие пробелы или метасимволы оболочки, следует заключить в кавычки.

'*--max-ppsize=N'*

Указывает максимальный размер стека защиты указателя как N расположения. По умолчанию равно 10000, но может быть увеличен, что позволит делать более сложные вычисления. В настоящий момент максимальное принятое значение равно 100000.

'*--max-mem-size=N'*

(Windows только). Указывает предел для объема памяти, который будет использоваться и для объектов ***R*** и для рабочих областей. Установлено по умолчанию в меньшее из количества физической RAM в машине для 32-разрядного ***R*** - 1.5Gb, и должно быть между 32 МБ и максимумом, позволенным на версии Windows.

'*--quiet'*

'*--silent'*

'*-q'*

Не печатать приветствие

'*--slave'*

Сделать исполнение ***R*** настолько спокойно насколько возможно. Эта опция предназначена к программам поддержки, которые используют ***R,*** чтобы вычислить результаты для них. Подразумевается *'--quiet'* и *'--no-save'*.

'*--ess'*

(Windows только). Установить *RteRm* для использования *R-inferior-mode* в **ESS**, включая утверждение интерактивного использования без редактора командной строки.

'*--args'*

Этот флаг не делает ничего кроме как вызывает пропуск последующих команд: это может быть полезно, чтобы получить значение от них с *commandArgs(TRUE)*.

Заметим, что ввод и вывод может быть перенаправлен обычным способом (используя ‘ <’ и ‘>’), но предел длины строки 4095 байтов все еще, применяется. Предупреждающие сообщения и сообщения об ошибках отправлены по каналу ошибок *(stderr)*.

Команды ***R*** CMD позволяют вызов различных инструментов, которые полезны совместно с ***R,*** но не предназначенных для "непосредственного" вызова. Общая форма:

R CMD command args

где *command* является именем средства, а *args* является аргументом, который ему передается.

В настоящее время доступны команды:

BATCH	Исполняет ***R*** в пакетном режиме. Исполняет ***R*** *--restore --save* с возможностью в дальнейшем дополнительных опций (смотри *?BATCH*).
SHLIB	Строит разделяемую библиотеку для динамической загрузки.
INSTALL	Устанавливает встроенные пакеты.
REMOVE	Удаляет встроенные пакеты.
build	Строит (т.е. пакет) встроенные пакеты.
check	Проверяет встроенные пакеты.
RpRof	После обработка файлов профиля ***R***.
r*dconv*	
rd2txt	Преобразует формат ***rd*** в разные другие форматы, включая HTML, LaTEX, простые тексты, и извлекает примеры. *rd2txt* может быть использован как краткая форма для *rd2conv -t txt*.
rd2pdf	Преобразует формат ***rd*** в PDF.
Stangle	Извлекает код ***S/R*** сиз документации Sweaven
Sweave	Обрабатывает документацию Sweave
rdiff	Отделяет выход ***R*** от заголовков
config	Получает информацию о конфигурации
open	(Только Windows) Открывает файл через ассоциацию файлов Windows
texify	(Только Windows) Обрабатывает файлы (La)TeX files в стиле файлов ***R***

Используй:

R *CMD command –help*

чтобы получить использование информации для каждого средства, доступного через интерфейс ***R*** *CMD*.

Кроме того, можно использовать *опции '-arch =', '--no-environ', '--no-init-file', '--no-site-file' and '--vanilla'* между ***R*** и *CMD*: они влияют на любые процессы ***R,*** исполняемые инструментами. (Здесь *'--vanilla'* эквивалентна *'--no-environ --no-site-file --no-init-file').*) Однако отметьте, что ***R*** *CMD* сама по себе не использует файлы запуска ***R*** (в частности ни пользовательские, ни файлы сайта *'renviron'*), и все процессы ***R,*** исполняемые этими инструментами, (кроме *BATCH*) используют *'--no-restore'*. Большинство используют *'--vanilla'* и так стартуют без файлов запуска ***R:*** текущие исключения - *INSTALL, REMOVE, Sweave* и *SHLIB (*где используются *'--no-site-file --no-init-file').*

R CMD cmd args

для любой другой исполнимой программы *cmd* на пути или данном абсолютном пути файла: это полезно, чтобы иметь ту же самую среду как ***R*** или определенные команды, под которыми исполняют, например, исполнить *ldd* или *pdflatex*. Под Windows *cmd* может быть исполнимой программой или пакетным файлом, или если у него расширение *.sh* или *.pl* с соответствующим интерпретатором (при наличии).

B.2. Вызов *R* под Windows

Есть два способа исполнить ***R*** под Windows. В пределах окна терминала (например, *cmd.exe* или более пригодная оболочка), методы, описанные в предыдущем разделе, могут использоваться для вызова R.exe или более непосредственно *Rterm.exe*. Для интерактивного использования (*Rgui.exe*) есть основанный на консоли GUI.

Процедура запуска под Windows подобна под UNIX, но ссылки на 'корневой каталог' должны быть разъяснены, поскольку это не всегда определяется на Windows. Если переменная окружения *R_USER* определена, то она дает корневой каталог. Затем, если переменная окружения *HOME* определена, то она дает корневой каталог. После этих двух управляемых пользователем настроек ***R*** пытается найти, что система определила корневые каталоги. Сначала пытается использовать Windows "персональный" каталог (обычно *C:\Documents and Settings\user name\My Documents* в Windows XP). Если это перестало работать, и переменные окружения *HOMEDRIVE* и *HOMEPATH* определены (а так и есть), то они определяют корневой каталог. При отсутствии берется корневой каталог в качества каталога запуска.

Следует гарантировать, что любая переменная окружения *TMPDIR, TMP* и *TEMP* или сброшена или одна из них указывает на допустимое место для создания временных файлов и каталогов.

Переменные окружения могут быть предоставлены как пары '*name=value*' в командной строке.

Если есть параметр, заканчивающийся '*.RData'*, (в любом случае) он интерпретируется как путь к рабочей области, который будет восстановлен: это подразумевает '*--restore*' и устанавливает рабочий каталог в родителя именованного файла. Этот механизм используется для перетаскивать-и-отбрасывать и зависимости файла с *RGui.exe*, но также и работает в *Rterm.exe*. Если именованный файл не существует, он устанавливает рабочий каталог, если родительский каталог существует.

Следующие дополнительные параметры командной строки доступны при вызове *RGui.exe*.

'--mdi'

'--sdi'

'--no-mdi'

Управляет, будет ли *Rgui* работать как программа MDI (со множественными дочерними окнами в пределах одного главного окна) или применение SDI (со множественными высокоуровневыми окнами для консоли, графики и страниц). Установка командной строки переопределяет установку в файле '*Rconsole*' пользователя.

'--debug'

Включает пункт меню *"Break to debugger"* в *Rgui*, и инициирует переход к отладчику во время обработки командной строки.

Под Windows с ***R*** *CMD* также может специфицировать свои собственные файлы *'.bat', '.exe', '.sh'* и *'.pl'*. Они исполнятся под соответствующим интерпретатором (Perl для *'.pl'*) с несколькими наборами переменных окружения соответственно, включая *R_HOME, R_OSTYPE, ПУТЬ, BSTINPUTS* и *TEXINPUTS*. Например, если уже есть '*latex.exe*' в пути, то:

R *CMD latex.exe mydoc*

то будет исполнен LATEX для '*mydoc.tex*', с путем для макроса ***R*** '*share/texmf*', присоединенном к *TEXINPUTS*. К несчатью это не поможет с построением *MiKTeX LATEX*, но для ***R*** CMD *texify mydoc* будет работать.

B.3. Вызов *R* под OS X

Есть два способа исполнить ***R*** под OS X. В пределах окна Terminal.app вызовом ***R*** применяются методы, описанные в первом подразделе. Есть также основанный на консоли GUI (R.app), который по умолчанию установлен в папке Применений на Вашей системе. Это - стандартное применение OS X.

Процедура запуска под OS X подобна под UNIX. 'Корневой каталог' - одна внутренняя часть, R.framework, но запуск и текущий рабочий каталог установлены как корневой каталог пользователя, если другой каталог запуска не дан в Привилегированном окне, доступном изнутри GUI.

B.4. Скрипты *R*

Если только необходимо исполнить команду ***R*** *'foo.**R** '*, рекомендуемый путь состоит в том, чтобы использовать ***R*** *CMD BATCH foo.**R**.* Если необходимо исполнить его в фоновом режиме или как использование пакетного задания специфичные для ОС средства, чтобы сделать так: например в большинстве оболочек на Unix подобных ОС ***R*** *CMD BATCH foo.* ***R*** *&* исполняет0ся низкоприоритетным заданием.

Можно передать параметры скриптам через дополнительные аргументы в командной строке: например (где необходимо точное заключение в кавычки будет зависеть от используемой оболочки):

R CMD BATCH "--args arg1 arg2" foo.R&

передаст параметры скрипту, который может быть получен как символьный вектор

args <-commandArgs (TRUE)

Можно сделать проще альтернативным *Rscript*, который может быть вызван:

Rscript foo. rarg1 arg2

и это может также использоваться, чтобы записать исполнимые файлы скрипта как (по крайней мере, на Unix-подобных ОС, и в некоторых оболочках Windows)

```
#! /path/to/rscript
args <- commandArgs(TRUE)
...
```

q(status=<exit status code>)

Если это вводится в текстовый файл ‘*RUnfoo*’, и это сделано исполнимой программой (*chmod 755 RUnfoo),* это может быть вызвано для различных параметров

runfoo arg1 arg2

Для дальнейших опций смотри *help("rscript").* Результат будет записан ***R*** в ‘*stdout*’ и ‘*stderr*’, и это может быть перенаправлено обычным способом к оболочке, исполняющей командой.

Если не желателен установленный путь к *rscript,* то в пути можно исользовать (что обычно имеет место для установленного ***R*** за исключением Windows, но например, пользователи Mac OS X, возможно, должны прибавить ‘/usr/local/bin’ к их пути):

#!/usr/bin/env rscript

...

По крайней мере, в Bourne #! механизм не позволяет дополнительные параметры, такие как *#!/usr/bin/env rscript --vanilla.*

Рассмотрим еще, к чему обращается *stdin().* Это банально, чтобы записать скрипты ***R*** с сегментами как:

chem <- scan(n=24)

2.90 3.10 3.40 3.40 3.70 3.70 2.80 2.50 2.40 2.40 2.70 2.20

5.28 3.37 3.03 3.03 28.95 3.77 3.40 2.20 3.50 3.60 3.70 3.70

и *stdin()* обращается к файлу скрипта, чтобы позволить такое традиционное использование. Если необходимо обратиться к процессу *‘stdin’,* то используйте *"stdin"* в качестве соединения *file*, например, *scan*("*stdin*"...).

Другой способ записать исполнимые файлы скрипта (предложено Francois Pinard) состоит в том, чтобы использовать *здесь документ* как:

#!/bin/sh

[environment variables can be set here]

R *--slave [otheroptions] <<EOF*

rprogram goes here...

EOF

но здесь *stdin()* обращается к источнику программы, и "*stdin"* не будет применим.

Очень короткие скрипты можно передать к *Rscript* в командной строке через флаг ***‘-e’***.

Заметим, что на Unix-подобных ОС входное имя файла (такое как ‘*foo.R*’), не должно содержать ни метасимволы оболочки, ни пробелы.

Приложение С. Редактор командной строки

Существует следующие варианты редактирования исходного кода R:

- для консоли R
- для скриптов R
- внешнего текстового редактора

При вызове RGui.exe открывается окно консоли ***R,*** в которой можно поместить любую последовательность команд, которые немедленно исполняются при распознании завершенности оператора языка R. Имеющийся в консоли текст может быть отредактирован. Возможности редактирования изложены в «Справка» в закладке «Консоль».

Любой новый текст на языке ***R*** может быть набран или отредактирован имеющийся с использованием понятия «Скрипт» из закладки «Файл». Открывается окно с название «Редактор R». Если это окно активно, то в «Справка» имеется закладка «Редактор», в которой изложена инструкция по использованию этого редактора.

Особенностью редактора для скриптов является возможность исполнения отдельного оператора или группы операторов по мере необходимости.

Оба указанные редакторы имеют крайне ограниченные возможности по редактированию и инструкции по ним минимальны. Поэтому здесь не будут излагаться, так как ими удобно воспользоваться при конкретной работе.

Среди сторонних редакторов следует Notepad++, который распознает программные тексты, написанные на ***R***. Имеет широкие возможности по написанию и редактированию кода на ***R.***

Приложение F. Ссылки

D. M. Bates and D. G. Watts (1988), Nonlinear regression Analysis and Its Applications. John Wiley & Sons, New York.

richard A. Becker, John M. Chambers and Allan r. Wilks (1988), The New S Language. Chapman & Hall, New York. This book is often called the “Blue Book”.

John M. Chambers and TrevorJ. Hastie eds. (1992), Statistical Models in S. Chapman & Hall, New York. This is also called the “White Book”.

John M. Chambers (1998) Programming with Data. Springer, New York. This is also called the “Green Book”.

A. C. Davison and D. V. Hinkley (1997), Bootstrap Methods and TheirApplications, Cambridge University Press.

Annette J. Dobson (1990), An Introduction to Generalized LinearModels, Chapman and Hall, London.

PeterMcCullagh and John A. Nelder(1989), Generalized LinearModels. Second edition, Chapman and Hall, London.

John A. rice (1995), Mathematical Statistics and Data Analysis. Second edition. Duxbury Press, Belmont, CA.

S. D. Silvey (1970), Statistical Inference.