

Методические указания по дисциплине

Разработка интеллектуальных систем на языке R

1.1. Установка, запуск и элементарные команды для среды R. Работа в консоли и в графической оболочке

Задание

Установить и запустить R в режиме терминала и GUI. Запустить все команды из разделов 1.1.1 и 1.1.2.

1.1.1. Установка и запуск R

R можно загрузить с сайта <http://cran.r-project.org/>. Предварительно скомпилированные загрузочные файлы доступны для Linux, Mac OS X, Windows. Функциональность программы можно расширять с помощью пакетов, они также доступны с сайта CRAN (Comprehensive R Archive Network). У этого сайта довольно много зеркал (копий), так что выбирайте подходящее. Скачать можно как исходный код для последующей компиляции, так и собранный бинарный пакет (последнее, не всегда возможно). Исходный код R компилируется как правило без проблем.

Linux. Если у вас какой-нибудь из распространенных дистрибутивов, то R наверняка входит в репозиторий прилагающихся к вашей системе пакетов. Единственное, что нужно учесть, - обновление пакетов часто отстает от выхода версий R (как правило, четыре раза в год).

Версия нумеруется тремя числами, первые два - это главная версия, обновляется два раза в год. С каждой главной версией в R вносятся изменения, часто довольно значительные. Как правило, это множество новых команд, исправленные алгоритмы выполнения старых, и разумеется, исправления ошибок. Кроме того, бывает так, что страдает обратная совместимость - новые версии начинают иначе истолковывать команды, а иногда команды или их опции перестают действовать. Естественно, разработчики стараются минимизировать такие изменения. В общем, рациональный подход такой: обновляйте R, но при этом всегда читайте список изменений версии. На каждую главную версию выходят, как правило, две минорные версии (нулевая и первая). Первая минорная версия обычно ничего нового, кроме исправления ошибок, не содержит.

Mac OS X. Для того чтобы начать работать с R в Mac OS X, надо сначала убедиться, что у вас последняя версия этой операционной системы.

Последние версии R выходят только под последние версии Mac OS. Установочный пакет скачивается с CRAN (не забудьте также скачать оттуда Tcl/Tk). Имейте в виду, что графическая оболочка R для Mac (R Mac GUI) обновляется быстрее, чем сам R, поэтому разработчики предусмотрели возможность скачивания и установки оболочки отдельно. Установка происходит практически так же, как установка любой программы под Mac. Надо еще отметить, что R завоевал себе популярность не в последнюю очередь тем, что он запускался под Mac, в то время как S-Plus под эту платформу был недоступен.

Windows. Для того чтобы запускать R, можно иметь любые версии этой операционной системы, начиная с Windows 95. Как и в предыдущем случае, установочный пакет скачивается с CRAN. Установка напоминает обычную установку приложения для Windows. Есть программа-установщик, которая задает несколько простых вопросов, от ответов на которые ничего серьезного не зависит. После инсталляции появляется ярлык, щелкнув на который, можно запускать R.

Здесь интересно заметить, что Windows-инсталляция R является portable и может запускаться, например, с флэшки или лазерного диска. R делает пару записей в реестр, но для работы они совершенно не критичны. Единственное, что надо при этом иметь в виду, рабочая папка должна быть открыта для записи, иначе вам некуда будет записывать результаты работы. Еще одна вещь, важная для всех операционных систем: R (в отличие от S-Plus) держит все свои вычисления в оперативной памяти, поэтому если в процессе работы, скажем, выключится питание, результаты сессии, не записанные явным образом в файл, пропадут. (Правда, существует пакет SOAR, который изменяет это поведение.)

Каждая операционная система имеет свои особенности работы. Но в целом можно сказать, что под все три упомянутые выше операционные системы существует так называемый терминальный способ запуска, а под Mac и Windows имеется свой графический интерфейс (GUI) с некоторыми дополнительными возможностями (разными в разных операционных системах).

Терминальный способ прост: достаточно в командной строке терминала набрать в командной строки, например, Debian:

```
$ R, -
```

и появится вводный экран системы

```
R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> b <- matrix(1:9, ncol=3)
> print(b)
      [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
>
```

Набираем пример с матрицей.

```
> b <- matrix(1:9, ncol=3)
```

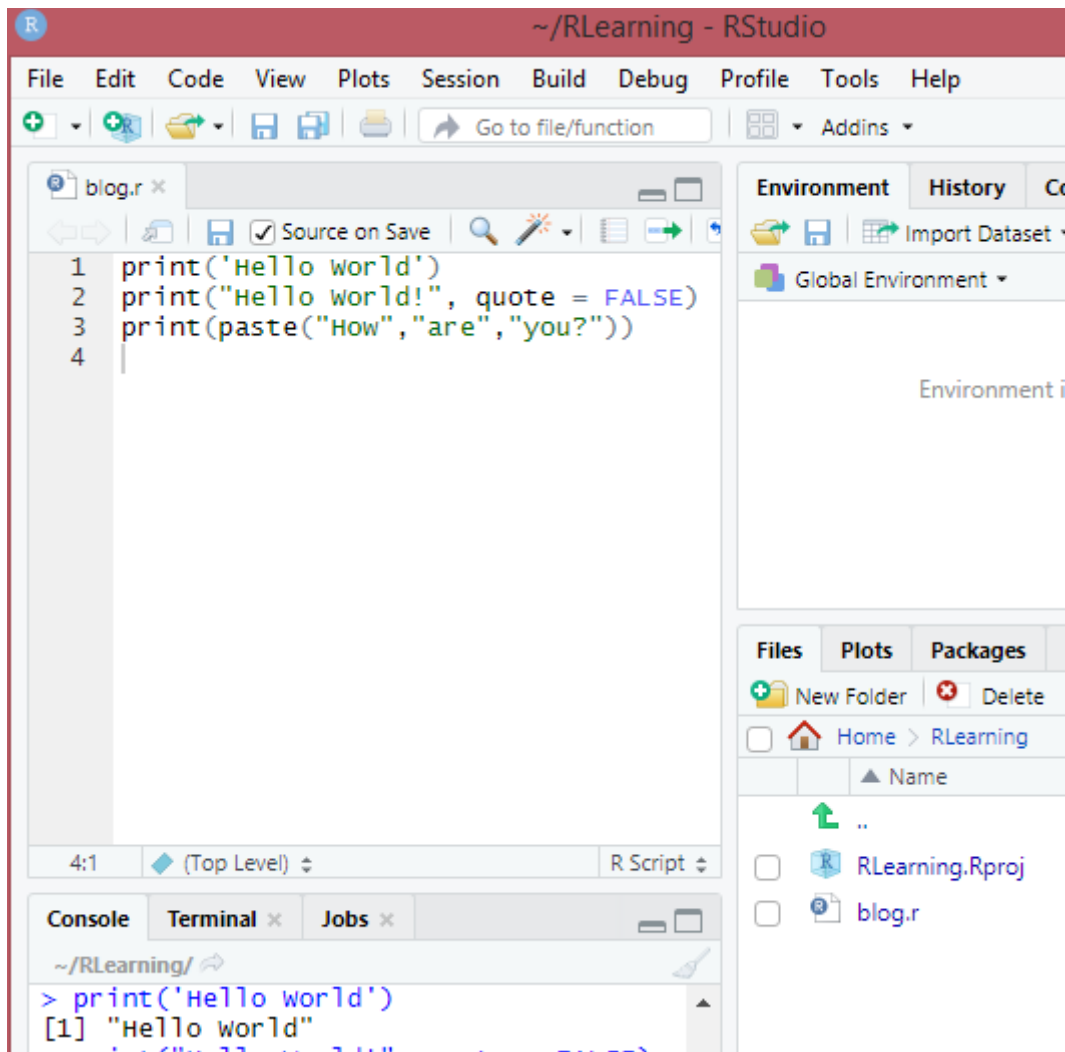
и выводим результат переменной `b` командой `print`.

```
> print(b)
```

Под Windows это немного сложнее необходимо вызывать программу `rgui.exe`, причем для этого надо либо находиться в той же папке, где находится эта программа, либо путь к этой папке должен быть прописан в переменной `PATH`. Кроме того, для того чтобы в русскоязычной Windows вводный экран был читаем, надо сменить в окне терминала кодировку (командой `chcp1251`) и установить соответствующий шрифт (скажем, `Lucida Console`). Если в UNIX терминал запущен без графической среды (X11), то все изображения будут скидываться в один многостраничный PDF-файл `Rplots.pdf`. Под Mac это будет происходить, даже если X11 запущен, так что полноценно использовать R под Mac можно только в GUI-варианте. Терминальный запуск под Windows таких ограничений не имеет.

В дальнейшем договоримся, что под сессией R будет иметься в виду терминальный запуск под X11 в Linux и GUI-запуск в Windows и Mac. GUI под эти операционные системы построены так, что они все равно запускают

терминал-подобное окно - общение с R возможно только в режиме диалога, "команда - ответ" . Полноценного GUI с R не поставляется, хотя существуют многочисленные попытки создать такую систему. Одна из лучших, на наш взгляд, это - R Commander, советуем также обратить внимание на RStudio.



По ссылке <https://rstudio.com/products/rstudio/download/> можно скачать RStudio (справочная информация по установке RStudio <https://rstudio.com/products/rstudio/#rstudio-desktop>).

Но система из меню и окошек не способна заменить полноценного интерфейса командной строки, особенно в случае таких сложных систем, как R. Особенность R GUI под Windows – ее можно запустить в много-(MDI) и однооконном (SDI) режимах. Рекомендуется применять второй, тем более что все остальные реализации R только его и "умеют".

После запуска R в начале появляется вводный экран. Если у Вас русифицированная система, то Вы увидите надпись по-русски. Если нужно

изменить интерфейс на английский язык, установите переменную `LANGUAGE` создайте файл `Renviron.site`, в который внесите строчку

```
LANGUAGE=en
```

Этот файл должен находиться в так называемой "домашней" директории R. Более подробно про это можно узнать, если прочитать помощь по команде `Startup()`. Под Linux можно просто внести эту строчку в строку вызова как опцию.

Выход из R

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

```
>q()
```

```
Save workspace image? [y/n/c]: n
```

Уже такой простой пример демонстрирует, что в R любая команда имеет аргумент в круглых скобках. Если аргумент не указан, скобки все равно нужны. Если вы забудете это сделать, то вместо выхода из R получите определение функции:

```
> q
```

```
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
```

```
<environment: namespace:base>
```

Как узнать, как правильно вызывать функцию? Для этого надо научиться получать справку. Есть два пути. Первый вызвать команду справки:

```
>help(q)
```

или

```
>?q
```

Вы увидите отдельное окно либо (если вы работаете в Linux) текст помощи в основном окне программы (выход из этого режима по клавише "q"). Если внимательно прочитать текст, становится ясно, что выйти из R можно, и не отвечая на вопрос, если ввести `q("no")`.

Зачем же нужен этот вопрос? Что будет, если ответить положительно? В этом случае в рабочую папку R (ту, из которой он вызван) запишутся два файла: бинарный `.RData` и текстовый `.Rhistory`. Первый содержит все объекты,

созданные вами за время сессии. Второй полную историю введенных команд.

Когда вы работаете в R, предыдущую команду легко вызвать, нажав клавишу-стрелку вверх. Но если вы сохраните файл `.Rhistory`, Ваши команды будут доступны и в следующей сессии - при условии, что Вы вызовете R из той же самой папки. И наоборот, если Вы случайно сохраните рабочую среду (эти два файла), то при следующем старте они загрузятся автоматически. Иногда такое поведение R становится причиной различных недоумений, так что будьте внимательны.

1.1.2. Основные команды

Для выполнения тех или иных операций R использует функции. Для запуска функции с именем `funcname` мы набираем `funcname (input1, input2)`, где входные параметры, или аргументы, `input 1` и `input2` сообщают R, как именно следует исполнить эту функцию. Например, для создания вектора с несколькими числами мы используем функцию конкатенации `c()`. Следующая команда говорит R объединить числа 1, 3, 2 и 5 и сохранить их в виде вектора с именем `x`. Когда мы наберем `x`, то в ответ получим этот вектор.

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

Обратите внимание на то, что `>` не является частью команды — R выводит этот знак просто, чтобы показать свою готовность к выполнению следующей команды. Мы можем сохранять объекты не только с помощью `<-`, но также и `=`:

```
> x = c(1, 6, 2)
> x
[1] 1 6 2
> y = c(1, 4, 3)
```

Многократное нажатие клавиши со стрелкой вверх приведет к показу предыдущих команд, которые можно отредактировать. Это полезно, поскольку необходимость повторения похожих команд возникает часто.

Кроме того, ввод команды `?funcname` всегда откроет новое окно со справочным файлом, содержащим дополнительную информацию по функции `funcname`.

Мы можем попросить R выполнить сложение двух чисел. В этом случае программа сначала добавит первое число из x к первому числу из y и т. д. Однако x и y должны быть одинаковой длины. Мы можем проверить их длину при помощи функции `length()`.

```
> length(x)
```

```
[1] 3
```

```
> length(y)
```

```
[1] 3
```

```
> x + y
```

```
[1] 2 10 5
```

Функция `ls()` позволяет просмотреть список всех объектов, таких как данные и функции, которые мы сохранили до этого момента. Функцию `rm()` можно использовать для удаления любого нежелательного объекта.

```
> ls()
```

```
[1] "x" "y"
```

```
> rm(x, y)
```

```
> ls()
```

```
Character (0)
```

Имеется также возможность удалить все объекты за один раз:

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

Функцию `matrix()` можно использовать для создания матрицы с числами. Перед применением функции `matrix()` мы можем узнать о ней больше:

```
> ?matrix
```

Справочный файл сообщает, что функция `matrix()` принимает несколько входных параметров, но пока мы сосредоточимся на первых трех: данные (элементы матрицы), количество строк и количество столбцов. Сначала создадим простую матрицу.

```
> x = matrix (data = c(1, 2, 3, 4), nrow = 2, ncol = 2)
```

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

Обратите внимание, что мы могли бы с тем же успехом не набирать `data =`, `nrow =` и `ncol =` в приведенной выше команде `matrix()`, а просто ввести

```
x = matrix(c(1, 2, 3, 4), 2, 2)
```

и это имело бы тот же эффект. Однако иногда бывает полезно указать имена аргументов, т. к. иначе R будет предполагать, что аргументы функции подаются на нее в том же порядке, который приведен в справочном файле этой функции. Как показано в данном примере, по умолчанию R создает матрицы путем последовательного заполнения столбцов. В качестве альтернативы можно использовать опцию `byrow = TRUE` для заполнения матрицы по строкам.

```
> matrix(c(1, 2, 3, 4), 2, 2, byrow = TRUE)
[,1] [,2]
[1,] 1  2
[2,] 3  4
```

Заметьте, что в приведенной выше команде мы не присвоили матрице никакого имени вроде `x`. В этом случае матрица выводится на экран, но не сохраняется для будущих вычислений. Функция `sqrt()` возвращает квадратный корень каждого элемента вектора или матрицы. Команда `x^2` возводит каждый элемент `x` в квадрат; возможно использование любой степени, включая дроби и отрицательные степени.

```
> sqrt(x)
      [,1] [,2]
[1,] 1.00 1.73
[2,] 1.41 2.00
> x^2
      [,1] [,2]
[1,]    1    9
[2,]    4   16
```


Функция `rnorm()` генерирует вектор случайных нормально распределенных значений, при этом аргумент n задает размер выборки. Каждый раз при вызове этой функций мы будем получать другой результат. Здесь мы создаем два коррелирующих набора чисел x и y и применяем `cor()` функцию `cor()` для расчета корреляции между ними.

```
> x = rnorm(50)
> y = x + rnorm(50, mean = 50, sd = .1)
> cor(x, y)
[1] 0.995
```

По умолчанию `rnorm()` создает случайные переменные, представляющие стандартное нормальное распределение со средним значением 0 и стандартным отклонением 1. Однако, как показано выше, среднее значение и стандартное отклонение можно изменить при помощи аргументов `mean` и `sd`. Иногда мы хотим, чтобы наш код в точности воспроизводил один тот же набор случайных чисел; для этого мы можем использовать `set.seed()` функцию `set.seed()`. Функция `set.seed()` принимает в качестве аргумента произвольное целое число.

```
> set.seed(1303)
> rnorm(50)
[1] -1.1440 1.3421 2.1854 0.5364 0.0632 0.5022 -0.0004
```

Мы используем `set.seed()` во всех лабораторных работах каждый раз, когда выполняем вычисления, содержащие случайные величины. Как правило, это должно помочь пользователю воспроизвести наши результаты. Однако следует отметить, что с появлением новых версий R могут возникнуть небольшие несоответствия между книгой и тем, что выдает R.

Функции `mean()` и `var()` можно использовать для вычисления среднего значения и дисперсии некоторого набора чисел. Применение `sqrt()` к результату работы `var()` даст стандартное отклонение, можно просто применить функцию `sd()`.

```
> set.seed(0)
> y = rnorm(100)
> mean(y)
[1] 0.0110
```

```
> var(y)
[1] 0.7329
> sqrt(var(y))
[1] 0.8561
> sd(y)
[1] 0.8561
```

1.2. Написание скриптов на языке R

Задание

Написать скрипт из команд разделов 1.1.1 и 1.1.2, сохранить и запустить скрипт. Подключить пакет и запустить функцию из пакета.

Просто открыть сессию R и вводить в окно программы команды, одну за другой — это лишь один из возможных способов работы. Гораздо более продуктивный метод, который является заодно и серьёзнейшим преимуществом R — это создание скриптов. Иначе говоря, программ, которые потом загружаются в R и интерпретируются им. С самого начала работы следует создавать скрипты, даже для таких задач, которые кажутся пустяковыми — это в будущем значительно сэкономит Ваше время.

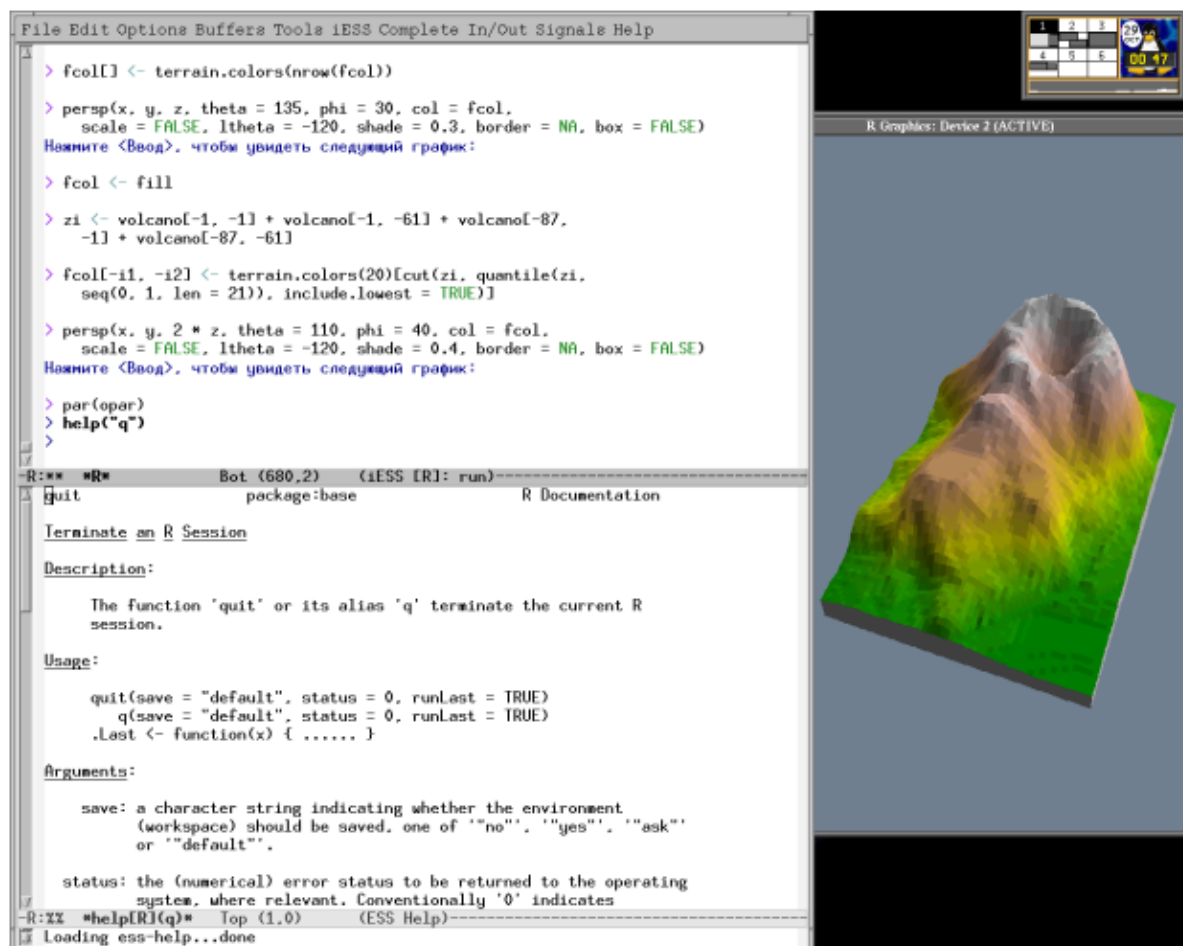
Создание скриптов по любому поводу и даже без особого повода — одна из основ культуры работы в R. Создать скрипт очень просто. Допустим, после открытия сессии была введена необходимая для получения искомого результата последовательность команд. Чтобы сделать свою работу воспроизводимой — надо просто сохранить историю команд. Лучше всего это сделать с помощью команды:

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

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

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

Опция `echo` добавлена для того чтобы можно было видеть сами команды, а не только результат их выполнения. Есть ещё более эффективный способ работы: Вы открываете Ваш скрипт в текстовом редакторе, а потом посылаете отдельные его строки прямо в R. Есть несколько редакторов, которые умеют так делать. Во-первых, это Emacs с установленным пакетом ESS («Emacs Speaks Statistics»).



Преимущество этой системы в том, что R запускается прямо в одном из окон редактора. Во-вторых, штатные R GUI под Windows и под Mac также имеют встроенные редакторы скриптов. К сожалению, Windows-редактор не подсвечивает синтаксис, и вообще довольно неудобен. Вместо него тем пользователям, которых пугает перспектива освоения Emacs, можно порекомендовать отличный редактор Tinn-R, специально «заточенный» под такую работу.

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

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

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

Пакеты. Ещё одно важное преимущество R — наличие для него многочисленных расширений или пакетов буквально на все случаи жизни. При установке R на компьютер, несколько пакетов уже в наличии: так

называемые базовые пакеты, без которых система просто не работает (пакет базовых функций `base` или пакет `grDevices`, который управляет выводом графиков), и «рекомендованные» пакеты (пакет для специализированного кластерного анализа `cluster`, пакет для анализа нелинейных моделей `nlme` и пр.). Кроме того можно поставить любой из почти полутора тысяч (!) доступных на CRAN пакетов.

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

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

Естественно, это всё следует делать, если R устанавливается не из стандартного репозитория дистрибутива GNU/Linux. В случае стандартной установки можно поискать пакеты по сочетанию `cran`. В Debian GNU/Linux Etch таких пакетов ровно 85 — это не 1500 пакетов с CRAN, но скорее всего там уже есть многое из того, что необходимо. Как только пакет установлен, то он сразу готов к работе. Нужно только инициализировать его перед употреблением. Для этого служит команда `library()`.

2.1. Использование базовых математических функций и операций

Задание

Запустить все команды из разделов 2.1.1 и 2.1.2. Разобраться в особенностях применения базовых математических операций и функций.

2.1.1. Логические операции

Логические операции, предусмотренные в R, приведены ниже:

Запись в R	Описание
$! x$	логическое отрицание
$x \& y$	поэлементное логическое И
$x \&\& y$	логическое И только для первых элементов векторов x и y
$x y$	поэлементное логическое ИЛИ
$x y$	логическое ИЛИ только для первых элементов x и y
$x \neq y$	если x и y различны, то получим TRUE, иначе FALSE

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

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

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

```
> 'a' < 'b'
```

```
[1] TRUE
```

```
> 'иэ' >= 'ия'
```

```
[1] FALSE
```

```
> 'иэ' != 'ия'
```

```
[1] TRUE
```

```
> 'и' > 'э'
```

```
[1] FALSE
```

```
> 'и' 'вг'>='вв'
```

```
[1] TRUE
```

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

```
> x=10000; y='а'; x>y
```

```
[1] FALSE
```

```
> x=10000;y='а';y>x
```

```
[1] TRUE
```

Также можно сравнивать и логические переменные:

```
> x=F;y=T
```

```
> x<y
```

```
[1] TRUE
```

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

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

```
> x=10;y=NA
```

```
> x>y
```

```
[1] NA
```

```
> x=NA;y=NaN > x>=y
```

```
[1] NA
```

2.1.2. Математические функции

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

```
> x=0.235;x
```

```
[1] 0.235
```

```
> x=.235;x
```

```
[1] 0.235
```

```
> x=23.5e-2;x
```

```
[1] 0.235
```

```
> x=235E-3; x
```

```
[1] 0.235
```

```
> x=0.00235e2; x
```

```
[1] 0.235
```

представлены различные способы записи одного и того же числа.

В R над числами можно выполнять обычные арифметические операции:

+ (сложение),

– (вычитание),

* (умножение),

/ (деление),

^ (возведение в степень),

%% (целочисленное деление),

%% (остаток от деления).

Операции имеют обычный приоритет, т.е. сначала выполняется возведение в степень, затем умножение или деление, потом уже сложение или вычитание. В выражениях также используются круглые скобки и операции в них имеют приоритет. Набранное в командной строке арифметическое выражение после нажатия на Enter вычисляется и результат сразу же отображается:

```
> 2*(7-9)^8+6/3
```

```
[1] 514
```

Ответ — 514.

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

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

Логарифмические и экспоненциальные функции

Экспоненты:

- `exp(x)` — вычисление e^x ;
- `expm(x)` — для $|x| \leq 1$ нахождение $e^x - 1$.

В R реализовано несколько логарифмических функций:

- $\log(x)$ и $\log_b(x)$ — натуральный логарифм числа x ;
- $\log_{10}(x)$ — логарифм по десятичному основанию;
- $\log_2(x)$ — логарифм по основанию 2;
- $\log_{1p}(x)$ — вычисление логарифма $\ln(1+x)$ для $|x| \leq 1$ (при $x \approx -1$ точность вычислений снижается);
- $\log(x, \text{base=})$ и $\log_b(x, \text{base=})$ — вычисление логарифмов по произвольному основанию.

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

Функции округления

Доступны следующие функции округления.

- $\text{ceiling}(x)$ — находит минимальное целое, не меньшее x (округление в сторону +1);
- $\text{floor}(x)$ — возвращает максимальное целое, не превосходящее x

(округление в сторону -1);

- `trunc(x)` — отбрасывает дробную часть (округление в сторону 0);
- `round(x)` — округляет к ближайшему целому;
- `round(x, dig)` — общий вид функции `round(x)`, дополнительный параметр `dig` указывает на число знаков после запятой, до которого нужно произвести округление;
- `signif(x, digits = 6)` — округляет до заданного (`digits`) числа значащих знаков (т.е. чисел, отличных от нуля), по умолчанию параметр `digits` равен 6.

```
> x=1.5555
```

```
> ceiling(x)
```

```
[1] 2
```

```
> floor(x)
```

```
[1] 1
```

```
> trunc(x)
```

```
[1] 1
```

```
> y=-1.5
```

```
> ceiling(y)
```

```
[1] -1
```

```
> floor(y)
```

```
[1] -2
```

```
> trunc(y)
```

```
[1] -1
```

```
> x=1/33;x
```

```
[1] 0.03030303
```

```
> round(x)
```

```
[1] 0
```

```
> round(x, 4)
```

```
[1] 0.0303
```

```
> signif(x, 5)
```

```
[1] 0.030303
```

Модуль и квадратный корень

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

```
> x=-5; y=F; > abs(x)
```

```
[1] 5
```

```
> abs(y)
```

```
[1] 0
```

```
> z=T; abs(z)
```

```
[1] 1
```

```
> z=5-1i*2 > abs(z)
```

```
[1] 5.385165
```

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

```
> x=4; sqrt(x)
```

```
[1] 2
```

```
> y=-4; sqrt(y)
```

```
[1] NaN
```

Предупреждение

In `sqrt(y)` : созданы NaN

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

```
> y=as.complex(y); sqrt(y)
```

```
[1] 0+2i
```

Специальные функции

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

- `gamma(x)` — гамма-функция $\Gamma(x)$, где x — действительное число, не равное нулю и не являющееся целым отрицательным.
- `lgamma(x)` — натуральный логарифм абсолютного значения гамма-функции.
- `digamma(x)` — первая производная натурального логарифма

гаммафункции.

- `trigamma(x)` — вторая производная натурального логарифма гаммафункции.

- `psigamma(x, deriv)` — вычисляет *deriv*-производную от ψ -функции (первой производной гамма-функции) (по умолчанию *deriv* = 0, т.е. $psigamma(x) = digamma(x)$).

```
> x=-8.9
```

```
> gamma(x)
```

```
[1] -3.507258e-05
```

```
> x=2.15
```

```
> gamma(x)
```

```
[1] 1.072997
```

```
> digamma(x)
```

```
[1] 0.5152385
```

```
> psigamma(x)
```

```
[1] 0.5152385
```

```
> trigamma(x)
```

```
[1] 0.5894157
```

- `beta(a, b)` — вычисляется значение бета-функции $B(a, b)$ с параметрами a и b , где параметры a и b больше нуля.

- `lbeta(a, b)` — натуральный логарифм бета-функции $B(a, b)$.

```
> a=2;b=3
```

```
> beta(a,b)
```

```
[1] 0.08333333
```

```
> lbeta(a,b)
```

```
[1] -2.484907
```

- `choose(n, k)` — число сочетаний C_n^k из n по k , где n — действительное число, k — целое положительное.

- `lchoose(n, k)` — натуральный логарифм числа сочетаний C_n^k .

- `factorial(x)` — факториал x , где $x \geq 0$.

- `lfactorial(x)` — натуральный логарифм факториала x .

```
> n=5.4;k=3;x=4.6;x1=4
```

```
> choose(n,k)
```

```
[1] 13.464
```

```
> choose(round(n),k)
```

```
[1] 10
```

```
> factorial(x)
```

```
[1] 61.55392
```

```
> factorial(x1)
```

```
[1] 24
```

Тригонометрические функции

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

Обратные тригонометрические функции:

- $\sin(x)$ — $\arcsin(x)$;
- $\cos(x)$ — $\arccos(x)$;
- $\tan(x)$ — $\arctg(x)$.

```
> sin(pi/2)
```

```
[1] 1
```

где π — это константа π .

Если у функции несколько аргументов, то они отделяются знаком , (запятая).

Например, $\text{atan2}(y, x)$ возвращает угол между осью Ox и вектором $(x; y)$.

Гиперболические функции:

- $\cosh(x)$ — гиперболический косинус аргумента x ;
- $\text{acosh}(x)$ — обратный гиперболический косинус x ;
- $\sinh(x)$ — гиперболический синус x ;
- $\text{asinh}(x)$ — обратный гиперболический синус x ;
- $\tanh(x)$ — гиперболический тангенс x ;
- $\text{atanh}(x)$ — обратный гиперболический тангенс x .

Операции над комплексными переменными

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

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

Специальные функции для комплексных переменных:

- $\text{Re}(z)$ — нахождение действительной части комплексного числа z ;
- $\text{Im}(z)$ — мнимая часть комплексного числа z ;
- $\text{Mod}(z)$ — модуль $\rho = \sqrt{x^2 + y^2}$ комплексного числа $z = x + iy$;
- $\text{Arg}(z)$ — аргумент ϕ комплексного числа $z = x + iy$, где $x = \rho \cos \phi$, $y = \rho \sin \phi$;
- $\text{Conj}(z)$ — комплексно сопряжённое к z число. К комплексным переменным можно применять и тригонометрические, и кумулятивные функции.

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

```
[1] 0.7853982
```

```
> Conj(x)
```

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

```
> cos(x)
```

```
[1] -1.565626-3.297895i
```

2.2. Написание программ с циклами и условными операторами

Задание

Запустить все команды из разделов 2.2.1-2.1.6. Разобраться в особенностях применения условных операторов и циклов.

2.2.1. Оператор If

Краткий вариант оператора `if`

`If (условие) выражение`

условие — любой оператор условия (`<`, `>`, `>=`, `<=`, `==`, `!=`), результатом выполнения которого является логический вектор единичный длины.

Если значение вектора `TRUE`, то выполняется выражение. Выражение — одно или несколько выражений, выполняемых в случае верности условия. Если задано несколько выражений, то они должны быть заключены в фигурные скобки `{}` и разделяться точкой с запятой (если на одной строке). Оператор возвращает значение выражения в случае верности условия или ничего не возвращает (`NULL`).

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

Условие выполнено и в результате получено значение `z`.

```
> x=5; y=4
> if (x<y) w=x+y
> w
```

Ошибка: объект `'w'` не найден

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

Пусть `x` и `y` являются векторами одинаковой длины (10). Зададим условие: если `x` не равен `y`, то берётся отношение `x/y`. В результате должны получить вектор, чья размерность совпадает с размерностью исходных векторов. Но из каких элементов он состоит?

```
> x=1:10; y=10:1
```



```

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

```

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

Предупреждение

```

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

```

Таким образом, длина условия должна быть равной единице.

Полный вариант оператора `if`

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

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

```

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

```

2.2.2. Оператор *Ifelse*

Функция

```
ifelse(условие, yes, no)
```

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

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

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

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

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

Предупреждение

In sqrt(x) : созданы NaN

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

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

Ещё один пример:

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

[2,]	1.000000e+00	1.000000e+00	1.000000e+00
[3,]	-1.000000e+00	-1.000000e+00	-1.000000e+00

2.2.3. Оператор *for*

`for(переменная in последовательность) выражение`

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

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

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

```
> x=1:10; y=10;1
> for(i in 1:10)
+ { if (x[i]<y[i]) {w=x[i]/y[i]}
+ else {w=x[i]*y[i]}
+ }
> w                                     # w – не вектор, а число
[1] 10
```

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

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

```
c=integer(0)
for (i in c) {m=i}
```

m

Ошибка: Объект 'm' не найден

2.2.4. Оператор while

Ещё один оператор цикла.

while (условие) выражение

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

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

2.2.5. Операторы repeat, break и next

Оператор repeat

repeat выражение

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

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

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

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

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

2.2.6. Оператор `switch`

Оператор

`switch`(управляющее выражение, альтернативные действия) позволяет выполнять одну из нескольких операций в зависимости от результатов управляющего выражения.

Управляющее выражение возвращает:

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

Если возвращаемое контрольным выражением значение не соответствует ни номеру выполняемой операции, ни имени, то результатом оператора `switch` будет `NULL`. Оператор `switch` не является самостоятельным, используется либо внутри функций, либо внутри других управляющих конструкций.

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

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

3.1. Работа с векторами, массивами и матрицами

Задание

Запустить все команды из разделов 3.1.1-3.1.4. Разобраться в способах ввода, редактирования и выполнения операций функций для этих типов объектов данных.

3.1.1. Векторы

Способы задания векторов

В R можно выделить несколько способов задания векторов :

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

Первый способ реализуется при помощи функции

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

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

```
> vector('numeric',10)
```

```
[1] 0 0 0 0 0 0 0 0 0 0
```

```
> vector('complex',10)
```

```
[1] 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i
```

```
> vector('logical',10)
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
> vector('character',10)
```

```
[1] "" "" "" "" "" "" "" "" "" ""
```

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

Если задать только длину вектора

```
vector(10)
```

то будет выдано сообщение об ошибке.

Альтернативой функции `vector()` являются `numeric()`, `logical()`, `complex()`, `character()`.

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

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

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

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

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

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


Наконец, можно задать вектор, набирая данные на клавиатуре. Это реализуется при помощи `scan()`.

.

Создадим числовой вектор `y` при помощи функции `scan()`.

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

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

```
1:
```

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

```
1: 2 34
```

нажатие на `Enter` означает переход на новую строку

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

```
7:
```

Двойное нажатие на клавишу `Enter` завершает ввод. Выводится сообщение о количестве считанных элементов вектора. Чтобы вывести созданный таким образом вектор, достаточно набрать его имя:

```
> y
```

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

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

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

Создадим два объекта — вектор `y`, состоящий из элементов разного типа, и фактор `z`.

```
> y=c(T, F, 1, -3, 2, 1+1i*2);y
```

```
[1] 1+0i 0+0i 1+0i -3+0i 2+0i 1+2i
```

```
> z=factor(y)
```

```
> z=factor(y);z
```

```
[1] 1+0i 0+0i 1+0i -3+0i 2+0i 1+2i
```

```
Levels: -3+0i 0+0i 1+0i 1+2i 2+0i
```

Проверим их на принадлежность к классу векторов.

```
> is.vector(y)
```

```
[1] TRUE
```

```
> is.vector(z)
```

```
[1] FALSE
```

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

Продолжим прошлый пример и переведем созданный фактор в вектор.

```
> is.vector(z)
```

```
[1] FALSE
```

```
> w=as.vector(z)
```

Проверим теперь, является `w` вектором

```
> is.vector(w)
```

```
[1] TRUE
```

и выведем его на экран

```
>w
```

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

Аргументами функции `c()` сами могут являться векторами. В этом случае

как результат получаем конкатенацию (объединение) этих векторов.

Скалярные значения (т. е. числа) воспринимаются R как векторы длины 1.

Таким образом, аргументами функции `c()` могут быть как векторы, так и скаляры.

Например,

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

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

Вектор, состоящий из последовательных чисел, можно получить с помощью команды

начальное-значение : конечное-значение

Например,

```
1:5
```

На эту команду похожа функция `seq()`, которая генерирует арифметическую прогрессию. Нужно задать начальное значение, конечное значение и либо шаг прогрессии:

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

либо количество элементов последовательности:

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

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

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

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

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

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

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

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

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

Над векторами можно выполнять арифметические операции и элементарные функции. Бинарная операция над двумя векторами одинаковой длины производится над каждой парой элементов, и результатом является вектор той же длины, что и исходные. В случае, когда размерность векторов не совпадает, производится приведение длины более короткого вектора к длине длинного.

Приведение выполняется следующим образом — элементы вектора дублируются необходимое число раз полностью (если длина большего вектора кратна длине меньшего вектора)

```

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

```

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

```

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

```

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

```

> x1=c(10,-3,6,2,-4,NA,1:5);x1
[1] 10 -3 6 2 -4 NA 1 2 3 4 5
> x1*2
[1] 20 -6 12 4 -8 NA 2 4 6 8 10
> x1/4
[1] 2.50 -0.75 1.50 0.50 -1.00 NA 0.25 0.50 0.75 1.00 1.25
> x1+5
[1] 15 2 11 7 1 NA 6 7 8 9 10
> x1-2

```

```
[1] 8 -5 4 0 -6 NA -1 0 1 2 3
```

Элементарные математические функции применяются к каждой компоненте вектора.

```
> x = c(0, pi/2, pi)
```

```
> sin(x)
```

```
[1] 0.000000e+00 1.000000e+00 1.224606e-16
```

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

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

```
> sqrt(x1)
```

```
[1] 3.162278 NaN 2.449490 1.414214 NaN NA 1.000000 1.414214
```

```
[9] 1.732051 2.000000 2.236068
```

Предупреждение

```
In sqrt(x1) : созданы NaN
```

Функции для работы с векторами

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

- `length()` — длина вектора.

```
> x=c(1:5,NA,NaN,6:10);x
```

```
[1] 1 2 3 4 5 NA NaN 6 7 8 9 10
```

```
> length(x)
```

```
[1] 12
```

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

```
> x=c(1:5,NA,NaN,6:10);x
```

```
[1] 1 2 3 4 5 NA NaN 6 7 8 9 10
```

```
> min(x)
```

```
[1] NaN
```

```
> max(x)
```

```
[1] NaN
```

```
> min(x, na.rm=T)
```

```
[1] 1
```

```
> max(x, na.rm=T)
```

```
[1] 10
```

• `pmax()` и `pmin()` — параллельный максимум (минимум) для любого заданного числа векторов. Результат — вектор, длина которого равна длине максимального из сравниваемых векторов и элементы которого есть максимальные (минимальные) значения сравниваемых векторов, находящиеся на одинаковых позициях. Т.е. `pmax(x, y) = (max(x1, y1), max(x2, y), . . .)`, `pmin(x, y) = (min(x1, y1), min(x2, y), . . .)`. Если векторы-аргументы имеют разную длину, то наименьший вектор приводится при помощи повторения элементов к длине наибольшего вектора.

```
> x=rpois(10,1);x # Генерируем случайную выборку
```

```
[1] 2 0 1 3 1 1 1 1 0 2
```

```
> y=rpois(6,1);y # с распределением Пуассона
```

```
[1] 0 4 2 1 0 0
```

```
> pmax(x,y)
```

```
[1] 2 4 2 3 1 1 1 4 2 2
```

```
> pmin(x,y)
```

```
[1] 0 0 1 1 0 0 0 1 0 1
```

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

```
> x=c(NA, NaN, NaN)
```

```
> y=c(NaN, NA, NaN)
```

```
> pmin(x,y, na.rm=T)
```

```
[1] NaN NA NaN
```

```
> pmax(x,y, na.rm=T)
```

```
[1] NaN NA NaN
```

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

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

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

```
> mean(x1)
```

```
[1] NA
```

```
> mean(x1,na.rm=T)
```

```
[1] 2.6
```

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

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

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

```
> range(x1)
```

```
[1] NA NA
```

```
> range(x1,na.rm=T)
```

```
[1] -4 10
```

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

```
> x=c(1:5,NA,6:10)
```

```
> sum(x)
```

```
[1] NA
```

```
> sum(x,na.rm=T)
```

```
[1] 55
```

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

```
> x=c(1:5,NA,NaN,6:10);x
```

```

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

```

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

```

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

```

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

```

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

```

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

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

```

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

```



```

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

```

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

```

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

```

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

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

```

> last=rank(x,na.last=T)
> first=rank(x,na.last=F)
> Na=rank(x,na.last=NA)

```

```

> keep=rank(x,na.last='keep')
> rbind(x,last,first,keep)
[,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
x 8 11.0 13 NA 7.0 11.0 10 NA 14 7.0
last 3 5.5 7 9 1.5 5.5 4 10 8 1.5
first 5 7.5 9 1 3.5 7.5 6 2 10 3.5
keep 3 5.5 7 NA 1.5 5.5 4 NA 8 1.5
> Na
[1] 3.0 5.5 7.0 1.5 5.5 4.0 8.0 1.5

```

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

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

```

> x
[1] 8 11 13 9 7 11 10 12 14 7
> y
[1] 9 10 9 10 9
> match(x,y)
[1] NA NA NA 1 NA NA 2 NA NA NA
> match(y,x)
[1] 4 7 4 7 4

```

Кумулятивные (накопительные) функции

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

- `cummin(x)` — кумулятивный минимум (последовательный минимум по элементам вектора).

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

Задание символьных векторов

Символьные переменные в R задаются при помощи двойных или одинарных кавычек, либо же при помощи функции

`character()`.

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

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

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

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

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

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

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

Присвоение значений таким векторам происходит при помощи индексов и управляющих конструкций.

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

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

Функция `paste()`

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

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

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

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

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

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

Можно заменить разделитель на другой:

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

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

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

Выделим различные подстроки из строки `сытое брюхо к учению глухо`.

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

Получим следующий символьный вектор `q`

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

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

```
phrase='сытое брюхо к учению глухо'  
strsplit(phrase,split=character(0))
```

```
[[1]]
[1] "с" "ы" "т" "о" "е" " " "б" "р" "ю" "х" "о" " " "к" " " "у" "ч" "е" "н" "и"
[20] "ю" " " "т" "л" "у" "х" "о"
```

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

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

Здесь разбиение производилось относительно пробела.

Ещё одна полезная функция, позволяющая определить количество символов в строке или символьном векторе. Это функция `nchar()`.

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

Логические векторы

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

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

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

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

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

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

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

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

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

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

Задание имён элементам векторов

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

Предположим, имеются следующие данные — количество студентов в различных группах (НК-201, НП-201, НП-202, НИ-201, НП-203).

Эти числа приведены в векторе `x`.

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

Однако, только тот, кто составлял этот вектор знает, какой элемент соответствует какой группе. Присвоим имена каждому элементу вектора, тогда смысл его станет понятен любому.

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

и посмотрим на результат

```
> group
НП-201 НП-202 НП-203 НК-201 НИ-201
```

```
17 19 25 13 7
```

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

```
> group[2]
```

```
НП-202
```

```
19
```

```
> group['НП-203']
```

```
НП-203
```

```
25
```

```
> group['НП-203']=18
```

```
> group
```

```
НП-201    НП-202    НП-203    НК-201    НИ-201
```

```
17        19        18        13        7
```

Индексация векторов

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

```
> u <- 1
```

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

```
> u
```

```
[1] 1 NA NA NA 5
```

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

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

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

```
> length(x1)
```



```
[1] 11
> x1[-6]
[1] 10 -3 6 2 -4 1 2 3 4 5
```

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

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

Векторы в качестве индексов

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

Что будет получено в результате применения в качестве индекса вектора?

Возможны следующие случаи:

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

```
> x1=c(10,-3,6,2,-4,NA,1:5);x1
[1] 10 -3 6 2 -4 NA 1 2 3 4 5
> x2=c(T,F,T,F,F)
> y=x1[x2];y
[1] 10 6 NA 2 5
> x3=rep(c(T,F),10);x3
[1] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
[13] TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE
> y=x1[x3];y
[1] 10 6 -4 1 3 5 NA NA NA NA
```

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

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

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

```
> x=c(1,3,5,15)
```

```
> y=x1[x];y
```

```
[1] 10 6 -4 NA
```

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

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

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

```
> x=c(-1,-3,-5)
```

```
> y=x1[x];y
```

```
[1] -3 2 NA 1 2 3 4 5
```

```
> y=x1[-(2:6)];y
```

```
[1] 10 1 2 3 4 5
```

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

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

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

```
> names(x1)=letters[1:length(x1)];x1
```

```
a b c d e f g h i j k
```

```
10 -3 6 2 -4 NA 1 2 3 4 5
```

```
>
```

```
> y=x1[c('a','c','f')];y
```

```
a c f
```

```
10 6 NA
```

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

Также в качестве индексов можно указывать различные операции, приводящие к созданию логического вектора.

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

или

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

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

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

Команда

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

заменяет значения NA и NaN нулями.

Выражение

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

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

```
> z=(x1 + 1)[(!is.na(x1)) & x1 > 0];z
```

```
[1] 11 7 3 2 3 4 5 6
```

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

Функция `which()`

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

Пусть задан вектор `x`.

```
> x
```

```
[1] 10 4 7 9 7 8 6 14 10 10 5 8 11 11 20
```

```
> length(x)
```

```
[1] 15
```

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

```
> which(x>10)
```

```
[1] 8 13 14 15
```

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

```
> x[13]
```

```
[1] 11
```

либо

```
> y=which(x>10)
```

```
> x[y[2]]
```

```
[1] 11
```

Найдём в векторе `x` второй элемент, кратный 5.

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

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

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

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

Найдём все элементы вектора, наиболее близкие к 12.

```
> x
[1] 10 4 7 9 7 8 6 14 10 10 5 8 11 11 20
> y=which(abs(x-12)==min(abs(x-12)));y
[1] 13 14
```

и выведем их на экран.

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

3.1.2. Матрицы

Задание матрицы

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

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

Для задания матрицы

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

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

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

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

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

но это не обязательно:

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

Предупреждение

```
In matrix(1:12, nrow = 5, ncol = 3) :
```

```
длина данных [12] не является множителем количества строк [5]
```

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

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

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

```
> A=matrix(1:12, ncol = 5);A
```

Предупреждение

```
In matrix(1:12, ncol = 5) :
```

```
длина данных [12] не является множителем количества столбцов [5]
```

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

Предупреждение

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

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

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

Функция `cbind(A, B)`

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

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

```
> A=matrix(1:12,nrow=3);A
[,1] [,2] [,3] [,4]
[1,] 1 4 7 10
[2,] 2 5 8 11
[3,] 3 6 9 12
> B=matrix(13:24,nrow=3);B
[,1] [,2] [,3] [,4]
```



```

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

```

Функция rbind(A, B)

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

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

```

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

```

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

```

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

```

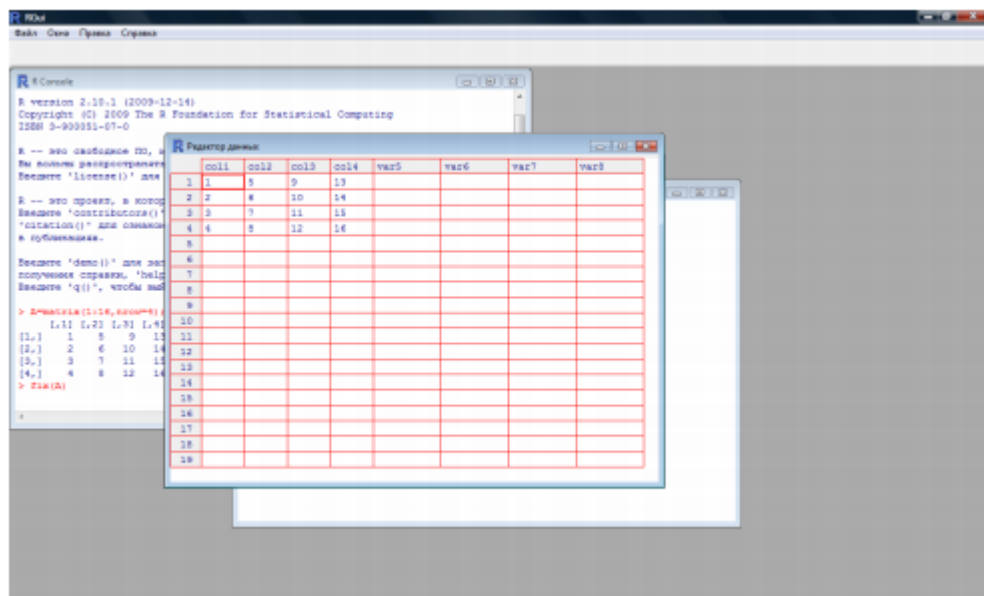
Для построения квадратной единичной матрицы нужно задать только число строк `nrow` в матрице (если задать число столбцов `ncol`, то будет выведено сообщение об ошибке).

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

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

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

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



Редактор матриц

Создадим матрицу A

```
>A=matrix(1:16,nrow=4);A
```

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

```
[1,] 1 5 9 13
```

```
[2,] 2 6 10 14
```

```
[3,] 3 7 11 15
```

```
[4,] 4 8 12 16
```

и изменим её в редакторе.

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

Операции над матрицами

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

```
A = matrix(1:9, nrow = 3);A
```

```
[,1] [,2] [,3]
```

```
[1,] 1 4 7
```

```
[2,] 2 5 8
```

```
[3,] 3 6 9
```

```
B = matrix(-(1:9), ncol = 3,byrow=T);B
```

```
[,1] [,2] [,3]
[1,] -1 -2 -3
[2,] -4 -5 -6
[3,] -7 -8 -9
```

A + B

```
[,1] [,2] [,3]
[1,] 0 2 4
[2,] -2 0 2
[3,] -4 -2 0
```

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

> A+3

```
[,1] [,2] [,3]
[1,] 4 7 10
[2,] 5 8 11
[3,] 6 9 12
```

> B+3

```
[,1] [,2] [,3]
[1,] 2 1 0
[2,] -1 -2 -3
[3,] -4 -5 -6
```

> (1:3)*A

```
[,1] [,2] [,3]
[1,] 1 4 7
[2,] 4 10 16
[3,] 9 18 27
```

> A*(1:3)

```
[,1] [,2] [,3]
```

```

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

```

Если длины объектов не кратны, то выводится предупреждение

```

> (0:3)*A
[,1] [,2] [,3]
[1,] 0 12 14
[2,] 2 0 24
[3,] 6 6 0

```

Предупреждение

```
In (0:3) * A :
```

длина большего объекта не является произведением длины меньшего объекта

Элементарные математические функции также применяются поэлементно:

```

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

```

```
[3,] 1.945910 2.0794415 2.197225
```

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

```
> x =1:5; x
```

```
[1] 1 2 3 4 5
```

```
> y =-2:3;y
```

```
[1] -2 -1 0 1 2 3
```

```
> outer(x, y, "*")
```

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

```
[1,] -2 -1 0 1 2 3
```

```
[2,] -4 -2 0 2 4 6
```

```
[3,] -6 -3 0 3 6 9
```

```
[4,] -8 -4 0 4 8 12
```

```
[5,] -10 -5 0 5 10 15
```

```
> outer(x, y, "^")
```

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

```
[1,] 1.0000000 1.0000000 1 1 1 1
```

```
[2,] 0.2500000 0.5000000 1 2 4 8
```

```
[3,] 0.1111111 0.3333333 1 3 9 27
```

```
[4,] 0.0625000 0.2500000 1 4 16 64
```

```
[5,] 0.0400000 0.2000000 1 5 25 125
```

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

```
> x%o%y
```

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

```
[1,] -2 -1 0 1 2 3
```

```
[2,] -4 -2 0 2 4 6
```

```
[3,] -6 -3 0 3 6 9
```

```
[4,] -8 -4 0 4 8 12
[5,] -10 -5 0 5 10 15
```

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

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

Для решения системы линейных уравнений $Ax = b$ с квадратной невырожденной матрицей A есть функция `solve(A, b)`:

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

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

```
> A = matrix(c(3,0,4,4), nrow = 2);A
```

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

```
[1,] 3 4
```

```
[2,] 0 4
```

```
> b=c(1,1)
```

```
> backsolve(A,b)
```

```
[1] 0.00 0.25
```

```
> B= matrix(c(3,4,0,4), nrow = 2);B
```

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

```
[1,] 3 0
```

```
[2,] 4 4
```

```
> forwardsolve(B,b)
```

```
[1] 0.33333333 -0.08333333
```

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

```
> det(A)
```

```
[1] -4
```

```
> solve(A)
```

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

```
[1,] -1 1.00
```

```
[2,] 1 -0.75
```

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

```
> X
```

```
[,1] [,2] [,3]
```

```
[1,] 1 2 3
```



```

[2,] 2 1 1
[3,] 4 1 2
> determinant(X)
$modulus
[1] 1.609438
attr(,"logarithm")
[1] TRUE
$sign
[1] -1
attr(,"class")
[1] "det"

```

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

```

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

```

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

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

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

```
> A=matrix(1:12,nrow=3);A
```

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

```
[1,] 1 4 7 10
```

```
[2,] 2 5 8 11
```

```
[3,] 3 6 9 12
```

```
> colSums(A)
```

```
[1] 6 15 24 33
```

```
> rowSums(A)
```

```
[1] 22 26 30
```

```
> colMeans(A)
```

```
[1] 2 5 8 11
```

```
> rowMeans(A)
```

```
[1] 5.5 6.5 7.5
```

Собственные вектора и собственные числа матриц находятся при помощи

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

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

```
> X
```

```

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

```

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

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

и

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

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

Операции с индексами

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

- i и j — *положительные целочисленные векторы*, тогда $A[i, j]$ — подматрица матрицы A , образованная элементами, стоящими на пересечении строк с номерами из вектора i и столбцов с номерами из j .

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

Возможен доступ к элементам матрицы с помощью одного индекса:

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

Можно задать имена столбцам и строкам матрицы

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

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

```
> A['a', 'C']
[1] 7
```

```

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

```

3.1.3. Многомерные массивы

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

Создадим массив размерности $3 \times 5 \times 4$

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

При вызове массив выводится послойно.

```

> A
, , 1
[,1] [,2] [,3] [,4] [,5]
[1,] 1 4 7 10 13
[2,] 2 5 8 11 14
[3,] 3 6 9 12 15
, , 2
[,1] [,2] [,3] [,4] [,5]
[1,] 16 19 22 25 28
[2,] 17 20 23 26 29
[3,] 18 21 24 27 30
, , 3
[,1] [,2] [,3] [,4] [,5]

```

```

[1,] 31 34 37 40 43
[2,] 32 35 38 41 44
[3,] 33 36 39 42 45
, , 4
[,1] [,2] [,3] [,4] [,5]
[1,] 46 49 52 55 58
[2,] 47 50 53 56 59
[3,] 48 51 54 57 60

```

Можно задать имена размерностям:

```

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

```

97

X1 X2 X3 X4 X5

A 46 49 52 55 58

B 47 50 53 56 59

C 48 51 54 57 60

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

Д.

```
> A[, , "Осень"]
```

X1 X2 X3 X4 X5

A 46 49 52 55 58

B 47 50 53 56 59

C 48 51 54 57 60

```
> A[, 'X2', "Зима"]
```

A B C

4 5 6

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

3.2. Работа со списками, фреймами данных, преобразование из одной структуры данных в другую

Задание

Запустить все команды из разделов 3.2.1-3.2.4. Выполнить упражнения из раздела 3.2.5.

3.2.1. Списки

Списки в R — это коллекции объектов, доступ к которым можно производить по номеру или имени. Список может содержать объекты (компоненты) разных типов, что отличает списки от векторов. Компонентами списка могут быть в том числе векторы и другие списки.

Функция

```
> list(объект1, объект2, ...)
```

создаёт список, содержащий указанные объекты.

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

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

Выведем на экран.

```
>writer
```

```
[[1]]
```

```
[1] "Шекспир"
```

```
[[2]]
```

```
[1] "Уильям"
```

```
[[3]]
```

```
[1] "1564"
```

```
[[4]]
```

```
[1] "1616"
```



```
[[5]]
[1] TRUE
[[6]]
[1] "драматург"
[[7]]
[1] "Хатауэй Анна"
[[8]]
[1] 3
[[9]]
[1] "Ромео и Джульетта"
[[10]]
[1] "Гамлет"
[[11]]
[1] "Отелло"
```

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

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

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

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

Можно создавать новые компоненты:

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

```

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

```

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

```

> names(writer) <- c("фамилия", "имя", "год рождения", "год смерти", "семейный
статус", "профессия", "имя жены", "число детей", "произведение1",
"произведение2", "произведение3")

writer
$фамилия
[1] "Шекспир"

$имя
[1] "Уильям"

$`год рождения`

```

```

[1] "1564"
$`год  сметри`
[1] "1616"
$`семейный статус`
[1] TRUE
$профессия
[1] "драматург"
100
$`имя  жены`
[1] "Хатауэй Анна"
$`число  детей`
[1] 3
$произведение1
[1] "Ромео и Джульетта"
$произведение2
[1] "Гамлет"
$произведение3
[1] "Отелло"
$<NA>
[1] "Англия"

```

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

```

> names(writer)[12]='Страна'
> writer
$фамилия
[1] "Шекспир"
$имя
[1] "Уильям"
$`год  рождения`
[1] "1564"
$`год  сметри`
[1] "1616"
$`семейный статус`
[1] TRUE

```

```
$профессия
[1] "драматург"
$`имя жены`
[1] "Хатауэй Анна"
$`число детей`
[1] 3
$произведение1
[1] "Ромео и Джульетта"
$произведение2
[1] "Гамлет"
$произведение3
[1] "Отелло"
$Страна
101
Страна
"Англия"
```

Имена компонент списка можно указать сразу при его создании:

```
writer=list(фамилия="Шекспир",имя="Уильям",'год рождения'="1564", 'год
смерти'="1616", 'семейное положение'=Т, профессия="драматург",'имя
жены'="Хатауэй Анна",'число детей'=3, произведение1="Ромео и Джульетта",
произведение2="Гамлет", произведение3="Отелло")
```

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

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

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

```
writer$имя жены
```

приведёт к появлению сообщения об ошибке.

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

```
writer[['имя']]
```

```
[1] "Уильям"
```

```
writer[["имя"]]
```

```
[1] "Уильям"
```

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

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

Правила использования индексов аналогичны соответствующим правилам для векторов. Важно понимать различие между правилами использования одинарных и двойных квадратных скобок:

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

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

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

```
> Pushkin <- list(name = "Александр Сергеевич Пушкин",  
year = 1799)
```

```
> Gogol <- list(name = "Николай Васильевич Гоголь",  
year = 1809)
```

```
> lst <- list(Pushkin, Gogol)
```

```
> lst[[1]]@name
[1] "Александр Сергеевич Пушкин"
> length(lst)
[1] 2
```

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

```
> clst <- c(Pushkin, Gogol)
> clst
@name
[1] "Александр Сергеевич Пушкин"
@year
[1] 1799
@name
[1] "Николай Васильевич Гоголь"
@year
[1] 1799
> length(clst)
[1] 4
> clst@name
[1] "Александр Сергеевич Пушкин"
> clst@year
[1] 1799
> clst[[3]]
[1] "Николай Васильевич Гоголь"
> clst[[4]]
[1] 1809
```

Функция `unlist(x, recursive, use.names = TRUE)` преобразовывает список `x` в вектор, элементами которого являются компоненты списка. Логический аргумент `recursive` определяет, нужно ли раскладывать компоненты списка на вектора (`logical`, `integer`, `real`, `complex`, `character`) или нет (по умолчанию `recursive = TRUE`). Логический аргумент `use.names` позволяет

сохранить имена компонент списка и присвоить их элементам создаваемого вектора (`use.names = TRUE` — значение по умолчанию).

Для матрицы `X` с помощью функции `determinant()` создадим список,

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

который преобразуем в вектор.

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

При преобразовании списка в вектор учитывается иерархия типов данных (`logical < integer < real < complex < character`). Соответственно, в результате преобразования будет получен вектор, чей тип данных определяется наибольшим согласно иерархии типом компоненты списка. В R предусмотрена и обратная функция `relist()`, позволяющая воссоздать список. Аргументы функции `relist()`:

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

Преобразуем полученный в предыдущем примере вектор `z`

```
> z
```

```
modulus sign
1.609438 -1.000000
```

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

```
> Z=list(c(1),c(1));Z
[[1]]
[1] 1
[[2]]
[1] 1
```

Воссоздаём список.

```
> Y=relist(z,Z);Y
[[1]]
[1] 1.609438
[[2]]
[1] -1
```

3.2.2. Факторы – *factor()*

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

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

```
> length(opros)
[1] 150
105
```

Выделим различные категории — варианты ответов.

```
> factor(opros)
```

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

```
[1] не знаю Андрей Рублев Иваново детство Сталкер
[5] Солярис Зеркало не знаю Андрей Рублев
[9] Иваново детство Сталкер Солярис Зеркало
[13] не знаю Андрей Рублев Иваново детство Сталкер
```


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

Levels: Андрей Рублев Зеркало Иваново детство не знаю Солярис Сталкер

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

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

Сократим исходный вектор `opros` до 30 элементов и воспользуемся `ordered()`.

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

```
> opros2
```

Сократили вектор и вывели его на экран.

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

Теперь проверим на упорядоченность категорий.

```
> is.ordered(opros2)
```

```
[1] FALSE
```

Используем `ordered()`

```
> opros3=ordered(opros2)
```

```
> opros3
```

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

```
6 Levels: Андрей Рублев < Зеркало < Иваново детство < не знаю <
... < Сталкер
```

и снова проверим на упорядоченность.

```
> is.ordered(opros3)
```

```
[1] TRUE
```

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

```
> is.factor(opros2)
```

```
[1] FALSE
```

```
> as.factor(opros2)
```

```
[1] Сталкер Солярис Зеркало не знаю
```

```
[5] Андрей Рублев Иваново детство Сталкер Солярис
```

```
[9] Зеркало не знаю Андрей Рублев Иваново детство
```

```
[13] Сталкер Солярис Зеркало не знаю
```

```
[17] Андрей Рублев Иваново детство Сталкер Солярис
```

```
[21] Зеркало не знаю Андрей Рублев Иваново детство
```

```
[25] Сталкер Солярис Зеркало не знаю
```

```
[29] Андрей Рублев Иваново детство Сталкер
```

```
Levels: Андрей Рублев Зеркало Иваново детство не знаю Солярис Сталкер
```

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

```
> x=rep(c(1,5,7,3,2),6);x
```

```
[1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
```

```
> y=factor(x);y
```

```
[1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
```

```
Levels: 1 2 3 5 7
```

```
> is.vector(y)
```

```
[1] FALSE
```

```
> z=as.vector(y);z
```

```
[1] "1" "5" "7" "3" "2" "1" "5" "7" "3" "2" "1" "5" "7" "3" "2" "1"
```

```
[17] "5" "7" "3" "2" "1" "5" "7" "3" "2" "1" "5" "7" "3" "2"
```

```
> is.vector(z)
```

```
[1] TRUE
```

Функция `gl()`

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

Полная запись

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

Разберём аргументы:

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

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

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

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

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

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

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

```
gl(5,3,15,labels=LETTERS[1:5])
[1] A A A B B B C C C D D D E E E
Levels: A B C D E
```

```
gl(5,3,15,labels=c('категория1','категория2','категория3','категория4','категория5'))
```

```
[1] категория1 категория1 категория1 категория2 категория2
категория2
```

```
[7] категория3 категория3 категория3 категория4 категория4
категория4
```

```
[13] категория5 категория5 категория5
```

```
Levels: категория1 категория2 категория3 категория4 категория5
```

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

```
gl(5,3,15,labels=c('категория1','категория2','категория3','категория4',  
'категория5'), ordered=T)
```

```
[1] категория1 категория1 категория1 категория2 категория2 категория2
```

```
[7] категория3 категория3 категория3 категория4 категория4 категория4
```

```
[13] категория5 категория5 категория5
```

```
Levels: категория1 < категория2 < категория3 < категория4 < категория5
```

3.2.3 Таблицы — `table()`

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

Применим к вектору `opros2` из предыдущего примера функцию `table()`.

```
> table(opros2)
```

```
opros2
```

```
Андрей Рублев Зеркало Иваново детство не знаю Солярис
```

```
5 5 5 5 5
```

```
Сталкер
```

```
6
```

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

Добавим в вектор `opros2` элементы `NA` и `NaN`.

```
> opros2=c(opros2,rep(c(NA,NaN),c(6,9)))
```

```
> opros2
```

```
[1] "Сталкер" "Солярис" "Зеркало" "не знаю"
```

```
[5] "Андрей Рублев" "Иваново детство" "Сталкер" "Солярис"
```

```

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

```

Построим таблицу, отбрасывая NA и NaN

```
> table(opros2)
```

```
opros2
```

```
Андрей Рублев Зеркало Иваново детство не знаю Солярис
5 5 5 5 5
```

```
Сталкер
```

```
6
```

и учитывая их

```
> table(opros2, exclude=NULL)
```

```
opros2
```

```
NaN Андрей Рублев Зеркало Иваново детство не знаю
9 5 5 5 5
```

```
Солярис Сталкер <NA>
```

```
5 6 6
```

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

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

```
> x=rep(c(1,5,7,3,2),6);x
```

```
[1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
```

```
> y=factor(x);y
```

```
[1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
Levels: 1 2 3 5 7
```

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

```
> is.table(x)
```

```
[1] FALSE
```

```
> is.table(y)
```

```
[1] FALSE
```

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

```
> z=as.table(x);z
```

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1
A1 B1 C1 D1
5 7 3 2
```

```
> is.table(z)
```

```
[1] TRUE
```

Аналогично и с фактором `y`.

```
> z=as.table(y);z
```

```
Ошибка в as.table.default(y) : не могу преобразовать в таблицу
```

```
> is.table(z)
```

```
[1] FALSE
```

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

```
> as.vector(z)
```

```
[1] 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2 1 5 7 3 2
```

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

3.2.3 Фреймы данных (таблицы данных) — *data frames*

Фреймы (таблицы) данных (*data frames*) — один из самых важных типов данных в R, позволяющий объединять данные различных типов вместе. Если слегка упростить, то таблица данных — это двумерная таблица, в которой (в отличие от числовых матриц), разные столбцы могут содержать данные разных типов (но все данные в одном столбце имеют один тип). Например, такая таблица может содержать результаты эксперимента. Создать фрейм данных можно с помощью функции `data.frame()`:

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

Здесь многоточие означает, что список данных может содержать произвольное число элементов. В качестве данных (`data1, data2, . . .`) могут выступать векторы (числовые, символьные или логические), факторы, матрицы (числовые, символьные или логические), списки или другие таблицы данных.

При этом все векторы должны иметь одинаковую длину, а матрицы и таблицы — одинаковое (такое же) число строк. Могут также встречаться векторы, длина которых меньше, но в этом случае эта длина должна являться делителем максимальной встречающейся длины. То же требование предъявляется к компонентам списков. Функция `data.frame` просто собирает все данные вместе. Символьные векторы конвертируются в факторы. Остальные данные собираются во фрейм такими, какие они есть.

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

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

При обращении получим следующую таблицу:

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


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

```
n = c(FALSE, TRUE, FALSE, FALSE)
```

```
y=c(2,1,3,2)
```

```
Y1=data.frame(Y,n,y)
```

```
Y1
```

```
год рождения год поступления n y
```

```
Иванов 1985 2002 FALSE 2
```

```
Ульянов 1987 2004 TRUE 1
```

```
Краснова 1986 2003 FALSE 3
```

```
Устюгов 1984 2001 FALSE 2
```

Переименуем последние столбцы и снова выведем таблицу

```
colnames(Y1)[3] = "задолженность"
```

```
colnames(Y1)[4] = "курс"
```

```
Y1
```

```
год рождения год поступления задолженность курс
```

```
Иванов 1985 2002 FALSE 2
```

```
Ульянов 1987 2004 TRUE 1
```

```
Краснова 1986 2003 FALSE 3
```

```
Устюгов 1984 2001 FALSE 2
```

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

```
> lst = list(1:4,FALSE, c("A", "B"))
```

```
> data.frame(lst)
```

```
X1.4 FALSE. c..A....B..
```

```
1 1 FALSE A
```

```
2 2 FALSE B
```

```
3 3 FALSE A
```

```
4 4 FALSE B
```

Доступ к элементам таблицы осуществляется двумя способами: в «матричном» или «списковом» стилях. В первом случае указываются номера или имена строки и столбца. Во втором таблица рассматривается как список, элементами которого являются столбцы таблицы, и чтобы обратиться к его элементу нужно указать имя или номер этого элемента (т. е. имя или номер столбца), а затем имя или номер строки.

```
Y1["Иванов", "курс"]
```

```
[1] 2
```

```
Y1["Устюгов", ]
```

```
год.рождения год.поступления задолженность курс
```

```
Устюгов 1989 2005 FALSE 2
```

```
Y1[4, 3]
```

```
[1] FALSE
```

```
Y1[, c(2, 4)]
```

```
год.поступления курс
```

```
Иванов 2005 2
```

```
Ульянов 2005 1
```

```
Краснова 2006 3
```

```
Устюгов 2005 2
```

```
Y1$курс
```

```
[1] 2 1 3 2
```

```
Y1[["задолженность"]][3]
```

```
[1] FALSE
```

```
Y1[["задолженность"]][2]
```

```
[1] TRUE
```

Редактировать таблицы данных можно и в редакторе.

3.2.4. Ввод-вывод данных в R

Рассмотрим следующие функции, позволяющие вводить данные в R:

- `scan()`;
- `read.table()`
- `read.csv()`.

Функция `scan()`

Данные считываются в вектор или в список с консоли или из файла.

Полный вид:

```
scan(file = "", what = double(0), nmax = -1, n = -1, sep = "",
quote = if(identical(sep, "\n")) "" else "'\"", dec = ".",
skip = 0, nlines = 0, na.strings = "NA",
flush = FALSE, fill = FALSE, strip.white = FALSE,
quiet = FALSE, blank.lines.skip = TRUE, multi.line = TRUE,
comment.char = "", allowEscapes = FALSE,
encoding = "unknown")
```

Аргументы:

- `file` — имя файла, откуда считываются данные; если `file=`, то производится ввод с клавиатуры; если задано только имя файла, то он (файл) ищется в текущей директории; иначе задаётся полный путь к файлу; может быть и URL.
- `what` — задаётся тип считываемых данных: `logical`, `integer`, `numeric`, `complex`, `character`, `list`; если считывается список, то строки в файле воспринимаются как поля списка указанных выше типов;
- `nmax` — целое положительное число — максимальное число данных для чтения или максимальное число записей в списке; при пропуске этого аргумента или при неправильном его задании файл считывается до конца;
- `n` — целое положительное число — максимальное число данных для чтения; неправильные значения или не типа `integer` игнорируются
- `sep` — разделитель полей; **по умолчанию - пробел**;
- `quote` — вид кавычек (двойные или одинарные);
- `dec` — десятичный разделитель (точка или запятая);
- `skip` — целое положительное число — число строк файла, которые следует пропустить перед чтением;

- `nlines` — целое положительное число — максимальное число строк для считывания;
- `na.strings` — символьный вектор — его элементы интерпретируются как пропущенные значения NA; пустые поля по умолчанию считываются как NA;
- `flush` — логический аргумент — значение TRUE позволяет добавлять комментарии к считываемым данным после последнего считанного поля (но не более одного);
- `fill` — логический аргумент — значение TRUE добавляются пустые поля к строкам, в которых количество полей данных меньше определённого параметром `what`;
- `strip.white` — логический вектор; используется, только если задан параметр `sep`, удаляет пустое пространство (пробел) перед символьными переменными и после них;
- `quiet` — логический аргумент; при значении FALSE функция выведет сообщение о том, сколько элементов было прочитано;
- `blank.lines.skip` — логический аргумент; при значении TRUE пустые строки игнорируются (не считываются) (заметим, что параметры `skip` и `nlines` всё равно будут учитывать все пустые строки);
- `multi.line` — логический аргумент; используется, если аргумент `what` принимает значение `list`; при значении FALSE все записи будут считаны в одну строку; если же и `fill=T`, то чтение при достижении конца строки будет прекращено;
- `comment.char` — символьный аргумент, определяет знак комментария;
- `allowEscapes` — логический аргумент — нужно ли следующие последовательности символов `\n`, `\a`, `\b`, `\f`, `\r`, `\t`, `\v` при чтении рассматривать как команды (TRUE) или просто как символы (FALSE) ;
- `encoding` — символьный аргумент, задаёт кодировку считываемого файла.

Функции `read.table()` и `read.csv()`

Если исходные данные представлены в виде таблицы и итоговый результат должен быть таблицей (фреймом данных), то удобнее воспользоваться функцией `read.table()` либо `read.csv()`.

Полная запись функции:

```
read.table(file, header = FALSE, sep = "", quote = "\"'",  
dec = ".", row.names, col.names,  
as.is = !stringsAsFactors,  
na.strings = "NA", colClasses = NA, nrows = -1,  
skip = 0, check.names = TRUE, fill = !blank.lines.skip,  
strip.white = FALSE, blank.lines.skip = TRUE,  
comment.char = "#",  
allowEscapes = FALSE, flush = FALSE,  
stringsAsFactors = default.stringsAsFactors(),  
fileEncoding = "", encoding = "unknown")
```

Упрощённые варианты функции:

```
read.csv(file, header = TRUE, sep = ",", quote="\"", dec=".",  
fill = TRUE, comment.char="", ...)  
read.csv2(file, header = TRUE, sep = ";", quote="\"", dec="," ,  
fill = TRUE, comment.char="", ...)  
read.delim(file, header = TRUE, sep = "\t", quote="\"", dec=".",  
fill = TRUE, comment.char="", ...)  
read.delim2(file, header = TRUE, sep = "\t", quote="\"", dec="," ,  
fill = TRUE, comment.char="", ...)
```

которые особенно полезны при считывании данных из файлов Excel. Для этого исходный файл сохраняется в формате `.csv` и затем прочитывается при помощи одной из четырёх приведённых выше функций (лучше воспользоваться `read.csv2()`).

Аргументы:

- `file` — обязательный аргумент, имя файла;
- `header` — логический параметр; при значении `TRUE` считываются имена переменных из файла;
- `sep` — разделитель полей; по умолчанию - пробел;
- `quote` — вид кавычек (двойные или одинарные);

- `dec` — десятичный разделитель в числах (точка или запятая);
- `row.names` — вектор имён строк; представляет собой либо вектор с именами строк итоговой таблицы; либо число — номер столбца исходной таблицы с названиями строк; либо имя столбца считываемой таблицы, где приведены названия строк; если этот параметр не задан, то строки в итоговой таблице будут пронумерованы;
- `col.names` — вектор имён столбцов в итоговой таблице; по умолчанию — «V<номер столбца>»;
- `as.is` — нужно ли символьные переменные, не преобразованные в числовые или логические, переводить в факторы. `as.is` — либо логический, либо числовой вектор, определяющий столбцы, неконвертируемые в факторы.
- `colClasses` — символьный вектор; определяет классы данных в столбцах (символьные, логические, числовые, даты). Возможные значения: `NA` — автоматическая конвертация типов данных, `NULL` — столбец пропускается (данные не преобразовываются), тип данных в который будут переведены элементы столбца, `factor`;
- `na.strings` — символьный вектор, элементы которого при чтении исходной таблицы в файле будут интерпретироваться как `NA`;
- `nrows` — целочисленный аргумент; определяет максимальное число считываемых строк;
- `skip` — положительный целочисленный аргумент; определяет число строк, пропускаемых перед чтением;
- `check.names` — логический аргумент; при значении `TRUE` имена переменных будут проверены на синтаксическую правильность и отсутствие дублирования;
- `fill` — логический аргумент; при значении `TRUE` строки разной длины будут приведены к единой (максимальной) добавлением пустых полей;
- `strip.white` — логический аргумент; используется только если определён разделитель `sep`, позволяет убирать пробелы перед и после символьных переменных;

Примечания:

- Функция `read.table()` является основной для считывания данных из таблиц.
- Поле таблицы считается пустым, если в нём ничего нет (до знака комментария или до символа окончания строки).
- Если параметр `row.names` не определён (т.е. не заданы имена строк результирующей таблицы), а длина заголовка на единицу меньше числа столбцов, то первый столбец будет рассматриваться как столбец с названиями строк.
- Число столбцов в считываемой таблице определяется автоматически после прочтения первых пяти строк.
- Всё, что находится в исходной таблице после знаков комментария, не считывается.
- Задание параметра `nrows` (пусть даже с очень избыточными значениями) позволяет уменьшить затраты памяти.
- Для чтения больших матриц предпочтительнее использовать функцию `scan()`.

Вывод данных в R

Здесь мы рассмотрим следующие функции:

- `write();`
- `cat();`
- `write.table();`
- `write.csv();`
- `write.csv();`

Функция `write()`

Функция `write()` предназначена для записи данных (в основном матриц) в R. Её вид:

```
write(x, file = "data",
ncolumns = if(is.character(x)) 1 else 5,
append = FALSE, sep = " ")
```

Аргументы:

- `x` — данные, которые нужно записать;

- `file` — имя файла, куда будет записана информация;
- `ncolumns` — число столбцов, в которых будет записана информация;
- `append` — логический аргумент — если значение `TRUE`, то данные допишутся в исходный файл, если же `FALSE`, то файл будет переписан заново;
- `sep` — разделитель столбцов (`\t` — табуляция) .

Если в качестве аргумента функции задать только данные — `write(x)`, то они будут выведены на экран.

Функция `cat()`

Функция `cat()` имеет вид

```
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL,
append = FALSE)
```

Её аргументы:

- . . . — объекты R, которые будут записаны в файл.
- `file` — имя файла, куда будет записана информация; если этот аргумент отсутствует, то данные будут выведены на экран;
- `sep` — разделитель элементов записываемого объекта;
- `fill` — логический или положительный целочисленный аргумент — контролирует создание новых строк в записываемом файле. Если `fill` — числовой, то задаётся длина строки (количество символов в строке). Если `fill` — логический и его значение `FALSE`, то новые строки создаются только при наличии в записываемых данных символа `"\n"`; если значение `TRUE`, то задаётся дополнительный аргумент `width`, определяющий длину создаваемой в файле строки;
- `labels` — символьный вектор, задающий названия строк. Игнорируется, если аргумент `fill=FALSE`.
- `append` — логический аргумент — используется, если задано имя файла. Если значение аргумента `TRUE`, то новые данные добавляются к исходному файлу, если `FALSE`, то записываются вместо старых.

Функция `cat()` преобразовывает исходные данные в символьные, объединяет их в единый символьный вектор и записывает в заданный файл.

Функции `write.table()`, `write.csv()` и `write.csv2()`

Функция `write.table()` записывает таблицу данных (или матрицу) в заданный файл. Если записываемый объект не является фреймом данных, то он автоматически будет конвертирован.

Вид функции:

```
write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ",  
eol = "\n", na = "NA", dec = ".", row.names = TRUE,  
col.names = TRUE, qmethod = c("escape", "double"))
```

Разновидности:

```
write.csv(...)
```

```
write.csv2(...)
```

Аргументы:

- `x` — записываемый объект. Предпочтительнее, чтобы это была матрица или таблица (фрэйм) данных.
- `file` — имя файла, в который будут записаны данные.
- `append` — логический аргумент — используется, если только задано имя файла. Если значение аргумента `TRUE`, то новые данные добавляются к исходному файлу, если `FALSE` — записываются вместо старых.
- `quote` — логический или числовой аргумент. Если `quote` является логическим аргументом и его значение `TRUE`, то все символьные переменные и факторы будут записаны в файл в двойных кавычках. Если значение аргумента `FALSE` — символьные переменные и факторы записываются без кавычек. Если `quote` — числовой вектор, то его элементы задают номера столбцов, в которых данные должны быть записаны в кавычках. Названия столбцов и строк по умолчанию будут записаны в кавычках.
- `sep` — аргумент, определяющий разделитель полей; значения в каждой строке исходных данных разделяются этим символом.
- `eol` — символ окончания строки; для Windows и Unix - `"\n"`, `"\r"`; для MacOS - `"\r"`.
- `na` — символьный аргумент для обозначения отсутствующих элементов в исходных данных.

- `dec` — десятичный разделитель в числах (точка или запятая).
- `row.names` — аргумент, задающий названия строк в файле. Аргумент либо логический (если значение `TRUE`, то используются имена строк, указанные в исходных данных), либо представляет собой символьный вектор, непосредственно задающий имена строк.
- `col.names` — аналогичный аргумент, задающий имена столбцов.
- `qmethod` — символьный вектор, определяющий как поступать с вложенными друг в друга кавычками — `" "a" "`. Два значения — `escape` (внешние кавычки убираются - по умолчанию) и `double` (все кавычки остаются).

Для функций `write.csv()` и `write.csv2()` аргументы `col.names`, `sep`, `dec` и `qmethod` не могут быть изменены.

Примечания:

- Если в исходной записываемой таблице данных нет столбцов, то при записи имена строк будут присвоены только в том случае, когда `row.names=TRUE`, и наоборот.
- Действительные и комплексные числа записываются с максимальной возможной точностью.
- Если в исходной записываемой таблице данных имелись столбцы с матричной структурой, то при записи каждый из столбцов этой матрицы будет представлен в виде отдельного столбца; в связи с этим аргументы `col.names` и `quote` должны соответствовать числу столбцов результирующей таблицы, а не исходной.
- Каждый столбец в таблице данных, являющийся списком или датой, будет переконвертирован в символы.
- Только символьные столбцы (или столбцы, переконвертированные в символы) будут при записи заключены в кавычки (если это указано в аргументе `quote`).
- Аргумент `dec` применяется только к тем столбцам, которые не были переконвертированы в символьные (т.е. только к числовым данным).

* .CSV файлы

Функции `write.csv()` и `write.csv2()` используются для записи файлов в формате `*.csv`. Если параметр `row.names=TRUE`, то значения параметров `qmethod` и `col.names` устанавливается `NA`; если же `row.names=FALSE`, то `qmethod=col.names=TRUE`.

Для функции `write.csv()` параметр `dec` принимает значение `"."`, а параметр `sep` — `","`. Для функции `write.csv2()` десятичный разделитель `dec=","`, разделитель полей `sep=";"`.

Замечания.

- Попытки изменить значения по умолчанию параметров `col.names`, `sep`, `dec` и `qmethod` игнорируются и будет выдано предупреждение.

- Если таблица данных содержит большое число столбцов (более ста), функция `write.table()` работает медленно, так как каждый столбец может быть переменной своего класса и поэтому обрабатывается отдельно.

Создадим таблицу данных:

```
> x<-data.frame(a=letters[1:3], b=1:3);x
```

```
a b
```

```
1 a 1
```

```
2 b 2
```

```
3 c 3
```

```
> rownames(x)=LETTERS[1:3]
```

```
> x
```

```
a b
```

```
A a 1
```

```
B b 2
```

```
C c 3
```

и затем при помощи различных функций запишем эту таблицу и прочитаем созданный файл.

Запись

```
write.table(x,      file="example.csv",      sep=";",      row.names=T,  
col.names=NA,  
qmethod="double")
```

и чтение

```
> read.table("example.csv", header=T, sep="," , row.names=1)
```

```
a b
```

```
A a 1
```

```
B b 2
```

```
C c 3
```

или

```
> write.csv(x, file="example.csv", row.names=T)
```

```
> read.csv("example.csv", row.names=1)
```

```
a b
```

```
A a 1
```

```
B b 2
```

```
C c 3
```

3.2.5. Упражнения

Все данные доступны в пакете ISLR, за исключением Boston и USArrests (они входят в состав базового дистрибутива R).

Вариант 1.

Это упражнение касается набора данных College, который можно найти в файле College.csv. Он содержит несколько переменных для 777 университетов и колледжей США. К этим переменным относятся:

- Private: индикатор того, является ли заведение публичным или частным;
- Apps: количество полученных заявлений на поступление;
- Accept: количество принятых абитуриентов;
- Enroll: количество новых зарегистрированных студентов;
- Top10perc: количество новых студентов, входивших в список 10% лучших учеников своего класса в средней школе;
- Top25perc: количество новых студентов, входивших в список 25% лучших учеников своего класса в средней школе;
- F.Undegrad: количество студентов, занимающихся полный рабочий день;

- `P.Undegrad`: количество студентов, занимающихся неполный рабочий день;
- `Outstate`: плата за обучение для студентов из других штатов;
- `Room.Board`: плата за проживание и питание;
- `Books`: стоимость учебников;
- `Personal`: примерные затраты на личные нужды;
- `PhD`: процент преподавателей с ученой степенью;
- `Terminal`: процент преподавателей с самой высокой ученой степенью в соответствующей области науки;
- `S.F.Ratio`: соотношение между численностью студентов и преподавателей;
- `perc.alumni`: процент бывших выпускников, делающих денежные пожертвования;
- `Expend`: затраты заведения на обучение одного студента;
- `Grad.Rate`: процент студентов, завершающих полный цикл обучения.

Перед загрузкой данных в R их можно просмотреть в программе Excel или в текстовом редакторе.

1. Примените функцию `read.csv()` для загрузки данных в R. Присвойте загруженным данным имя `college`. Убедитесь, что Вы задали правильную директорию, в которой хранятся эти данные.

2. Просмотрите данные при помощи функции `fix()`. Вы должны заметить, что первый столбец просто содержит названия университетов. Мы не очень хотим, чтобы этот столбец рассматривался R как данные. Однако эти названия могут оказаться полезными позднее. Попробуйте следующие команды:

```
> rownames(college) = college[, 1]
> fix(college)
```

Вы должны увидеть, что теперь появился столбец `row.names` с названиями всех исследованных университетов. Это означает, что система R присвоила каждой строке имя, соответствующее тому или иному университету. R не

будет пытаться выполнять вычисления над именами строк. Однако нам все еще нужно удалить первый столбец, в котором хранятся названия университетов. Попробуйте

```
> college = college[, -1]
> fix(college)
```

Теперь вы должны видеть, что первым столбцом в данных является `Private`. Заметьте, что перед столбцом `Private` появился еще один столбец с заголовком `row.names`. Однако этот столбец не является частью данных — это имена, присвоенные R каждой строке.

3. Примените функцию `summary()` для вывода количественной сводки по имеющимся в таблице переменным.

4. Создайте новую качественную переменную `Elite`, разбив на классы переменную `Top10perc`. Мы собираемся разделить университеты на две группы в зависимости от того, превышает ли 50% доля студентов, входивших в список 10% лучших учеников своих классов в средней школе.

```
> Elite = rep("No", nrow(college))
> Elite[college$Top10perc > 50] = "Yes"
> Elite = as.factor(Elite)
> college = data.frame(college, Elite)
```

Примените функцию `summary()` для нахождения числа элитных университетов.

Упражнение 2.

Это упражнение касается набора данных `Boston`.

1. Данные `Boston` являются частью пакета `MASS`.

```
> library(MASS)
```

Теперь данные содержатся в объекте `Boston`.

```
> Boston
```

Почитайте об этом наборе данных:

```
> ?Boston
```

Сколько строк в этом наборе данных? Сколько столбцов? Что представляют собой эти строки и столбцы?

2. Какие из пригородов Бостона имеют особенно высокие уровни преступности? Налоговые ставки (`tax`) ? Отношение численности учащихся к численности учителей (`ptratio`)?
3. Через сколько пригородов из этого набора данных протекает река Чарльз (`chas`)?
4. У какого количества пригородов из этого набора данных среднее число комнат в доме (`rm`) больше семи? Больше восьми? Опишите пригороды, где число комнат в доме в среднем выше восьми.

4.1. Генерация псевдослучайной выборки для заданного распределения. Моделирование данных с заданными статистическими характеристиками.

Задание

Выполнить все команды и упражнения из разделов 4.1.1-4.1.4.

4.1.1. Базовый пример

Если Вы хотите получать одинаковые результаты от генератора случайных чисел, необходимо указать стартовую позицию. Это важно, чтобы можно было повторить результаты имитации и для отладки кода.

Попробуйте выполнить команды:

```
>vecpoisson=rpois(100,5)
>mean(vecpoisson)
## [1] 4.74
```

Если это делать много раз, Вы получите разные значения. Если рассмотреть выборку полученных значений, то Вы получите выборку распределения средних значений 100 случайных переменных для распределения Пуассона.

Чтобы выдавался один и тот же набор случайных чисел используйте функцию `set.seed()`.

```
>set.seed(198911)
>vecpoisson=rpois(100,5)
>mean(vecpoisson)
## [1] 4.8
>set.seed(198911)
>vecpoisson=rpois(100,5)
>mean(vecpoisson)
## [1] 4.8
```

4.1.2. Применение функции `apply()`

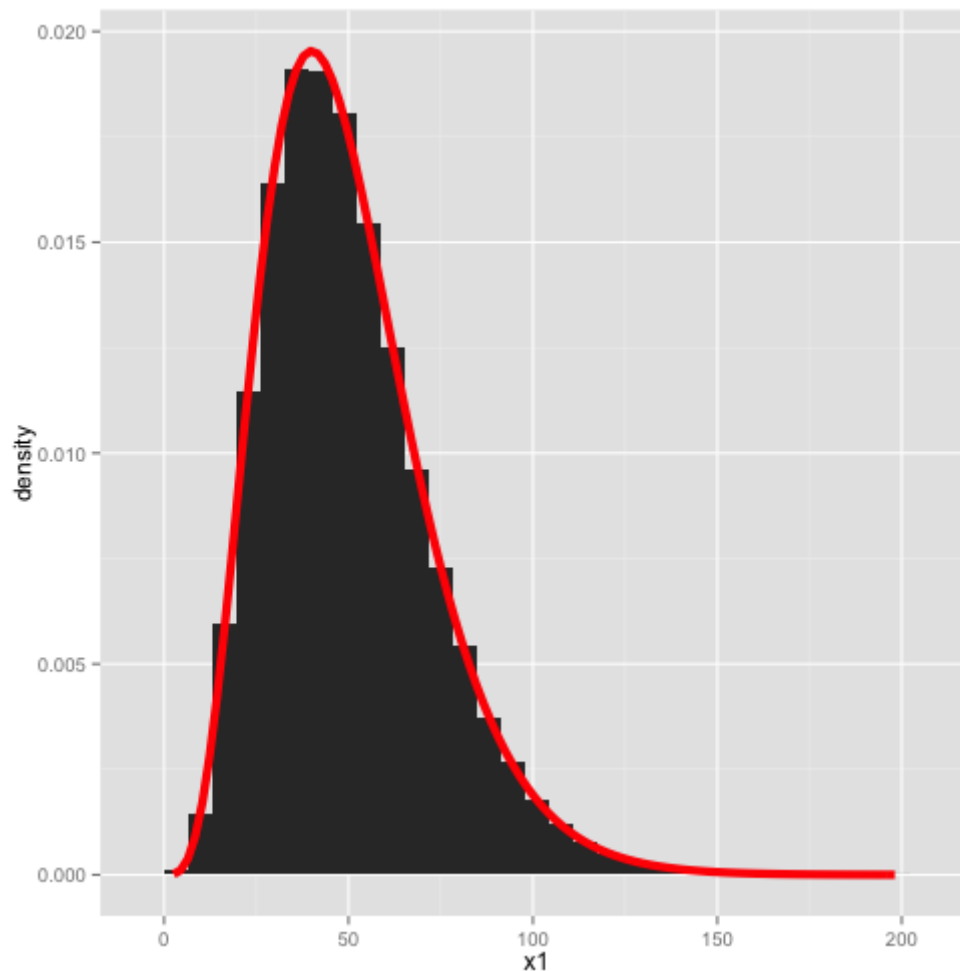
Чтобы не использовать медленные циклы в R для имитационного моделирования лучше использовать такие функции как `apply()`.

Сгенерируем 5 случайных значений экспоненциального распределения с параметром 0.1 и сложим их вместе: `sum(rexp(n=nexps, rate=rate))`. Известно, что сумма экспоненциально распределенных случайных величин дает гамма-распределение. Функция `replicate()` позволяет выполнять одни и те же действия, не увеличивая слишком код. Ниже выполним 10000 раз наборов из 5 чисел.

```
reps <- 50000
nexps <- 5
rate <- 0.1
set.seed(0)
system.time(
  x1 <- replicate(reps, sum(rexp(n=nexps, rate=rate)))
) # replicate
##      user  system elapsed
##    0.304    0.003    0.307
head(x1)
## [1] 38.01 42.64 85.13 58.21 16.67 81.43
```

Чтобы проверить получение гамма-распределения, построим гистограмму распределения `qqplot`.

```
require(ggplot2)
ggplot(data.frame(x1), aes(x1)) +
  geom_histogram(aes(y=..density..)) +
  stat_function(fun=function(x) dgamma(x, shape=nexps, scale=1/rate),
               color="red", size=2)
plot of chunk unnamed-chunk-6
```



Можно сделать проверку гамма-распределения можно использовать другие команды, откройте справку `help(replicate)` и Вы увидите, что есть такие функции как `sapply()`, `lapply()` и `vapply()`. Они связаны в свою очередь с функциями `apply()` и `tapply()`.

Код с функцией `sapply()` будет следующим:

```
set.seed(0)

system.time(x1 <- sapply(1:reps, function(i){sum(rexp(n=nexps, rate=rate))}))
# simple apply

##      user  system elapsed
## 0.277    0.017    0.295

head(x1)

## [1] 38.01 42.64 85.13 58.21 16.67 81.43
```

`lapply()`:

```
set.seed(0)

system.time(x1 <- lapply(1:reps, function(i){sum(rexp(n=nexps, rate=rate))}))
# list apply

##      user  system elapsed
```

```
##    0.231    0.001    0.232
head(x1)
## [[1]]
## [1] 38.01
##
## [[2]]
## [1] 42.64
##
## [[3]]
## [1] 85.13
##
## [[4]]
## [1] 58.21
##
## [[5]]
## [1] 16.67
##
## [[6]]
## [1] 81.43
```

Если мы применяем простую функцию как `sum()`, как правило, лучше сделать матрицу со всеми данными имитации и применить данную функцию к соответствующим частям матрицы. Функции `rowSums()` и `colSums()` демонстрируют особую эффективность в этом случае.

```
set.seed(0)
system.time(x1 <- apply(matrix(rexp(n=nexps*reps, rate=rate), nrow=nexps), 2, sum))
# apply on a matrix
##      user  system elapsed
##    0.114    0.003    0.117
head(x1)
## [1] 38.01 42.64 85.13 58.21 16.67 81.43
set.seed(0)
system.time(x1 <- colSums(matrix(rexp(n=nexps*reps, rate=rate), nrow=nexps)))
# using colSums
##      user  system elapsed
##    0.018    0.001    0.018
head(x1)
## [1] 38.01 42.64 85.13 58.21 16.67 81.43
```

Если у Вас многоядерный процессор, то скорость обработки можно повысить посредством параллельной обработки данных. В пакете `parallel` есть функция `mclapply()`, которая является параллельным вариантом `lapply()`. По умолчанию используется один процессор, для изменения числа процессор измените `mc.cores`.

```
require(parallel)

set.seed(0)

system.time(x1 <- mclapply(1:reps, function(i){sum(rexp(n=nexps,
rate=rate))})) # multi-cluster apply

##      user  system elapsed
##  0.385    0.095    0.250

head(x1)

## [[1]]
## [1] 28.92
##
## [[2]]
## [1] 67.89
##
## [[3]]
## [1] 26.47
##
## [[4]]
## [1] 37.11
##
## [[5]]
## [1] 31.21
##
## [[6]]
## [1] 24.9
```

4.1.3 Генерация нормально-распределенных случайных величин

Пример получения 1000 случайных значений нормального распределения $N(0,1)$:

```
samples = rnorm(1000, 0, 1)
```

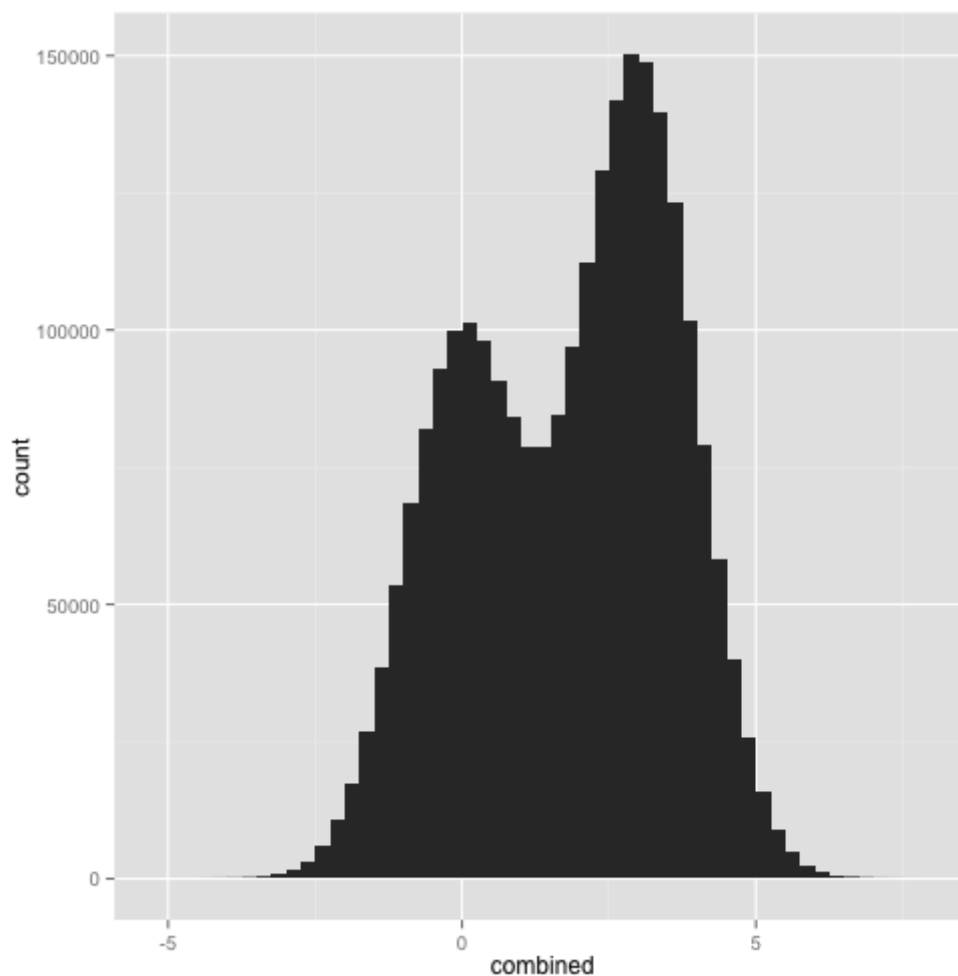
Возьмем случайную выборку из 1,000,000 значений для нормального распределения с математическим ожиданием 0 и стандартным

отклонением 1 $N(0,1)$ и выборку размером 1,500,000 $N(3,1)$. Построим гистограмму с шириной столбца 0.25:

```
require(ggplot2)
sampa=rnorm(1000000,0,1)
sampb=rnorm(1500000,3,1)
combined = c(sampa, sampb)

plt = ggplot(data.frame(combined), aes(x=combined)) +
  stat_bin(binwidth=0.25, position="identity")

plt
```



Упражнение. Ответьте на вопросы

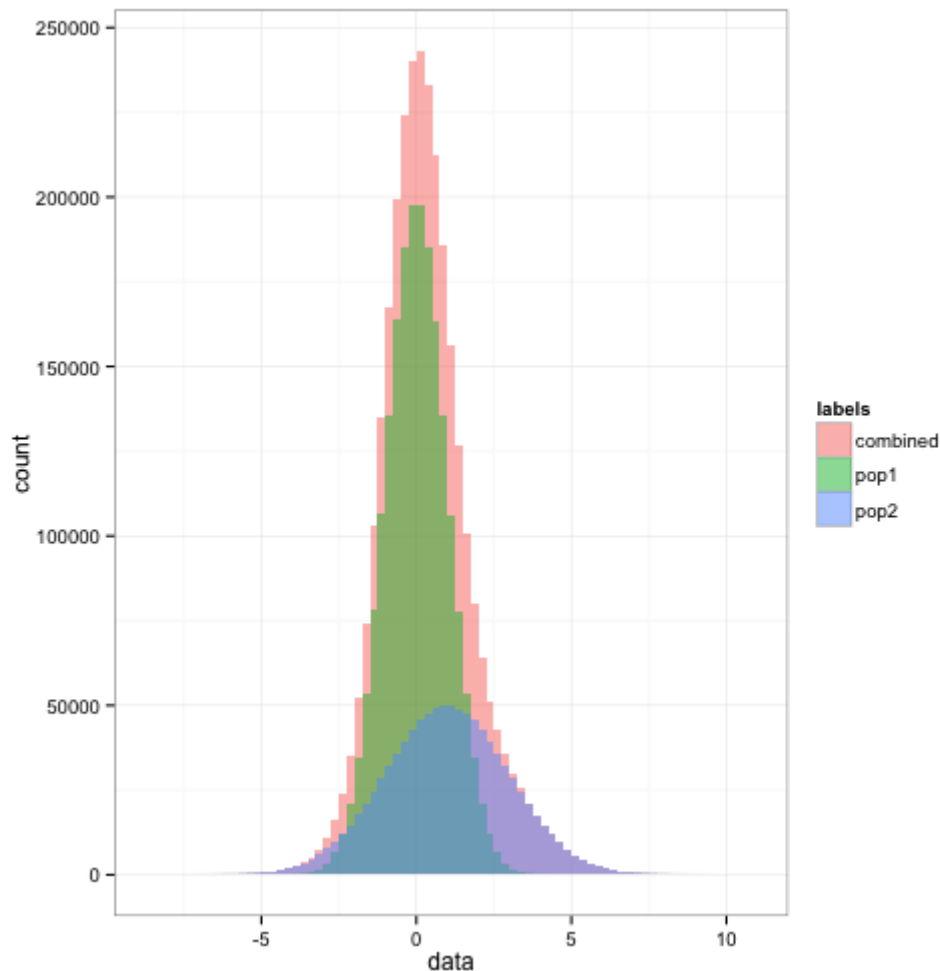
- Сколько у данного распределения мод. Почему так произошло?
- Нарисуйте график распределения частот, используя `geom_density()`.

Теперь построим 3 000 000 точек из смеси двух случайных выборок для нормального распределения.

```
pop1=rnorm(2000000)
pop2=rnorm(1000000, 1, 2)
combined = c(pop1, pop2)

plt= ggplot(data.frame(data=c(combined, pop1, pop2), labels=rep(c("combined", "pop1",
"pop2"), c(3e6, 2e6, 1e6))), aes(x=data)) + stat_bin(aes(fill=labels),
position="identity", binwidth=0.25, alpha=0.5) + theme_bw()

plt
```



Упражнение. Ответьте на вопросы

3) Какое отличие от предыдущей гистограммы.

4) Постройте диаграмму частот для pop1 и pop2 на одном графике.

Сгенерируем два многомерных нормальных кластера из 100 000 точек каждый:

```
require(MASS)
```

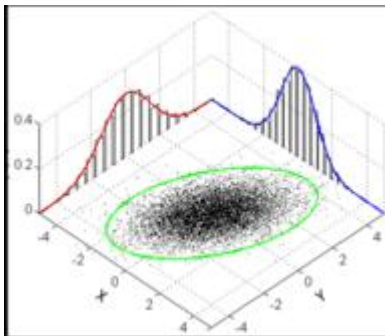
```

Sigma=matrix(c(5,3,3,2),2,2)
ex1=mvrnorm(100000,rep(0,2),Sigma)
Sigma=matrix(c(9,-5,-1,5),2,2)
ex2=mvrnorm(n=100000, rep(3, 2), Sigma)

```

Упражнение

5) Постройте функции частот распределения.



Для этого используйте `geom_point()` и `geom_density2d()`.

4.1.4. Метод Монте-Карло

Мы покажем, как вычислить вероятность простых событий с помощью имитационного моделирования методом Монте-Карло.

Предположим, мы бросили два кубика. Какова вероятность, что их сумма не меньше 7? Мы подойдем к этому, моделируя множество бросков двух честных игральных костей, а затем вычисляя долю тех испытаний, сумма которых равна не менее 7. Будет удобно написать функцию, которая имитирует испытания и возвращает TRUE, если сумма не менее 7 (мы называем это событием), в противном случае – FALSE (функция `sample()` возвращает случайную выборку).

```

isEvent = function(numDice, numSides, targetValue, numTrials){
  apply(matrix(sample(1:numSides, numDice*numTrials, replace=TRUE), nrow=numDice),
+2, sum) >= targetValue
}

```

Мы разработали функцию, будем применять её для метода Монте-Карло для 5 попыток.

```
set.seed(0)
```

```
#try 5 trials
outcomes = isEvent(2, 6, 7, 5)
mean(outcomes)
## [1] 1
```

Теоретически ответ должен быть $28/36$, т.е. около 0.6; наш ответ 1 слишком далек. Попробуем сделать 10 000 испытаний:

```
set.seed(0)
outcomes = isEvent(2, 6, 7, 10000)
mean(outcomes)
## [1] 0.5843
```

Как видим результат приблизился к теоретическому. В методе Монте-Карло используются простые вычисления, но их требуется достаточно много, поэтому лучше их распараллелить с помощью пакета `parallel`.

```
require(parallel)
```

Используем функцию `pvec()`, для этого сделаем пользовательскую функцию `isEventPar()`.

```
isEventPar = function(numDice, numSides, targetValue, trialIndices){
  sapply(1:length(trialIndices), function(x) sum(sample(1:numSides, numDice,
replace=TRUE)) >= targetValue)
}
```

Вызовем `pvec()` :

```
set.seed(0)
outcomes = pvec(1:10000, function(x) isEventPar(2, 6, 7, x))
mean(outcomes)
## [1] 0.5931
```

Результаты параллельного и обычного алгоритмов практически не отличаются за счет использования `set.seed(0)`.

4.2. Повышение эффективности работы алгоритмов

Задание

Выполнить все команды и упражнения из разделов 4.2.1-4.2.3.

4.2.1. Использование утилиты Rprof

Мы будем использовать библиотеки `profvis` и `bench`.

```
library(profvis)
```

```
library(bench)
```

Рассмотрим следующую функцию `f()`

```
f <- function() {  
  pause(0.1)  
  g()  
  h()  
}
```

```
g <- function() {  
  pause(0.1)  
  h()  
}
```

```
h <- function() {  
  pause(0.1)  
}
```

Использована функция `pause` из библиотеки `profvis`.

Если мы будем профилировать код для `f()`, останавливаясь каждые 0.1 сек, то мы увидели бы следующую выдачу

```
"pause" "f"  
"pause" "g" "f"  
"pause" "h" "g" "f"  
"pause" "h" "f"
```

Из этой выдачи было бы понятно, что код тратит 0.1 с на непосредственное выполнение `f()`, 0.2 с на `g()` и 0.1 с на `h()`.

Если использовать утилиту `Rprof()`, то мы получим следующее:

```
tmp <- tempfile()
Rprof(tmp, interval = 0.1)
f()
Rprof(NULL)
writeLines(readLines(tmp))
#> sample.interval=100000
#> "pause" "g" "f"
#> "pause" "h" "g" "f"
#> "pause" "h" "f"
```

При работе профилировщик замедляет выполнение кода, поэтому при профилировании необходимо обеспечить баланс между точностью выдачи профилировщика (а это требует вмешательство в работу программы) и замедлением из-за вмешательства. Поэтому каждый раз результаты профилирования немного отличаются друг от друга.

Профилировщик `Rprof()` создает достаточно большой объем выборки (больше 100 запусков) для функции, которая работает пару секунд. В то же время не хватает визуализации, которую обеспечивает библиотека `profvis`.

4.2.2. Визуализация процесса профилирования

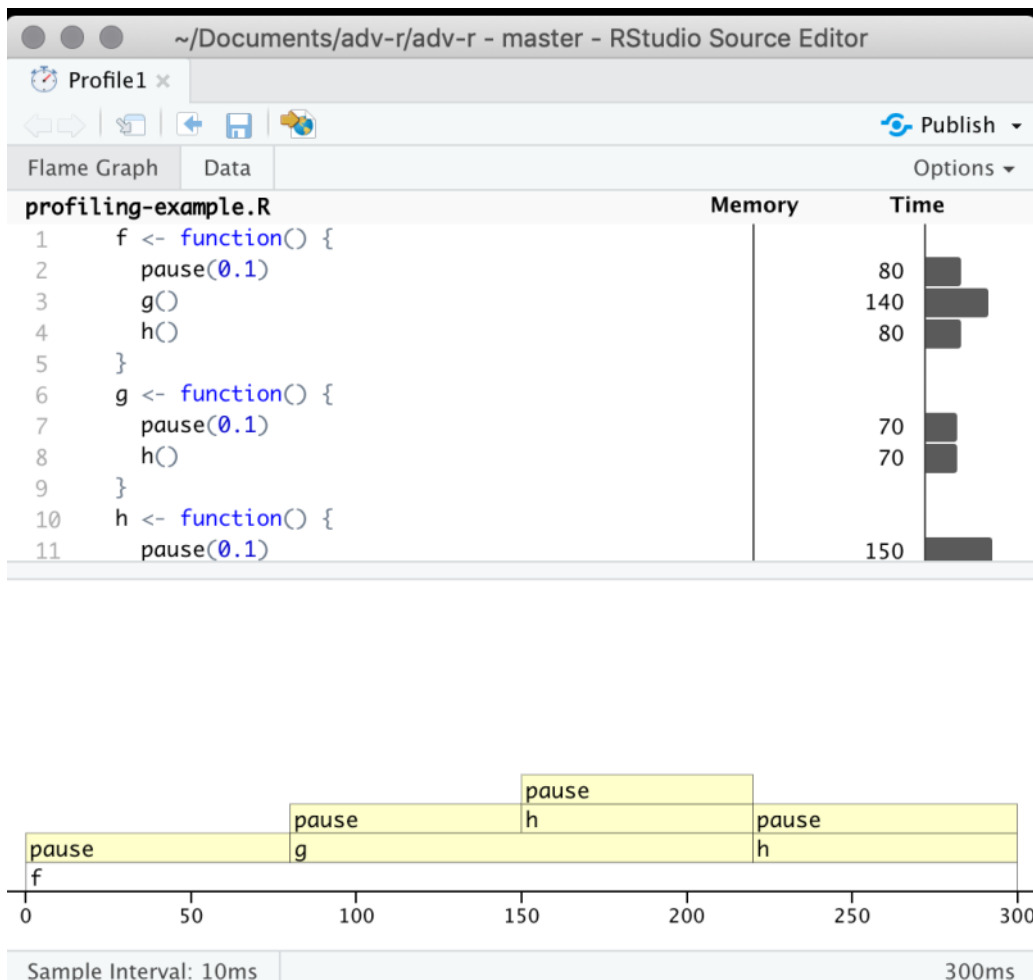
Запустить `profvis` можно 2 способами:

1. Из меню Profile в RStudio.

2. Подключить функцию `profvis()` из библиотеки `profvis`. Лучше код программы сохранить в файле `.r`, а затем сослаться на этот файл при профилировании.

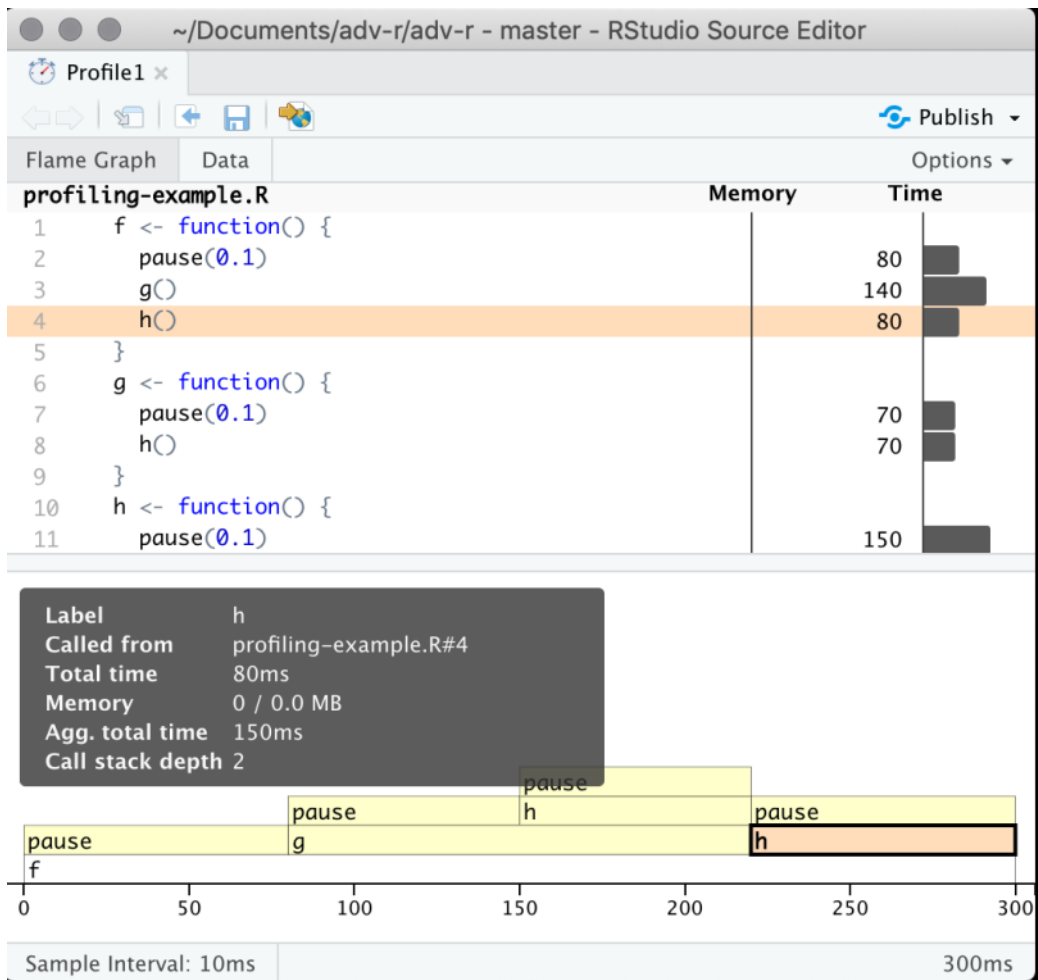
```
source("profiling-example.R")
profvis(f())
```

После профилирования `profvis` откроет интерактивный html-документ, где можно изучить результаты профилирования. Первая активная вкладка - это Flame Graph.

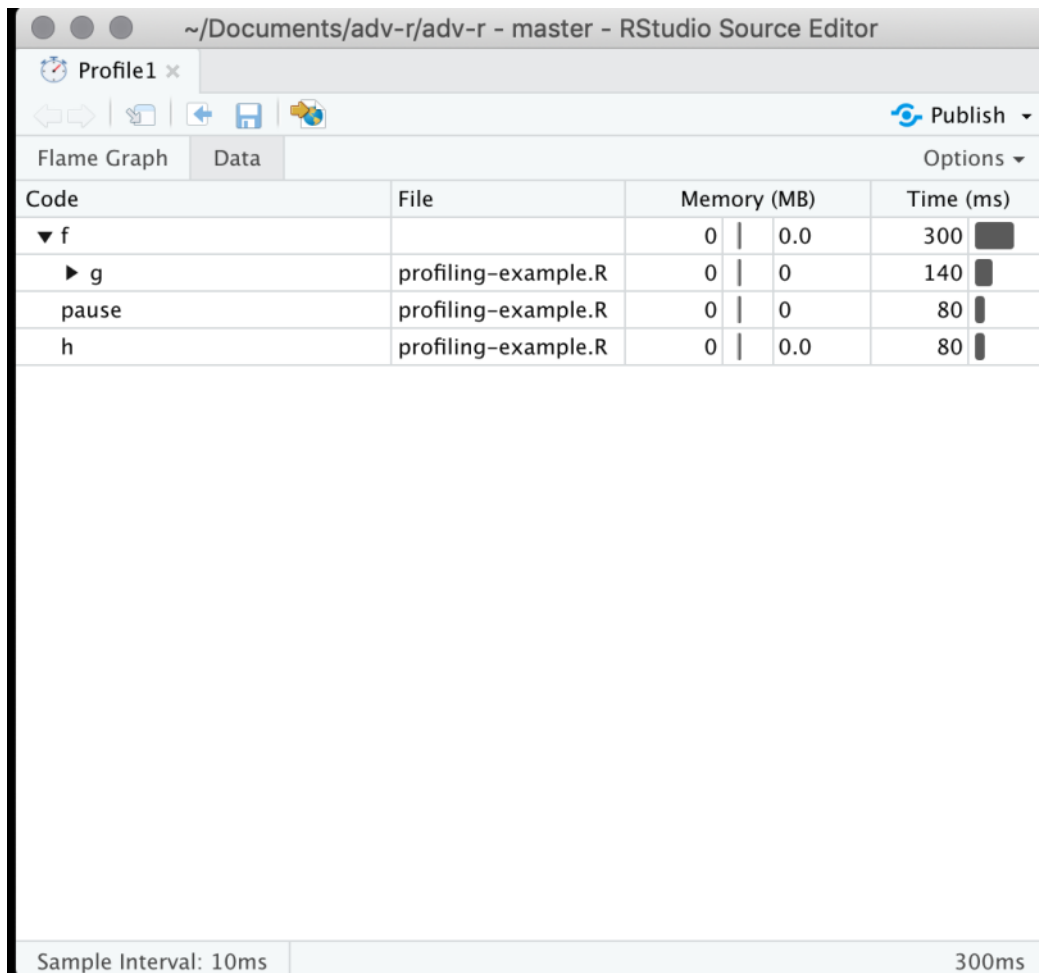


Вы можете увидеть использование памяти и времени разными функциями программы (Memory и Time справа). Можно увидеть в диаграмме использования времени, что функция `h()` вызывается 2 раза, именно поэтому общее время в два раза больше, чем у функции `g()`.

Внизу мы видим очередность вызова функций. Если Вы наведете мышь на конкретный вызов функции, то можно будет показана строка кода и более подробная информация об этом вызове.



Также есть вкладка о данных Data. Это по сути дело другое представление данных из Flame Graph для больших кодов.

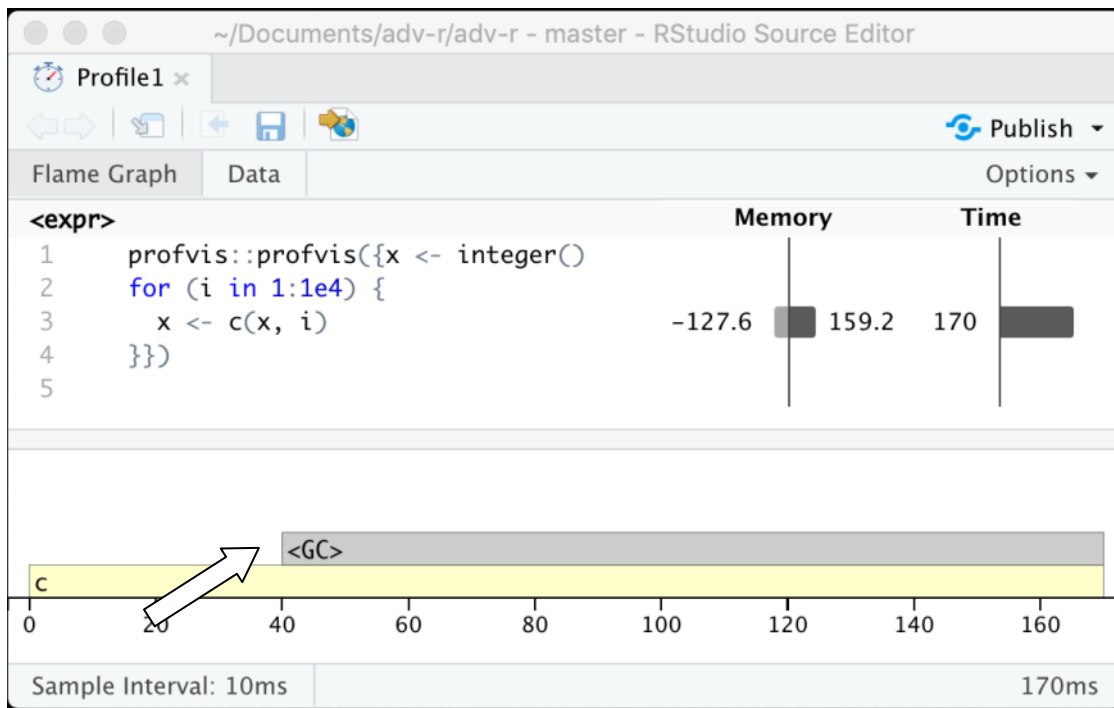


Во вкладке flame graph внизу Вы можете увидеть особый вид вызова <GC>, это Garbage Collector – сборщик мусора. Если <GC> отнимет много времени работы программы, то у Вас много копирования мало живущих объектов данных.

Создайте код.

```
x <- integer()
for (i in 1:1e4) {
  x <- c(x, i)
}
```

Выполните его профилизацию и Вы увидите, что большая часть времени ушла на сбор мусора.



Можно увидеть, что программы использовала еще также много памяти, которая то заполнялась (до 159.2), а потом освобождалась (127.6). Проблем в том, что использовался цикл `for`, который создавал каждый раз новый объект для `x`.

Упражнение

Выполните профилизацию данного кода

```
f <- function(n = 1e5) {
  x <- rep(1, n)
  rm(x)
}
```

4.2.3. Микротесты (Microbenchmarking)

Микротесты оценивают эффективность маленького кусочка кода, на работу которого от наносекунд (ns) до микро (µs) или миллисекунд (ms). Для этого можно использовать пакет (библиотеку) `bench`.

```
x <- runif(100)
(lb <- bench::mark(
  sqrt(x),
  x ^ 0.5
))
#> # A tibble: 2 x 6
#>   expression      min median `itr/sec` mem_alloc `gc/sec`
```

```
#>   <bch:expr> <bch:tm> <bch:tm>      <dbl> <bch:byt>      <dbl>
#> 1 sqrt(x)      865ns   1.05µs   679021.    848B         0
#> 2 x^0.5        3.78µs   4.17µs   203205.    848B         0
```

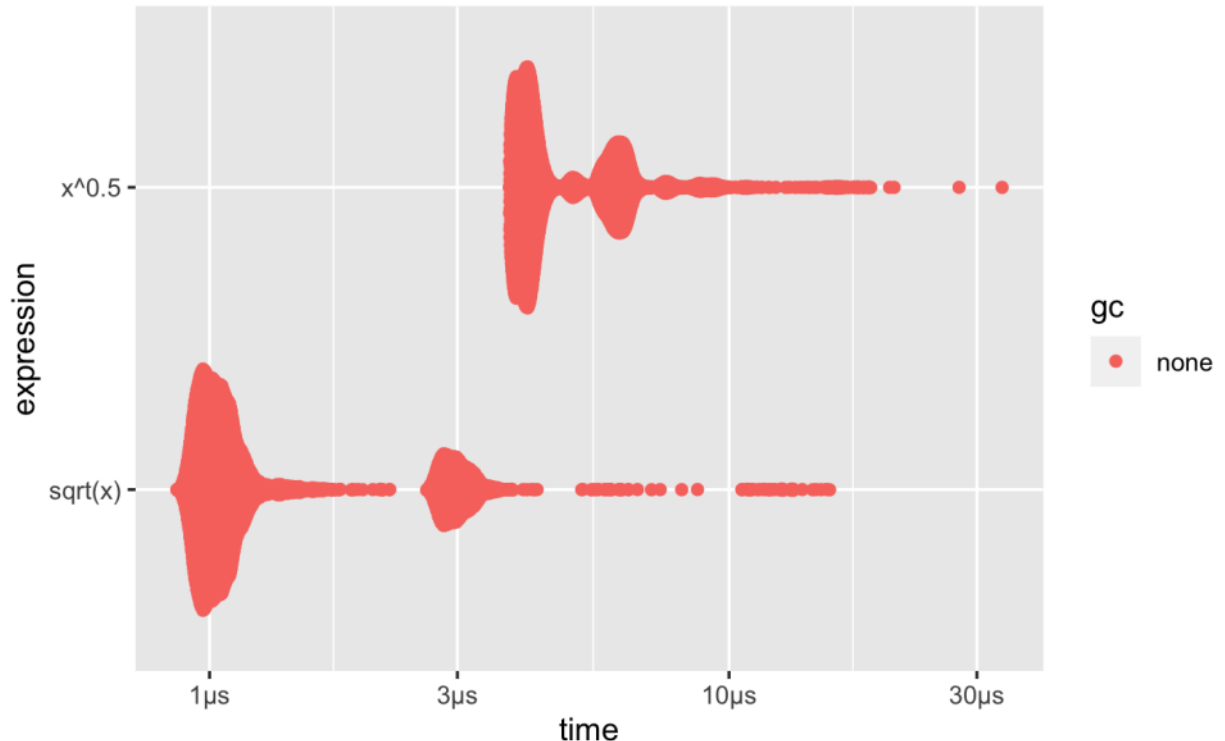
По умолчанию `bench::mark()` выполняет каждое выражение минимум 1 раз (`min_iterations = 1`), столько итераций сколько можно выполнить за 0.5 секунд (`min_time = 0.5`). Функция `mark()` проверяет, чтобы каждый запуск программы возвращал одно и то же значение. Если необходимо, сравнивать скорость работы при разных возвращаемых значениях установите `check = FALSE`.

Функция `bench::mark()` предоставляет результаты в виде фрейма данных формата `Tibble` с данными о минимальном, медианном времени выполнения, число итераций в секунду, данные об использовании памяти и сборщика мусора, а также другие значения.

Визуализируйте результирующие данные `bench::mark()` с помощью `plot()`.

```
plot(lb)
```

```
#> Loading required namespace: tidyrr
```



Можно увидеть мультимодальность распределения времени выполнения и смещенность вправо. Часто такой график связан с работой других программ на компьютере.

Упражнение

Проведите microbenchmarking для двух функций:

$x^{1/2}$

$\exp(\log(x)/2)$

5.1. Расчет базовых статистических характеристик для ряда данных

Задание

Рассчитать статистические характеристики ряда с помощью функций `mean()`, `median()`, `sd()`, `prop.table()`, `var()`, `IQR()`, `summary()` для колонки `Enroll` набора данных `College`.

5.2. Расчет базовых статистических характеристик для ряда данных

Задание

Тесты Стьюдента, Уилкоксона, Шапиро-Уилкса, ANOVA, хи-квадрат Пирсона, тесты бинарной классификации для колонки `Enroll` набора данных `College`.

6.1. Проведение МНК для парной линейной регрессии с помощью `lm()`. Анализ качества построенной модели.

Задание

Воспользуйтесь функцией `lm()` для подгонки простой линейной регрессии с `mpg` в качестве отклика и `horsepower` в качестве предиктора для набора данных `auto`. Примените функцию `summary()` для вывода результатов на экран. Прокомментируйте эту выведенную информацию.

- 1) Есть ли линейная связь между предиктором и откликом?
- 2) Насколько сильна связь между предиктором и откликом?
- 3) Каково направление связи между предиктором и откликом?
- 4) Чему равно модельное значение `mpg` при `horsepower = 98`? Каковы соответствующие 95%-ные доверительные интервалы для регрессионной прямой и для предсказанного значения?
- 5) Постройте график зависимости отклика от предиктора.

Используйте функцию `abline()` для добавления регрессионной линии, найденной по методу наименьших квадратов.

- 6) Воспользуйтесь функцией `plot()` для построения диагностических графиков для модели, подогнанной по методу наименьших квадратов. Прокомментируйте любые обнаруженные вами проблемы с этой моделью.

6.2. Расчет параметров модели множественной регрессии. Анализ качества построенной модели. Устранение мультиколлинеарности посредством `step()` и `vif()`.

Задание

Для смоделированных данных

```
> set.seed(1)
> x1 = runif(100)
> x2 = 0.5*x1 + rnorm(100)/10
> y = 2 + 2*x1 + 0.3*x2 + rnorm(100)
```

Постройте линейную модель, в которой y является функцией от x_1 и x_2 .

- 1) Выполните расчет параметров модели множественной регрессии.
- 2) Проведите анализ качества построенной модели.
- 3) Устраните мультиколлинеарность посредством `step()` и `vif()`.

7.1. Построение точечных диаграмм

Задание

1) Примените функцию `pairs ()` для построения матрицы диаграмм рассеяния по первым десяти числовым столбцам. Примените функцию `plot()` для 5 пар столбцов.

7.2. Построение столбчатых, круговых диаграмм, гистограмм и функций плотности

Задание

1) Воспользуйтесь функцией `hist()` и для 4 количественных переменных набора `College()` постройте гистограммы с разным числом классов. Полезной для вас может оказаться команда

```
par (mf row = c (2, 2))
```

она разделит окно графического устройства на четыре области, что сделает возможным построение четырех графиков одновременно. Изменение аргументов этой функции приведет к разделению экрана иным образом.

2) Постройте функцию плотности распределения для колонки `Enrol`. Разбейте университеты на классы по значению `Enrol`. Сделайте круговую диаграмму для этих классов

8.1. Визуализация многомерных данных для классификации. Кластерный анализ для заданной выборки

Задание

Воспользуйтесь набором данных USArrests. Выполните иерархическую кластеризацию штатов.

- 1) Выполните иерархическую кластеризацию штатов, используя полное присоединение и евклидово расстояние.
- 2) Выполните расщепление дендрограммы на высоте, которая приводит к выделению трех кластеров. Какие штаты входят в каждый из этих кластеров?
- 3) Выполните иерархическую кластеризацию штатов на основе полного присоединения и евклидова расстояния после масштабирования переменных, в результате которого их стандартные отклонения становятся равными 1.
- 4) Какой эффект на полученную иерархическую кластеризацию оказывает масштабирование переменных? На ваш взгляд, следует ли выполнять масштабирование переменных перед вычислением различий между наблюдениями? Обоснуйте свой ответ.

8.2. Применение метода опорных векторов и деревьев решений

Задание

Примените методы, основанные на опорных векторах, для предсказания уровня расхода топлива автомобилем (высокий или низкий) по данным из таблицы Auto.

- 1) Создайте бинарную переменную, принимающую значение 1 для автомобилей, у которых расход топлива превышает соответствующее медианное значение, и 0 для автомобилей, у которых расход топлива меньше этого медианного значения.
- 2) Постройте по этим данным классификатор на опорных векторах с разными значениями `cost` для предсказания уровня расхода топлива (низкий или высокий). Приведите частоты ошибок, полученные в ходе перекрестной проверки при разных значениях этого параметра. Прокомментируйте свои результаты.

3) Теперь повторите (2), используя SVM с радиальным и полиномиальным ядрами для нескольких значений параметров `gamma`, `degree` и `cost`. Прокомментируйте свои результаты.

4) Постройте графики, подтверждающие Ваши ответы для пунктов (2) и (3).

Подсказка: можно применить `plot()` со следующими параметрами:

```
> plot(svmfit, dat, x1 ~ x4)
```

в которой `svmfit` содержит построенную вами модель, а `dat` — это таблица с вашими данными, Вы можете выполнить для изображения только первой и четвертой переменных. Однако Вы должны заменить `x1` и `x4` на правильные имена переменных.