ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ
им. проф. М. А. БОНЧ-БРУЕВИЧА»
(СПбГУТ)**

Ф.В. Филиппов

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ R

ПРАКТИКУМ

СПбГУТ)))

**САНКТ-ПЕТЕРБУРГ
2017**

УДК 004.31(004.42)

Рецензент
кандидат технических наук, доцент кафедры робототехники
и автоматизации производственных систем СПбГЭТУ «ЛЭТИ»
А.В. Шевченко

Утверждено редакционно-издательским советом СПбГУТ
в качестве практикума

Филиппов, Ф.В.
Программирование на языке R: практикум / Ф. В. Филиппов; СПбГУТ, – СПб., 2017. – 77 с.

Рассматриваются практические аспекты языка R, используемые для обработки информации и анализа контента.

Пособие предназначено для бакалавров направления 09.03.02 Информационные системы и технологии и будет полезно при изучении дисциплин «Технология программирования», «Технологии обработки информации» и «Технологии проектирования программного обеспечения информационных систем».

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

Язык программирования *R* является эффективным и удобным средством для работы с данными, моделирования и визуализации. *R* развивается в рамках *open-source* проекта и доступен для различных платформ. В последние годы язык *R* заслуженно набирает популярность и широко используется в университетах мира в учебном процессе. *R* сочетает объектно-ориентированный и функциональный подходы к программированию. *R* – скриптовый язык, поэтому, в частности, в нём нет объявления переменных.

Основная мощь системы программирования на *R* заключена в дополнительных пакетах, которые устанавливаются отдельно (например, командой *install.packages*). В настоящее время репозитарий *CRAN* (*Comprehensive R Archive Network*) для языка программирования *R* включает более десяти тысяч доступных пакетов. Их количество растет, а назначение расширяется, по мере роста задач в различных научных областях. Полную информация о доступных пакетах всегда можно получить на [1].

Язык *R* постоянно совершенствуется - как в плане возможностей самого языка, так и среды разработки. Обновления, содержащие новые и усовершенствованные пакеты появляются несколько раз в неделю. Чтобы оставаться в курсе этих многочисленных изменений нужно периодически обращаться к интернет-ресурсам, которые информируют о том, что происходит в мире *R*. Основным является сайт [2], а на сайте [3] оперативно публикуется информация обо всех новых и обновленных пакетах, и содержатся ссылки на *CRAN* для каждого из них.

Удобной интегрированной средой разработки программ на языке *R* является *RStudio*. Инсталляторы интерпретатора языка *R* [2] и среды разработки *RStudio* [4] доступны для всех наиболее распространенных операционных систем, в частности *Windows Vista/7/8/10*, *Mac OS X 10.6+*, *Debian 8+/Ubuntu 12.04*, *Fedora 19+/RedHat 7+/openSUSE 13.1+*.

В пособии рассматриваются практические аспекты языка *R*, используемые для обработки информации и анализа контента. Основное внимание уделено подготовке исходных данных и повышению эффективности циклической обработки за счет использования специальных функций.

Для удобства, пособие содержит краткое описание базовых функций (раздел 5), которые удобно использовать при программировании. Функции разбиты по тематическим разделам для облегчения поиска. В последнем разделе приведены задания к упражнениям.

1. Среда разработки *RStudio*

Интерфейс интегрированной среды разработки *RStudio* состоит из четырех панелей (рис. 1).

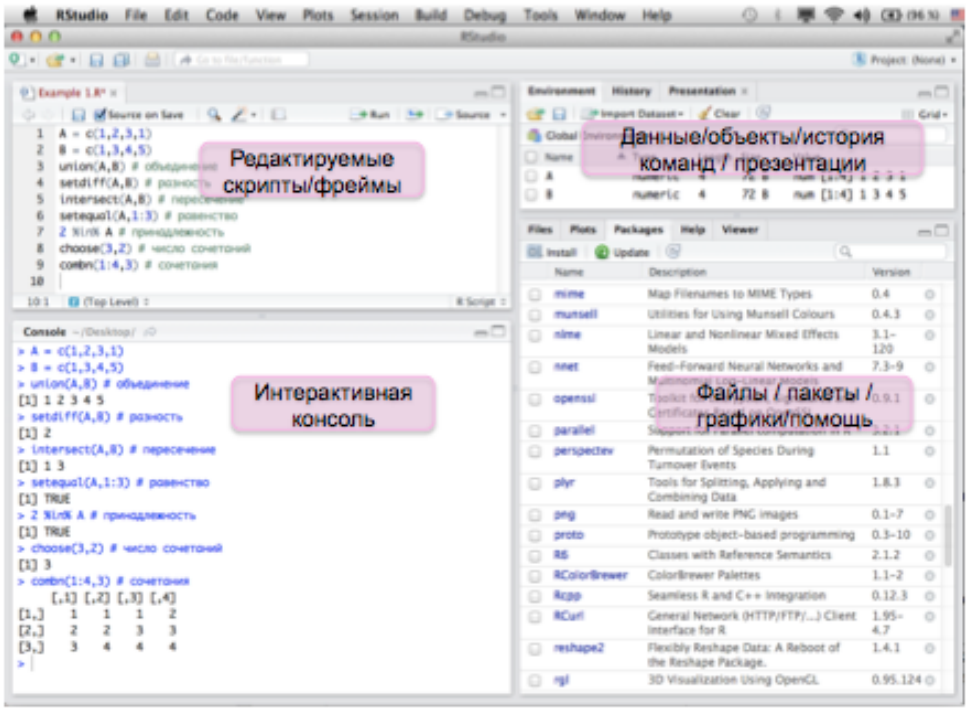


Рис.1. Панели среды разработки *RStudio*

Верхняя левая панель называется *Source* и служит для размещения кода скриптов, которые подлежат отладке и исполнению. Кроме того, там можно посмотреть содержимое некоторых объектов, например фреймов, имена которых в закладке *Environment* правой верхней панели снабжены специальными иконками.

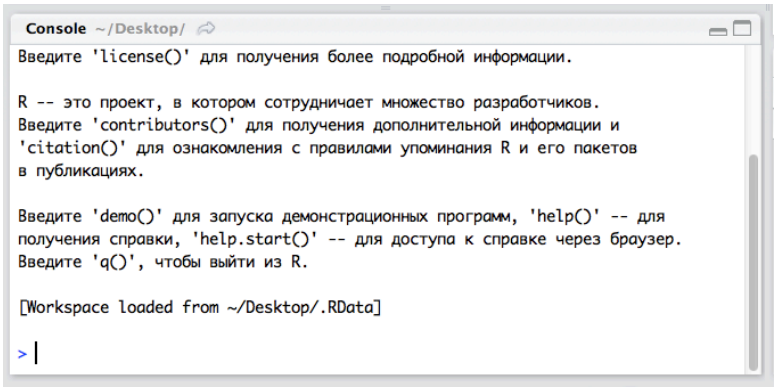


Рис.2. Содержимое консоли при запуске RStudio

Нижняя левая панель называется консоль. Каждый раз при запуске

RStudio на ней будет указана версия языка *R* и его краткая характеристика (рис. 2). Ниже находится приглашение (командная строка), где вводятся команды для *R* интерпретатора. Команды здесь можно вводить прямо с клавиатуры, либо запуская скрипты на верхней панели покомандно (*Run*) или целиком (*Source*).

Правая верхняя панель содержит три закладки: история команд (*History*), рабочее пространство (*Environment*) и презентация (*Presentation*). В закладке *Environment* содержится информация обо всех созданных в процессе работы объектах. В закладке *History*, можно посмотреть все команды, которые вводились и исполнялись в процессе работы. Закладка *Presentation* служит для визуализации и отладки слайдов презентаций, создаваемых средствами пакета *knitr*.

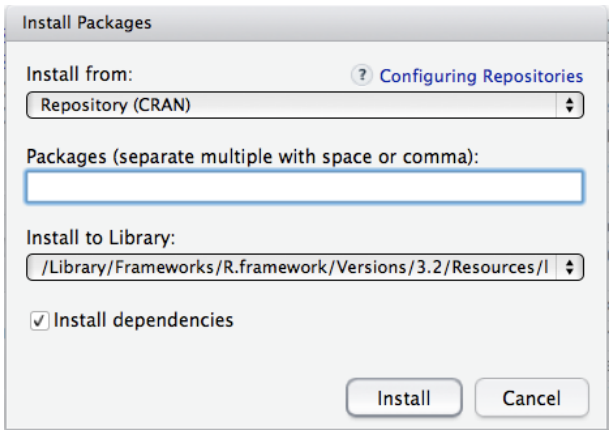


Рис.3. Меню для инсталляции пакетов

Правая нижняя панель содержит пять закладок: файлы (*Files*), графики (*Plots*), пакеты (*Packages*), помощь (*Help*) и обозреватель (*Viewer*). По закладке *Files* осуществляется доступ к файловой системе компьютера. Пространство закладки *Plots* используется для вывода графиков и изображений. По закладке *Packages* осуществляется доступ к любому пакету системной библиотеки *R*. Двойной щелчок по названию пакета осуществляет доступ к описанию всех его функций, а щелчок по иконке *Install* позволяет выбрать и инсталлировать из интернета любой отсутствующий пакет (рис. 3). Здесь можно задать источник (по умолчанию *CRAN*), наименование пакета (или нескольких пакетов) и директорий размещения. При установленном флаге *Install dependencies* наряду с указанным пакетом также будут инсталлированы пакеты, необходимые для работы последнего.

Закладка *Help* служит для вывода справочной информации, а закладка *Viewer* позволяет просматривать локальный веб-контент. Его содержимое может быть либо статическими файлами *HTML*, записанными во временный каталог, либо локально выполняемым веб-приложением.

2. Структуры данных

Как и все *Matlab*-подобное программное обеспечение, язык *R* предназначен для обработки различных таблиц. Существуют различные виды таблиц: векторы и списки (таблицы размерности 1), матрицы и фреймы (таблицы размерности 2) и массивы (таблицы произвольной размерности *n*).

Важной особенностью, отличающей различные структуры данных является тип их контента. Различают структуры, в которых все элементы одинакового типа (гомогенные) и структуры с элементами отличающегося типа (гетерогенные) [5].

Указанные выше особенности разделяют данные на пять типов базовых структур, которые наиболее часто используются в языке R для анализа данных (табл. 1).

Таблица 2.1

Структуры данных в языке R

Размерность	Гомогенные	Гетерогенные
1	*Vector*	*List*
2	*Matrix*	*Data Frame*
n	*Array*	

Вектор – это рабочая лошадка в языке *R* [6]. Трудно представить *R* код или просто интерактивную сессию, которые бы не использовали векторы.

Матрица в соответствии с математической концепцией является прямоугольной таблицей, и по сути – это вектор с двумя дополнительными атрибутами – числом строк и столбцов.

Вот простейший код, использующий матрицу и векторы:

```
> m=rbind(c(2,5),c(3,3))
> m
     [,1] [,2]
[1,]    2    5
[2,]    3    3
> m %*% c(1,1)
     [,1]
[1,]    7
[2,]    6
```

Во-первых, мы использовали функцию *rbind()* (от *row bind*) для построения матрицы из двух векторов и запоминания результата в *m*. Затем, мы ввели имя объекта *m* и нажав *Ente*r посмотрели получившуюся матрицу.

Наконец, мы умножили матрицу *m* на вектор (1,1). Для того, чтобы использовалось *матричное* умножение применен оператор %*%.

Список в языке *R* является аналогом структуры в языке *C*, то есть является контейнером для размещения данных различных типов. Общее применение списков состоит в том, чтобы упаковать возвращаемые значения сложных функций. Например, функция *lm()* (от *linear model*) выполняет регрессионный анализ, вычисляя не только расчетный коэффициент, но и остатки, статистику проверки гипотезы и так далее. Эти результаты упаковываются в список и таким образом формируется одно возвращаемое значение.

Типичный набор данных содержит данные различных типов, например, числа и строковые символы. Несмотря на то, что результат скажем, *n* наблюдений *k* переменных имеют "внешний вид" матрицы, он не является таковым в *R*. Вместо этого мы имеем фрейм данных *R*.

Фактически фрейм это список, в котором каждый компонент является вектором, соответствующим столбцу "матрицы" наблюдений. Проектировщики языка *R* сделали так, что многие операции с матрицами могут быть также применены к фреймам данных.

Перейдем к подробному изучению базовых структур данных.

2.1. Векторы

В языке *R* нет скалярных величин. Когда мы определяем значение простой переменной, например *s* = -1, то она интерпретируется как вектор единичной длины:

```
> s = -1; s
 [1] -1
```

однако это не мешает использовать s как константу "-1".

Элементы вектора

Вектор является фундаментальной структурой данных в *R*. Любой объект в действительности рассматривается как отдельный элемент вектора. Все элементы вектора должны быть одного типа: целыми (*integer*), вещественными (*double*), строковыми (*character*), булевскими (*logical*), комплексными (*complex*) или объектами (*object*). Тип числовой (*numeric*) объединяет два типа: *integer* и *double*. Индекс первой координаты (компоненты, элемента) вектора начинается с 1. Компоненты вектора в памяти хранятся последовательно, поэтому невозможно вставлять или удалять элементы вектора. При необходимости в подобных операциях следует вместо векторов использовать списки (*list*).

Компонентам вектора могут быть не присвоены значения – эта ситуация отображается как *NA* (от *not available*). В статистических наборах данных часто встречаются недостающие данные, соответствующие наблюдениям, для которых значения отсутствуют. Для многих статистических функций, мы можем указать функции не обрабатывать любые пропущенные значения.

Векторы имеют три общих свойства, которые можно узнать, используя соответствующую функцию:

- *typeof()* - тип, из каких типов элементов состоит;
- *length()* - длина, как много элементов содержит;
- *attributes()* - атрибуты, дополнительные произвольные метаданные.

Способы задания векторов

Существуют различные способы задания векторов.

Первый способ – создаем «нулевой» вектор определенного типа и длины, а затем заполняем его нужными значениями. Для этого можно использовать функцию *vector()*:

```
> vector('integer',5)
[1] 0 0 0 0 0
> vector('complex',7)
[1] 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i 0+0i
> vector('character',12)
 [1] "" "" "" "" "" "" "" "" "" "" "" ""
```

можно использовать функцию для задания конкретного типа:

```
> logical(10)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> double(6)
[1] 0 0 0 0 0 0
```

Второй способ задания вектора – непосредственно определить значения его элементов.

Можно просто задать интервал значений с помощью оператора двоеточие:

```
> 5:12
[1]  5  6  7  8  9 10 11 12
```

можно использовать функцию *c()* (от *combine*):

```
> c(3,5,7,9)
[1] 3 5 7 9
```

```
> c(1:5,"6")
[1] "1" "2" "3" "4" "5" "6"
```

или функцию *seq():*

```
> seq(1, 12, by = pi)
[1]  1.000000  4.141593  7.283185 10.424778
```

Можно также использовать функцию *rep()* при формировании часто повторяющихся значений данных:

```
> rep(2:6,3)
 [1] 2 3 4 5 6 2 3 4 5 6 2 3 4 5 6
```

Наконец, можно задать вектор, набирая данные на клавиатуре. Это реализуется с помощью функции *scan()*.

В каждом из этих случаев формируется вектор определенной длины из элементов одного типа.

Рассмотрим примеры выбора отдельных элементов вектора на примере вектора *x*:

```
> x = seq(-1,1,by=0.2); x
 [1] -1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8  1.0
> x[4:7] # выбираем с 4 по 7 элемент вектора x
[1] -0.4 -0.2  0.0  0.2
> x[c(5,9:11)] # выбираем 5 и с 9 по 11 элемент вектора x
[1] -0.2  0.6  0.8  1.0
> x[-(2:10)] # удаляем элементы со 2 по 10
[1] -1  1
> x>0
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x[x>0]
[1] 0.2 0.4 0.6 0.8 1.0
```

При необходимости можно присвоить имя каждой координате вектора. Так, элементы вектора *x* не имеют имен:

```
> names(x)
NULL
```

присвоим им имена

```
> names(x) =letters[1:length(x)]; x
   a    b    c    d    e    f    g    h    i    j    k
```

```
-1.0 -0.8 -0.6 -0.4 -0.2  0.0  0.2  0.4  0.6  0.8  1.0
> x["i"]
  i
0.6
```

Для формирования имен координат здесь использовался встроенный вектор *letters* содержащий 26 строчных букв латинского алфавита (вектор *LETTERS* содержит 26 прописных букв – проверьте!).

Конечно, формировать любые имена координатам вектора можно при его определении:

```
> c(a=3,b=12,delta=-7)
   a    b delta
   3   12    -7
```

При таком способе формирования имен важно использовать значок =, так как использование <- приведет к тому, что значение компоненте вектора присвоено будет, а имя нет:

```
> c(a<-3,b=12,delta<-7)
   b
 3 12 -7
```

Для формирования имен координатам существующего вектора можно использовать функцию *c()*:

```
> x = c(-3,12,7)
> names(x) = c("a","b","delta")
> x
   a    b delta
  -3   12    7
```

Для любого вектора можно определить его тип с помощью функции *typeof()* или проверить принадлежность к конкретному типу с помощью функций: *is.character(), is.double(), is.integer(), is.logical()* или *is.complex()*.

При попытке объединить элементы разных типов в одном векторе произойдет принудительное преобразование к одному наиболее гибкому типу. Порядок представления типов от наименее к наиболее гибкому: *logical, integer, double* и *character*. Например, комбинация *character* и *integer* даст *character*:

```
> str(c("a",1,"2"))
 chr [1:3] "a" "1" "2"
```

Когда логический вектор принудительно преобразовывается в целочисленный или *double, FALSE* заменяется на 0, а *TRUE* на 1.

2.2. Матрицы и массивы

Матрица – это вектор с двумя дополнительными атрибутами, числом строк и числом столбцов.

Многоразмерные векторы в *R* называются массивами (*array*). Двухразмерные массивы называются матрицами, над которыми выполняются обычные матричные математические операции.

Индексы строк и столбцов матрицы начинаются с 1, так что, например, верхний левый угол матрицы *a* обозначается *a[1,1]*. Внутреннее линейное хранение матрицы в памяти осуществляется по столбцам, сначала хранятся все элементы столбца 1, затем столбца 2 и т.д.

Одним из способов создания матрицы - использование матричной функции *matrix(),* например:

```
> y <- matrix(c(1,-3,7,4),nrow=2,ncol=2) ; y
     [,1] [,2]
[1,]  1    7
[2,] -3    4
```

Хотя внутреннее хранение матрицы осуществляется по столбцам, мы можем использовать в функции *matix()* аргумент *byrow = TRUE*, для того, чтобы указать, что данные, которые мы используем, чтобы заполнить матрицу интерпретировались как заданные по строкам, например:

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

В качестве заголовков строк и столбцов создаваемой матрицы автоматически выводятся соответствующие индексные номера (строки: [1,], [2,], и т.д.; столбцы: [,1], [,2], и т.д.). Для придания пользовательских заголовков строкам и столбцам матриц используют функции *rownames()* и *colnames()* соответственно. Например, для обозначения строк матрицы *m* именами *First* и *Second* необходимо выполнить команду *rownames(m) = c("First", "Second").*

Матрицу можно собрать также из нескольких векторов, используя функции *cbind()* (от *colum* и *bind – столбец* и *связывать)* или *rbind()* (от *row* и *bind – строка* и *связывать)*:

```
> x <- c(1, 2, 3, 4)
> y <- c(5, 6, 7, 8)
> z <- c(9, 10, 11, 12)
> cbind(x,y,z)
     x y  z
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
> rbind(x,z,y)
  [,1] [,2] [,3] [,4]
x    1    2    3    4
z    9   10   11   12
y    5    6    7    8
```

Практически все векторные операции одинаково применимы в отношении матриц и массивов. Так, путем индексирования мы можем извлекать из матриц необходимые элементы и далее подвергать их требуемым преобразованиям. Рассмотрим примеры выбора отдельных элементов на примере матрицы *A*:

```
> A = matrix(c(-5:6),nrow=4); A
     [,1] [,2] [,3]
[1,]   -5   -1    3
[2,]   -4    0    4
[3,]   -3    1    5
[4,]   -2    2    6
> A[2,] # вторая строка
[1] -4  0  4
> A[,3] # третий столбец
[1] 3 4 5 6
> A[2,3] # элемент матрицы
[1] 4
> A[,2]*A[,3] # поэлементное произведение 2 и 3 столбца
[1] -3  0  5 12
> t(A) # транспонированная матрица A
     [,1] [,2] [,3] [,4]
[1,]   -5   -4   -3   -2
[2,]   -1    0    1    2
[3,]    3    4    5    6
```

Последний пример показывает получение транспонированой матрицы с помощью функции *t()*.

Для формирования массива можно воспользоваться функцией *array(),* указав в качестве аргументов вектор значений и вектор размерностей *dim*.

Кроме того, можно задать имена для каждой размерности *dimnames*, например:

```
> array_3D <- array(1:24,
  dim = c(4, 3, 2),
  dimnames = list(c("x1", "x2", "x3", "x4"), c("y1", "y2", "y3"), c("z1", "z2") ) )
> array_3D
  , , z1
     y1  y2  y3
x1   1    5    9
x2   2    6  10
x3   3    7  11
x4   4    8  12
, , z2
      y1 y2 y3
x1  13 17 21
x2  14 18 22
x3  15 19 23
x4  16 20 24
```

Доступ к отдельным элементам массива осуществляется с помощью использования их индексов:

```
> array_3D[2,3,1]
[1] 10
> array_3D[2,3,2]
[1] 22
> array_3D[1:2,3,2]
x1 x2
21 22
> array_3D[1:2,2:3,2]
   y2 y3
x1 17 21
x2 18 22
```

2.3. Списки

Списки отличаются от векторов тем, что их элементы могут быть разного типа. Это позволяет эффективно, т.е. в одном объекте, хранить разнородную информацию. Каждый компонент списка может являться переменной, вектором, матрицей, фактором или другим списком.

Списки можно создавать с помощью функции *list()*:

```
> z = list("abc", 5:7, c(-5.25,6.54), c(FALSE, TRUE))
> str(z)
vList of 4
 $ : chr "abc"
```

```
 $ : int [1:3] 5 6 7
 $ : num [1:2] -5.25 6.54
 $ : logi [1:2] FALSE TRUE
```

С помощью функции *str()* мы раскрыли структуру созданного списка *z* и показали типы его элементов.

Рассмотрим особенности доступа к элементам списка на следующем примере. Сформируем список *our.list* из трех разнотипных векторов:

```
> v1 = c("Alpha","Beta","Gamma")    #вектор 1
> v2 = seq(-2, 1.75, 0.75)                  #вектор 2
> v3 = c(FALSE, FALSE, TRUE) )         #вектор 3
> our.list = list(Names=v1, Number=v2, Logic=v3); our.list)
$Names
[1] "Alpha" "Beta"  "Gamma"
$Number
[1] -2.00 -1.25 -0.50  0.25  1.00  1.75
$Logic
[1] FALSE FALSE  TRUE
```

К элементам списка можно получить доступ посредством трех различных операций индексации. Для обращения к поименованным компонентам применяют знак *$*, например:

```
> our.list$Names
[1] "Alpha" "Beta"  "Gamma"
```

Имеется возможность извлекать из списка не только его поименованные компоненты-векторы, но и отдельные элементы, входящие в эти векторы. Для этого необходимо воспользоваться индексацией при помощи квадратных скобок. Единственная особенность работы со списками здесь состоит в том, что сначала необходимо указать имя компонента списка, используя знак *$*, а уже затем номера отдельных элементов этого компонента:

```
> our.list$Names
[1] "Alpha" "Beta"  "Gamma"
> our.list$Names[2]
[1] "Beta"
> our.list$Number[2:4]
[1] -1.25 -0.50  0.25
```

Извлечение компонентов списка можно осуществлять также с использованием двойных квадратных скобок, в которые заключается номер компонента списка:

```
> our.list[[2]]
[1] -2.00 -1.25 -0.50  0.25  1.00  1.75
> our.list[[2]][2:4]
[1] -1.25 -0.50  0.25
```

Списки иногда называют рекурсивными векторами, поскольку список может содержать другие списки. Это коренным образом отличает их от векторов.

```
> y <- list(list(list()))
> str(y)
List of 1
 $ :List of 1
  ..$ : list()
```

Существует функция *is.recursive()*, которая позволяет проанализировать объект на предмет наличия рекурсии.

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

Списки обычно используются как контейнеры для размещения данных различных типов.

2.4. Фреймы (таблицы)

Фрейм данных (*data frame)* представляет собой объект *R*, по структуре напоминающий лист электронной таблицы. Каждый столбец таблицы является вектором, содержащим данные определенного типа. При этом действует правило, согласно которому все столбцы должны иметь одинаковую длину.

Таблицы (фреймы) – это основной класс объектов *R*, используемых для хранения данных, и обычно они экпортируются извне. Для формирования таблицы средствами языка *R* используют функцию *data.frame()*. Например, создадим фрейм из четырех векторов с данными о метрополитенах трех крупнейших городов России:

```
> v1 = c("Moscow", "St.Petersburg", "N.Novgorod")
> v2 = c(206, 67, 14)
> v3 = c(1935, 1955, 1985)
> v4 = c(2384.5, 741.79, 37.24)
> Metro = data.frame(City = v1, Stations = v2, Year = v3, Passengers = v4)
> Metro
             City Stations   Year Passengers
1          Moscow       206  1935     2384.50
```

```
2  St.Petersburg         67  1955       741.79
3    N.Novgorod          14  1985        37.24
```

Извлечь отдельные компоненты таблиц для выполнения необходимых вычислений, как и в примерах со списками, можно с использованием знака $, двойных квадратных скобок [[]], квадратных скобок с указанием двух индексов [<номер_строки>, номер_столбца>], либо непосредственно по имени столбца:

```
> Metro$Year
[1] 1935 1955 1985
> Metro[[3]]
[1] 1935 1955 1985
> Metro[,3]
[1] 1935 1955 1985
> Metro["Year"]
  Year
1 1935
2 1955
3 1985
```

После имени или индексного номера столбца можно указывать индексные номера отдельных ячеек таблицы, что позволяет извлекать содержимое этих ячеек:

```
> Metro$Passengers[2:3] # 2-й и 3-й элемент из столбца Passengers
[1] 741.79  37.24
> Metro$Year[Metro$Year > 1940]
[1] 1955 1985
> Metro[2,4]
[1] 741.79
```

3. Подготовка данных

Возможности системы *R* по вводу и редактированию данных весьма многообразны. Импортирование данных в систему *R* часто вызывает проблемы у тех, кто только начинает работать с *RStudio*. Тем не менее, ничего сложного в этом нет. Ниже будут подробно рассмотрены наиболее распространенные способы импорта таблиц данных в рабочую среду *R*, однако сначала стоит ознакомиться с правилами подготовки загружаемых файлов:

- В импортируемой таблице с данными не должно быть пустых ячеек. Если некоторые значения по тем или иным причинам отсутствуют, вместо них следует ввести *NA*.
- Импортируемую таблицу с данными рекомендуется преобразовать в простой текстовый файл с одним из допустимых расширений. На практике обычно используются файлы с расширением *.txt*, в которых значения переменных разделены знаками табуляции (*tab-delimited files*), а также файлы с расширением *.csv* (от *comma separated values*), в которых значения переменных разделены запятыми.
- В качестве первой строки в импортируемой таблице рекомендуется ввести заголовки столбцов-переменных. Такая строка – удобный, но не обязательный элемент загружаемого файла. Если она отсутствует, то об этом необходимо сообщить в описании команды, которая будет управлять загрузкой файла (например, *read.table()* – см. ниже). Все последующие строки файла в качестве первого элемента содержат заголовки строк (если таковые предусмотрены), после которых следуют значения каждой из имеющихся в таблице переменных.
- В именах столбцов таблицы не допускается наличие пробелов. Кроме того, имена столбцов (так же как и имена строк) не должны начинаться с точки или чисел. Во избежание связанных с кодировкой проблем все текстовые величины в импортируемых файлах рекомендуется создавать с использованием букв латинского алфавита.
- Подлежащий импортированию файл рекомендуется поместить в рабочую папку программы, т.е. папку, в которой интерпретатор *R* по умолчанию будет "пытаться найти" этот файл. Чтобы выяснить путь к рабочей папке *R* используйте команду *getwd()* (от *get working directory*), например:

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

Изменить рабочий директорий можно при помощи команды setwd() (от *set working directory*):

```
> setwd("C:/MyDocuments/")
# при выполнении приведенной команды внешне ничего не произойдет,
# однако последующее применение команды getwd() покажет,
# что путь к рабочей папке изменился:
> getwd()
[1] "C:/My Documents/"
```

Ниже приведен фрагмент типичной таблицы данных (табл. 3.1), которая может быть успешно загружена для анализа в среду *R*. Используйте этот фрагмент в качестве образца при оформлении своих таблиц с данными.

Таблица 3.1

Типичная таблица данных

	Group	*Variable1*	*Variable2*	*Variable3*
Ivan	A	102	1.3	14
Vitaliy	A	98	1.4	11
Sergey	B	45	NA	8
Mikhail	B	50	3.2	6

Как видим, приведенный фрагмент имеет размерность 5x5, т.е. состоит из пяти строк и пяти столбцов. В первой строке представлены заголовки всех имеющихся в таблице столбцов, за исключением первого. Первый столбец, хотя и не имеет собственного заголовка, не является пустым – он содержит имена добровольцев, участвовавших в некотором эксперименте (*Ivan, Vitaliy*, и т.д.). Второй столбец имеет заголовок *Group* и содержит метки, по которым можно выяснить принадлежность испытуемых к той или иной экспериментальной группе (*A, B*, и т.д.). В терминах языка *R* переменная *Group* называется фактором. В последующих столбцах (с заголовками *Variable1,Variable2,* и т.д.) содержатся значения измеренных в ходе исследования переменных. В приведенном фрагменте таблицы имеется одно отсутствующее значение, вместо которого введено *NA*.

Пожалуй, одним из наиболее доступных и удобных средств подготовки данных для их последующего анализа при помощи *R*, является программа *Microsoft Excel*. Для сохранения *Excel*-таблиц в виде *txt*- или *csv*-файлов используйте опцию *Сохранить как* (*Save as*) в разделе *Файл* (*File*) главного меню этой программы.

3.1. Функция *read.table()*

Основной функцией для импортирования данных в рабочую среду *R* является *read.table()*. Эта мощная функция позволяет достаточно тонко настроить процесс загрузки внешних файлов, в связи с чем она имеет

большое количество управляющих аргументов. Наиболее важные из этих аргументов перечислены ниже в таблице (подробнее см. файл помощи, доступный в *RStudio* по команде *?read.table*).

Таблица 3.2

Назначение аргументов функции *read.table()*

Аргумент	Назначение
file	Служит для указания пути к импортируемому файлу. В качестве имени можно также указывать полную *URL*-ссылку на файл, который предполагается загрузить из интернета. Начиная с версии *R 2.10*, появилась возможность импортировать архивированные файлы в zip-формате.
header	Служит для сообщения программе о наличии в загружаемом файле строки с заголовками столбцов. По умолчанию принимает значение *FALSE*. Если строка с заголовками столбцов имеется, этому аргументу следует присвоить значение *TRUE*.
row.names	Служит для указания номера столбца, в котором содержатся имена строк (например, в рассмотренном выше примере это был первый столбец, поэтому *row.names = 1*). Важно помнить, что все имена строк должны быть уникальными, т.е. одинаковые имена для двух или более строк не допускаются.
sep	Служит для указания используемого в файле разделителя значений переменных (*separator* – разделитель). По умолчанию предполагается, что значения переменных разделены "пустым пространством", например, в виде пробела или знака табуляции (*sep = ""*). В файлах формата csv значения переменных разделены запятыми, и поэтому для них *sep = ","*.
dec	Служит для указания знака, используемого в файле для отделения целой части числа от дроби. По умолчанию *sep = "."*. Однако во многих странах в качестве десятичного знака применяют запятую, о чем важно вспомнить перед загрузкой файла и, при необходимости, использовать *dec = ","*.
nrows	Выражается целым числом, указывающим количество строк, которое должно быть считано из загружаемой таблицы. Отрицательные и иные значения игнорируются. Пример: *nrows = 100*.
skip	Выражается целым числом, указывающим количество строк в файле, которое должно быть пропущено перед началом импортирования. Пример: *skip = 5*

Для загрузки тщательно подготовленных файлов (см. правила выше) достаточно использовать минимальный набор аргументов функции *read.table()*. В качестве примера предположим, что нам необходимо загрузить файл *new_data.txt*, который хранится в рабочей папке *R*. Загружаемую таблицу данных мы намерены сохранить в виде объекта с именем *data_1*. Функция *read.table()* в этом случае может быть применена следующим образом:

```
> data_1 = read.table(file = "new_data.txt", header = TRUE)
```

Как отмечено выше, часто импортируемые в *R* файлы имеют формат *csv*. Для их загрузки можно воспользоваться той же функцией *read.table(),*

но при этом следует указать, что в качестве разделителя значений переменных в файле используется запятая:

```
> data_1 = read.table(file = "new_data.csv", header = TRUE, sep = ",")
```

3.2. Функция *read.csv()*

Аналогом функции *read.table()* для считывания *csv*-файлов является функция *read.csv()*:

```
> data_1 = read.csv(file = "new_data.csv", header = TRUE)
```

Если подлежащий загрузке файл хранится в папке, отличной от рабочей папки *R*, то следует указать полный путь к нему. При этом пользователям операционных систем *Windows* необходимо помнить, что для указания полных путей к файлам в программе *R* используется не обратный одинарный слэш (\), а прямой одинарный (/) либо двойной обратный слеш (\\). Например, следующие две команды будут успешно восприняты *R* и приведут к идентичному результату – загрузке файла *new_data.txt* и сохранению его в виде объекта *data_1*:

```
> data_1= read.csv(file = "D:\\Documents\\data_1.txt", header = TRUE)
> data_1 = read.csv(file = "D:/Documents/data_1.txt", header = TRUE)
```

Для интерактивного выбора загружаемого файла, который хранится вне рабочей папки *R*, можно применить вспомогательную функцию *file.choose()* (*выбрать файл*). Выполнение этой команды приводит к открытию обычного диалогового окна операционной системы *Windows*, в котором пользователь выбирает папку с необходимым файлом. Очень удобно совмещать *file.choose()* с командами *read.table()* или *read.csv()*, например:

```
> data_1 = read.table(file = file.choose(), header = TRUE, sep = ",")
```

3.3. Предварительный анализ

Всякий раз, когда мы начинаем работать с новым набором данных, первое, что необходимо сделать, это изучить его: формат данных, размеры, имена и способ хранения переменных, есть ли недостающие данные, существуют ли какие-либо недостатки в данных?

Ответим на эти вопросы, используя встроенные функции языка *R*. Возьмем массив из базы данных департамента сельского хозяйства США [7]. Сохраним данные, приведенные в таблице [8] в переменной plants.

Начнем с проверки класса переменной *plants*, что даст нам ключ к пониманию общей структуры данных:

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

Это очень общая характеристика данных, просто говорящая о том, что они сохранены в виде фрейма. Этот класс в языке *R* по умолчанию приписывается объектам, полученным путем чтения данных с использованием функций *read.csv()* и *read.table()*.

Так как набор данных сохраняется во фрейме, мы знаем, что эта таблица имеет два измерения (строки и столбцы) и можно использовать функцию *dim(),* чтобы точно узнать, сколько в наборе данных *plants* строк и столбцов:

```
> dim(plants)
[1] 5166   12
```

Можно узнать количество строк и столбцов, используя функции *nrow()* и *ncol(),* но при этом результат не изменится:

```
> nrow(plants)
[1] 5166
> ncol(plants)
[1] 12
> ncol(plants)==dim(plants)[2]
[1] TRUE
> nrow(plants)==dim(plants)[1]
[1] TRUE
```

Если нам необходимо знать, как много пространства набор данных занимает в памяти, можно использовать функцию *object.size()*:

```
> object.size(plants)
975808 bytes
```

Теперь, когда мы знаем форму и размеры набора данных, узнаем имена переменных внутри таблицы с помощью функции *names()*, которая возвратит названия столбцов:

```
> names(plants)
 [1] "Accepted.Symbol"          "Synonym.Symbol"
 [3] "Scientific.Name"          "Duration"
 [5] "Active.Growth.Period"             "Foliage.Color"
```

```
 [7] "pH..Minimum."                "pH..Maximum."
 [9] "Precipitation..Minimum."     "Precipitation..Maximum."
[11] "Shade.Tolerance"             "Temperature..Minimum...F."
```

Здесь подобраны достаточно содержательные имена переменных для этого набора данных, но это не всегда будет так. Следующим логичным шагом будет взглянуть на фактические данные. Однако, наш набор содержит более 5000 наблюдений (строк), так что весьма непрактично, оценивать все это сразу.

Функция *head()* позволяет просмотреть небольшую часть набора данных:

```
> head(plants)
  Accepted.Symbol Synonym.Symbol                Scientific.Name
1            ABELM            NA                     Abelmoschus
2             ABES            NA          Abelmoschus esculentus
3             ABIES           NA                           Abies
4             ABBA            NA                    Abies balsamea
5            ABBAB            NA  bies balsamea var. balsamea
6             ABUTI           NA                        Abutilon
            Duration  Active.Growth.Period Foliage.Color pH..Minimum.
1                                                               NA
2 Annual, Perennial                                             NA
3                                                               NA
4          Perennial  Spring and Summer          Green          4
5          Perennial                                            NA
6                                                               NA
  pH..Maximum. Precipitation..Minimum. Precipitation..Maximum.
1           NA                     NA                      NA
2           NA                     NA                      NA
3           NA                     NA                      NA
4            6                     13                      60
5           NA                     NA                      NA
6           NA                     NA                      NA
  Shade.Tolerance Temperature..Minimum...F.
1                                        NA
2                                        NA
3                                        NA
4        Tolerant                       -43
5                                        NA
6                                        NA
```

Проанализируем полученную распечатку. Каждая строка помечена номером наблюдения, а каждый столбец именем переменной. Поскольку экран монитора не достаточно широк, чтобы просмотреть все 10 столбцов

подряд, *R* отображает столько столбцов, сколько сможет разместить в один ряд, прежде чем продолжить на следующем.

По умолчанию, функция *head()* показывает первые шесть строк данных. Можно изменить это поведение, передав в качестве второго аргумента число строк, которые необходимо просмотреть, например *head(plants,10).* То же самое относится и к функции *tail(),* чтобы просмотреть 10 последних строк набора данных можно использовать вызов *tail(plants,10).*

После предварительного просмотра данных в верхней и нижней части, можно заметить много *NA*, которые являются заполнителями для отсутствующих значений. Для того, чтобы получить лучшее представление о том, как распределяются значения каждой переменной и сколько данных отсутствует можно использовать функцию *summary()*:

```
> summary(plants)
 Accepted.Symbol      Synonym.Symbol                  Scientific.Name
 ABBA    :  1         Mode:logical    Abelmoschus                   :  1
 ABBAB   :  1         NA's:5166       Abelmoschus esculentus        :  1
 ABELM   :  1                         Abies                         :  1
 ABES    :  1                         Abies balsamea                :  1
 ABIES   :  1                         Abies balsamea var. balsamea:  1
 ABTH    :  1                         Abutilon                      :  1
  (Other) :5160                       (Other)                       :5160
                Duration          Active.Growth.Period          Foliage.Color
 Perennial          :3031                      :4334                      :4334
                    :1030   Spring and Summer  : 447    Dark Green     :  82
 Annual             : 682   Spring             : 144    Gray-Green     :  25
 Annual, Perennial  : 179   Spring, Summer, Fall :  95  Green          : 692
 Annual, Biennial   :  95   Summer             :  92    Red            :   4
 Biennial           :  57   Summer and Fall    :  24    White-Gray     :   9
 (Other)            :  92   (Other)            :  30    Yellow-Green   :  20
      pH..Minimum.        pH..Maximum.      Precipitation..Minimum.
 Min.   :3.000       Min.   : 5.100      Min.   : 4.00
 1st Qu.:4.500       1st Qu.: 7.000      1st Qu.:16.75
 Median :5.000       Median : 7.300       Median :28.00
 Mean   :4.997       Mean   : 7.344      Mean   :25.57
 3rd Qu.:5.500       3rd Qu.: 7.800      3rd Qu.:32.00
 Max.   :7.000       Max.   :10.000      Max.   :60.00
 NA's   :4327        NA's   :4327        NA's   :4338
 Precipitation..Maximum.  Shade.Tolerance          Temperature..Minimum...F.
 Min.   : 16.00                    :4329      Min.   :-79.00
 1st Qu.: 55.00      Intermediate  : 242       1st Qu.:-38.00
 Median : 60.00      Intolerant    : 349       Median :-33.00
 Mean   : 58.73      Tolerant      : 246       Mean   :-22.53
 3rd Qu.: 60.00                                3rd Qu.:-18.00
 Max.   :200.00                                Max.   : 52.00
 NA's   :4338                                  NA's   :4328
```

Функция *summary()* предоставляет различную информацию для каждой переменной, в зависимости от ее класса. Для числовых данных она отображает минимум, 1-й квартиль, медиану, среднее, 3-й квартиль и максимум. Эти значения помогают нам понять, как распределяются данные.

Для категориальных переменных (факторов в *R*), функция *summary()* отображает сколько раз каждое значение встречается в данных. Например, каждое значение *Scientific_Name* появляется только один раз, так как оно является уникальным названием для конкретного растения. В противоположность этому функция *summary()*, для *Duration* (также фактор-переменная) показывает, что набор данных *plants* содержит 3031 многолетнее растение, 682 однолетних растений и т.д.

Можно заметить, что интерпретатор *R* усекает некоторые резюме, например для *Active_Growth_Period*, включая категорию под названием *'Other'*. Так как это "фактор" переменные можно узнать, сколько раз каждое значение на самом деле появляется в данных с помощью функции *table()* с указанием нужной переменной. Например, для переменной *Active_Growth_Period*:

```
> table(plants$Active.Growth.Period)

                                Fall, Winter and Spring                                          Spring
                                                   4334                                              15                                             144
                       Spring and Fall                                 Spring and Summer                                  Spring, Summer, Fall
                                                     10                                             447                                              95
                                Summer                                  Summer and Fall                                      Year Round
                                                     92                                              24                                               5
```

Каждая из рассмотренных функций предоставляет определенную информацию для того, чтобы помочь лучше понять структуру данных. Тем не менее, возможно наиболее полезной для понимания структуры будет функция *str(),* которая в сжатой форме даст характеристику набора данных:

```
> str(plants)
'data.frame':    5166 obs. of  12 variables:
 $ Accepted.Symbol        : Factor w/ 5166 levels "ABBA","ABBAB",..: 3 4 5 1 2 7 6 8 15 16 ...
 $ Synonym.Symbol         : logi  NA NA NA NA NA NA ...
 $ Scientific.Name        : Factor w/ 5166 levels "Abelmoschus",..: 1 2 3 4 5 6 7 8 9 10 ...
 $ Duration               : Factor w/ 9 levels "","Annual","Annual, Biennial",..: 1 5 1 8 8 1 2 1 8 8 ...
 $ Active.Growth.Period   : Factor w/ 9 levels "","Fall, Winter and Spring",..: 1 1 1 5 1 1 1 1 5 1
...
 $ Foliage.Color          : Factor w/ 7 levels "","Dark Green",..: 1 1 1 4 1 1 1 1 4 1 ...
 $ pH..Minimum.           : num  NA NA NA 4 NA NA NA NA 7 NA ...
 $ pH..Maximum.           : num  NA NA NA 6 NA NA NA NA 8.5 NA ...
 $ Precipitation..Minimum.  : int  NA NA NA 13 NA NA NA NA 4 NA ...
 $ Precipitation..Maximum.  : int  NA NA NA 60 NA NA NA NA 20 NA ...
```

```
$ Shade.Tolerance        : Factor w/ 4 levels "","Intermediate",..: 1 1 1 4 1 1 1 1 3 1 ...
$ Temperature..Minimum...F.: int  NA NA NA -43 NA NA NA NA -13 NA ...
```

Функция *str()* привлекательна тем, что она сочетает в себе многие черты других функций, которые мы рассмотрели, причем все в сжатом и удобном для чтения формате. Вначале она говорит нам о том, что набор данных представлен в виде фрейма *'data.frame'* и что он включает 5166 наблюдений и 12 переменных. Затем она дает нам имя и класс каждой переменной, а также предварительный просмотр ее содержимого.

Функция *str()* на самом деле очень общая функция, которую можно использовать для большинства объектов (набор данных, функция и т.д.) в языке *R*, чтобы понять его структуру.

3.4. Неопределенные значения

Недостающие значения играют важную роль в области анализа данных. Часто, пропущенные значения не должны быть проигнорированы, а наоборот тщательно изучены, чтобы увидеть как их можно восстановить. В языке *R*, обозначение *NA* используется для представления любого значения, которое "не доступно" (*not available*).

Любая операция со значением *NA* обычно дает в качестве результата *NA*. Чтобы проиллюстрировать это, давайте создадим вектор с *(44, NA, 5, NA)*, присвоим его переменной *x* и умножим на 3:

```
> x = c(44,NA,5,NA)
> x*3
[1] 132  NA  15  NA
```

Обратите внимание на то, что элементы результирующего вектора, которые соответствуют значениям *NA* в *x*, также имеют значения *NA*.

Создадим вектор *y*, содержащий 1000 выборок из стандартного нормального распределения и вектор *z*, содержащий 1000 значений *NA*:

```
> y = rnorm(1000)
> z = rep(NA,1000)
```

Теперь, выберем случайным образом 100 элементов из векторов *y* и *z*, сформировав тем самым вектор *my_data*:

```
my_data = sample(c(y,z),100)
```

Сначала выясним, где *NA* расположены в наших данных *my_data*. Функция *is.na()* говорит нам о том, какой элемент вектора является *NA*. Используем ее для вектора *my_data* и результат запишем в *my_NA*:

```
> my_NA = is.na(my_data); my_NA
 [1]  FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE
FALSE
[14]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
FALSE
[27] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
TRUE
[40]  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE
TRUE
[53] FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
TRUE
[66] FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE
TRUE
[79]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE
```

Везде, где мы видим *TRUE*, соответствующий элемент *my_data* является *NA*. Точно так же, везде где *FALSE*, соответствующий элемент *my_data* является одной из случайных выборок из стандартного нормального распределения. Протестируем, дает ли выражение *my_data == NA* те же результаты, что и *is.na(my_data)*:

```
> my_data == NA
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[23] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[45] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[67] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
NA NA NA NA NA NA NA NA NA NA
```

Причина получения вектора из всех *NA* в том, что *NA* не является каким-либо реальным значением, это просто заполнитель места, которое не доступно. Поэтому логическое выражение является неполным и интерпретатор *R* не имеет никакого выбора, кроме как возвратить вектор той же длины, что и *my_data*, который содержит все *NA*.

Итак, вернемся к нашей задаче. Теперь, когда у нас есть вектор, *my_NA*, который имеет значение *TRUE* для каждого *NA* и *FALSE* для каждого числового значения, мы можем вычислить общее количество *NA* в наших данных. Хитрость заключается в том, что в *R* значение *TRUE* представляется как число 1, а *FALSE* как число 0. Таким образом, если просуммировать кучу *TRUE* и *FALSE*, то мы получим общее количество *TRUE*:

```
> sum(my_NA)
[1] 54
```

Теперь посмотрим на второй тип отсутствующего значения – *NaN* (*not a number*), что означает "не число" или неопределенное значение. Для создания *NaN* можно, например разделить 0 на 0:

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

Наконец, третий тип неопределенного значения – бесконечное значение *Inf*, которое может быть получено, например делением 1 на 0:

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

Попробуем установить связь между *NaN* и *Inf*. Например, выполним арифметическую операцию вычитания с *Inf*:

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

В заключение отметим, что бесконечное значение *Inf* имеет знак:

```
> -1/0
[1] – Inf
```

что естественным образом согласуется со школьным курсом алгебры.

4. Обработка данных

4.1. Логические выражения

Есть два логических значения, также называемые булевскими - это *TRUE* и *FALSE*. В языке *R* можно построить логические выражения, которые при вычислении будут давать результат либо *TRUE*, либо *FALSE*.

Создание логических выражений требует логических операторов. Первый логический оператор, который мы обсудим это оператор эквивалентности или сравнения, который записывается в виде двух знаков равенства ==, например:

```
> TRUE==TRUE
[1] TRUE
```

достаточно тривиальный результат. Так же, как арифметические, логические выражения могут быть сгруппированы в круглые скобки так, что все выражение *(FALSE == TRUE) == FALSE* принимает значение *TRUE*.

Оператор эквивалентности может быть использован для сравнения чисел и строковых переменных. Используем его, чтобы увидеть равны ли числа 6 и 7, а также имена "Анна" и "Аня":

```
> 6 == 7
[1] FALSE
> "Анна" == "Аня"
[1] FALSE
```

и в обоих случаях получим логическое нет.

Предыдущие выражения имеют значение *FALSE*, поскольку 6 меньше 7, а имена "Анна" и "Аня" не совпадают. К счастью, есть операторы неравенства, которые позволяют нам проверить, если значение меньше или больше другого значения. Оператор < проверяет, является ли значение слева от оператора меньше, чем значение справа от оператора. Применим его:

```
> 6 < 7
[1] TRUE
> "Анна" < "Аня"
[1] TRUE
```

Понятно, что получено логическое да, поскольку 6 < 7 в первом случае. Во втором случае, строковые выражения сравниваются последовательно по буквам, точнее по *ASCII* кодам символов. Поскольку первые две буквы "Ан" совпадают, логическое выражение приняло значение *TRUE* потому, что "н" < "я", точнее код 237 < 255.

Аналогичным образом можно использовать операторы <=, >, >= и !=. Обсудим оператор 'не равно', который представлен как !=. Для того, чтобы изменить значение логического выражения на противоположное используется оператор НЕ в виде восклицательного знака. Выражение *!TRUE* становится эквивалентным *FALSE*, а *!FALSE* означает *TRUE*.

Очевидно, что выражение *TRUE != FALSE* имеет значение *TRUE*. А какое значение будет иметь выражение *(TRUE != FALSE) == !(6 == 7)*:

```
> (TRUE != FALSE) == !(6 == 7)
[1] TRUE
```

Пожалуй, чтобы ответить на этот вопрос до нажатия клавиши Enter, пришлось вычислить отдельно значения правого и левого выражения и результаты сравнить на эквивалентность.

По этой причине стоит избегать сложных логических выражений подобного рода при программировании, они могут привести к ошибкам. Если в какой-то момент, требуется изучить взаимосвязи между несколькими ло-

гическими выражениями нужно дополнительно использовать операторы логическое И и логическое ИЛИ.

В языке R есть два оператора логического И: *&* и *&&*. Оба оператора работают аналогично, за исключением случая, когда одним из операндов является вектор:

```
> TRUE & c(TRUE, FALSE, TRUE,TRUE)
[1]  TRUE FALSE  TRUE  TRUE
> TRUE && c(TRUE, FALSE, TRUE,TRUE)
[1] TRUE
```

Как видно из этого примера, логический оператор *&* вычисляет логическое И для всех элементов вектора, т.е. эта операция идентична операции *c(TRUE, TRUE, TRUE) & c(TRUE, FALSE, FALSE)*. Логический оператор *&&* вычисляет значение только для первого элемента вектора.

Оператор логическое ИЛИ следует аналогичному набору правил. Оператор | обрабатывает весь вектор, в то время как || вычисляет логическое ИЛИ только для первого элемента вектора.

Логические операторы могут быть соединены друг с другом так же, как арифметические операторы. Точно также, как арифметические операции, логические операторы имеет порядок вычислений – сначала вычисляются все операторы *&*, а затем ||.

Теперь, после знакомства с логическими операторами языка *R* воспользоваться рядом функций, которые язык *R* предоставляет для работы с логическими выражениями.

Функция *isTRUE()* оценивает один аргумент и если этот аргумент имеет значение *TRUE*, функция вернет *TRUE*, в противном случае вернет *FALSE*.

Функция *identical()* возвращает *TRUE*, если два объекта *R*, переданные ей в качестве аргументов идентичны. Например:

```
> identical("Анна","Анна")
[1] TRUE
> identical("Анна","анна")
[1] FALSE
> identical(Inf,1/0)
[1] TRUE
> identical(sin(0),tan(0))
[1] TRUE
```

Функции *xor()* выполняет логическую операцию исключающее ИЛИ (или сумма по модулю два) которая возвращает значение *TRUE* тогда и только тогда, когда два ее аргумента имеют противоположные значения. Например:

```
> xor(!!TRUE, !!FALSE)
[1] TRUE
```

При рассмотрении следующих нескольких вопросов, нам потребуется вектор целых чисел. Создадим этот вектор с помощью команды *sample()*:

```
> ints = sample(10); ints
 [1]  5  8  7  4  6  9  3  1 10  2
```

Вектор *ints* является случайной выборкой чисел от 1 до 10 без повторений. Скажем, мы хотим задать некоторые логические вопросы по поводу содержимого вектора *ints*. Например, поинтересуемся тем, какие элементы вектора больше 5:

```
ints > 5
 [1] FALSE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE FALSE
```

Полученный логический вектор мы можем использовать для того, чтобы задать другие вопросы о векторе *ints*.

Функция *which()* использует в качестве аргумента логический вектор и возвращает индексы (или номера) тех элементов вектора, которые являются истинными. Например:

```
> which(ints > 5)
[1] 2 3 5 6 9
```

Подобно функции *which(),* функции *any()* и *all()* используют в качестве аргумента логические векторы. Функция *any()* возвращает значение *TRUE*, если один или более элементов логического вектора имеют значение *TRUE*. Функция *all()* будет возвращать *TRUE*, если все элементы логического вектора равны *TRUE*. Например:

```
> all(ints > 0)
[1] TRUE
> any(ints == 10)
[1] TRUE
> any(ints < 1)
[1] FALSE
```

4.2. Дата и время

R имеет особый способ представления дат и времени, что может быть полезно, если вы работаете с данными, которые показывают, как что-то

меняется с течением времени или если ваши данные содержат некоторую другую временную информацию, такую как даты рождения, периоды времени и тому подобное.

Даты представлены классом *Date*, а время представлено классами *POSIXct* и *POSIXlt*. Значения дат хранятся как количество дней, прошедших с 1970-01-01, а значения времени сохраняются в виде количества секунд, прошедших с 1970-01-01 (для *POSIXct*) или в виде списка секунды, минуты, часы и т.д. (для *POSIXlt*). С помощью команды *Sys.Date()* можно получить текущую дату и сохранить его в переменной *d1*:

```
> d1 = Sys.Date(); d1
[1] "2017-01-09"
```

Используем функцию *class()* для подтверждения того, что *d1* является объектом класса *Date*:

```
> class(d1)
[1] "Date"
```

Можно использовать функцию *unclass(),* чтобы увидеть, как *d1* выглядит внутри и узнать точное количество дней, прошедших с 1970-01-01 до *d1*:

```
> unclass(d1)
[1] 17175
```

Что делать, если нужно ссылаться на дату до 1970-01-01? Нужно создать переменную *d2 = as.Date ("1969-01-01"):*

```
> d2 = as.Date ("1969-01-01"); d2
[1] "1969-01-01"
```

и используя *unclass()* еще раз, увидеть, как *d2* выглядит внутри:

```
> unclass(d2)
[1] -365
```

Как можно было и ожидать, мы получим отрицательное число. В этом случае, это -365, так как 1969-01-01 составляет ровно один календарный год (т.е. 365 дней) перед датой 1970-01-01.

Теперь посмотрим, как *R* хранит время. Можно получить доступ к текущей дате и времени с помощью функции *Sys.time()* без аргументов. Сделаем это, и сохраним результат в переменной t1:

```
> t1 = Sys.time(); t1
[1] "2017-01-09 18:28:59 MSK"
```

Определим класс *t1*:

```
> class(t1)
[1] "POSIXct" "POSIXt"
```

Как упоминалось выше, *POSIXct* является одним из двух способов, с помощью которого *R* представляет информацию о времени (можно игнорировать второе значение *POSIXt*, которое просто используется в качестве общей связки между *POSIXct* и *POSIXlt*). Используем *unclass(),* чтобы увидеть, как *t1* выглядит внутри:

```
> unclass(t1)
[1] 1483975740
```

это количество секунд, прошедших с 01-01-1970. По умолчанию, *Sys.time()* возвращает объект класса *POSIXct*, но мы можем преобразовать результат в *POSIXlt* с помощью функции *as.POSIXlt()*:

```
> t2 = as.POSIXlt(Sys.time()); t2
[1] "2017-01-09 18:59:28 MSK"
> class(t2)
[1] "POSIXlt" "POSIXt"
```

Как можно увидеть, распечатанный формат *t2* является идентичным *t1*, хотя они относятся к разным классам. Теперь, с помощью команды *unclass(t2)* посмотрим, как они отличается внутренне. Для более компактного представления результата используем команду:

```
> str(unclass(t2))
List of 11
 $ sec   : num 28.5
 $ min   : int 59
 $ hour  : int 18
 $ mday  : int 9
 $ mon   : int 0
 $ year  : int 117
 $ wday  : int 1
 $ yday  : int 8
 $ isdst : int 0
 $ zone  : chr "MSK"
 $ gmtoff: int 10800
 - attr(*, "tzone")= chr [1:3] "" "MSK" "MSD"
```

Переменная *t2*, как и все объекты *POSIXlt*, это просто список значений, составляющих дату и время. Если, например, мы хотим использовать только минуты от времени, хранящегося в *t2*, мы можем получить доступ к ним с помощью *t2$min*:

```
> t2$min
[1] 59
```

Теперь, когда мы получили все три типа даты и времени объектов, рассмотрим несколько функций, которые извлекают полезную информацию от любого из этих объектов - *weekdays(), months()* и *quarters()*.

Функции *weekdays()* и *months()* возвращают день недели и название месяца для любой даты:

```
> weekdays(d1)
[1] "понедельник"
> months(t1)
[1] "января"
```

Функция *quarters()* возвращает квартал текущего года (*Q1-Q4*) для любой даты или времени:

```
> quarters(t1)
[1] "Q1"
```

Часто дата и время в наборе данных представлены в таком формате, который *R* не распознает. Функция *strptime()* может быть полезной в этой ситуации. Она преобразует набор символов в *POSIXlt*. В этом смысле она похожа на *as.POSIXlt(),* за исключением того, что входные данные не должны быть в определенном формате (*YYYY-MM-DD*). Для того, чтобы посмотреть, как он работает, сохраним следующую строку символов "23 августа 1974 14:24" в переменной с именем *t3*:

```
> t3 = "23 августа 1974 14:24"
```

Теперь, используем функцию *strptime(t3, "%d %B %Y %H:%M")*, чтобы помочь *R* преобразовать объект даты/времени в формат *POSIXlt*. Результат поместим в новую переменную под названием *t4*:

```
> t4 = strptime (t3, "%d %B %Y %H:%M"); t4
[1] "1974-08-23 14:24:00 MSK"
```

```
> class(t4)
[1] "POSIXlt" "POSIXt"
```

Теперь, с помощью команды *str(unclass(t4))* посмотрите, как этот объект представлен внутренне.

И, наконец, существует целый ряд операций, которые можно выполнять с объектами даты и времени, в том числе арифметические операции (+ и -) и операции сравнения (<, == и т.д.).

Переменная *t1* содержит время, когда ее создали (с использованием *Sys.time()*). Убедимся, что прошло некоторое время с момента создания t1 с помощью оператора 'больше', чтобы сравнить его с текущим временем:

```
> Sys.time() > t1
[1] TRUE
```

Итак, мы знаем, что прошло некоторое время, но сколько? Попробуем вычесть *t1* из текущего времени:

```
> Sys.time() - t1
Time difference of 42.740828 mins
```

Полученная разность дает нам количество времени, которое прошло с момента создания *t1*.

Также можно использовать и другие операторы сравнения. Если нужно установить точный временной интервал между определенными моментами времени, то будет полезна функция *difftime(),* который позволяет указать нужные единицы измерения (*units*). Например, измерим интервал времени прошедший от создания переменной *t1*, до текущего момента в днях:

```
> difftime(Sys.time(), t1, units = 'days')
Time difference of 0.006860386 days
```

Итак мы познакомились с основами того, как работать с датами и временем в *R*. Если появляется необходимость работы с датами и временем часто, то нужно более подробно ознакомиться с пакетом *lubridate*.

4.3. Функции

Функции являются одним из основных строительных блоков на языке *R*. Это небольшие кусочки кода, которые можно использовать, как и любой другой объект *R*. Мы уже использовали некоторые функции и видели,

что у каждой есть имя, за которым обязательно следуют скобки. Большинство функций в *R* возвращают некоторое значение.

Например, функция *Sys.Date()* возвращает строку, представляющую текущую дату и это возвращаемое значение основано на использовании оборудования компьютера. Наряду с этим, другие функции манипулируют входными данными для вычисления возвращаемого значения.

Функция *mean(),* берет вектор чисел в качестве входных данных и возвращает среднее значение всех чисел во входном векторе. Исходные данные для функций называют аргументами или входными параметрами. Передаваемые аргументы размещаются внутри скобок функции. Например, передавая аргумент *c(2, 4, 5)* функции *mean(),* получим:

```
> mean(c(2,4,5))
[1] 3.666667
```

среднее значение трех входных чисел.

Напишем простейшую функцию *just_return*, которая берет входной аргумент *x* и возвращает его без изменений:

```
just_return <- function(x){x}
```

теперь давайте проверим как она работает:

```
> just_return("Первая функция!")
[1] "Первая функция!"
```

Действительно, функция как и планировалось возвратила входной аргумент без изменений.

При написании функций, можно получить серьезное понимание того, как работает интерпретатор *R*. Создатель языка *R*, Джон Чемберс однажды сказал: «Чтобы понять вычисления в *R*, полезны два лозунга: (1) все, что существует, является объектом; (2) все, что происходит является вызовом функции».

Если вы хотите, увидеть исходный код любой функции, просто введите ее имя без каких-либо аргументов или скобки. Например:

```
> just_return
function(x) {x}
```

Давайте создадим более полезную функцию. Например, повторим функциональность функции *mean()*, создав функцию под названием: *our_mean()*. Для расчета среднего значения всех чисел в векторе нужно найти сумму всех чисел в векторе, а затем разделить эту сумму на количе-

ство чисел в векторе. Для нахождения суммы чисел в векторе будем использовать функцию *sum()*, а для нахождения количества всех чисел в векторе - функцию *lenght()*:

```
our_mean <- function(vector) {
  sum(vector)/length(vector)
}
```

Проверим, как работает наша функция:

```
> our_mean(c(4,7,12))
[1] 7.666667
```

Далее, напишем функцию с аргументами по умолчанию. Установка по умолчанию значений для аргументов функции, может быть полезна, если пользователи функции устанавливают определенный аргумент в то же значение чаще всего.

Напишем функцию *reminder()*, которая будет иметь два входных аргумента: некоторое число (*num*) и делитель (*divisor*). Результатом выполнения функции должен быть остаток от деления *num/divisor*.

Положим, что как правило, пользователи хотят знать остаток от деления на 2, поэтому установим для делителя значение по умолчанию равное 2. Теперь напишем функцию:

```
remainder <- function(num, divisor = 2) {
  num %% divisor
}
```

Проверим, как работает наша функция:

```
> remainder(33)
[1] 1
> remainder(33,7)
[1] 5
```

В первом случае используется значение делителя по умолчанию (divisor= 2) и результат равен 33 mod 2 = 1, а во втором – берется значение второго входного параметра 33 mod 7 = 5. Отметим, что когда задаются два параметра важен порядок их написания, однако если записывать имена входных параметров, то порядок неважен, например:

```
> remainder(divisor=7,num=33)
[1] 5
```

```
> remainder(7,num=33)
[1] 5
```

Последний результат показывает, что достаточно указать имена параметров частично и порядок задания параметров может быть произвольным. В общем, нужно всегда следовать принципу чтобы программный код был как можно понятней для понимания. Поэтому приемы с указанием имен параметров частично просто сбивают с толку и их надо избегать.

При всем этом говорить об аргументах функции, может быть интересно, если есть способ (кроме чтения документации), с помощью которого их можно видеть. К счастью, такой способ есть, для этого можно использовать функцию *args()*! Например:

```
> args(remainder)
function (num, divisor = 2)
```

Последний пример позволяет установить довольно интересный факт: несмотря на то, *reminder()* является функцией в записи *args(reminder)* ей отводится роль аргумента или входного параметра другой функции. Действительно, в языке *R* функцию можно передать другой функции в качестве аргумента. Это очень мощная концепция. Давайте напишем скрипт, чтобы увидеть как это работает.

Передать функцию в качестве аргументов другим функциям, можно подобно тому, как передаются входные параметры. Определим следующие функции:

```
add_two_numbers <- function(num1, num2) {num1 + num2}
multiply_two_numbers <- function(num1, num2) {num1 * num2}

some_function <- function(func) {func(2, 4)}
```

У функции *some_function()* мы используем имя аргумента "*func*" показывая тем самым, что в качестве аргумента может быть передано имя функции. В теле функции *some_function()* для любой переданной функции func мы задаем два входных аргумента 2 и 4, по умолчанию.

Теперь достаточно очевидны результаты следующих двух вызовов функции *some_function()*:

```
> some_function(add_two_numbers)
[1] 6
> some_function(multiply_two_numbers)
[1] 8
```

Давайте создадим некоторую «универсальную» вычислительную функцию *evaluate()*, которая способна использовать в качестве аргумента целый ряд функций, например *sum()*, *mean()*, *sin()* и даже *just_return()*. Эта функция должна выполнять следующее:

1) *evaluate(sum,1:3)* должна вычислить 6;
2) *evaluate(mean,c(1,2,6))* должна вычислить 3;
3) *evaluate(sin,pi/4)* должна вычислить 0.7071068;
4) *evaluate(just_return, "Hi, World! ")* должна вычислить "*Hi, World!* ".

Как видно из этого задания, второй входной параметр функции *evaluate()* может быть любого типа, поэтому универсальным способом его описания может быть некоторый абстрактный объект *dat*. Попробуем записать функцию *evaluate()* следующим образом:

```
evaluate <- function(func, dat) {func(dat)}
```

Проверим, как работает наша функция:

```
> evaluate(sum,1:3)
[1] 6
> evaluate(mean,c(1,2,6))
[1] 3
> evaluate(sin,pi/4)
[1] 0.7071068
> evaluate(just_return, "Hi, World! ")
[1] "Hi, World! "
```

Функция правильно выполнила все задания. Нетрудно заметить, что построенная нами функция *evaluate()* действительно «универсальна», она может принимать в качестве входного аргумента и другие функции. Попробуйте использовать ее для вычисления другой, известной вам функции.

Идея передачи функции в качестве аргумента других функций является важным и фундаментальным понятием в программировании. Может показаться удивительным, что можно передать функцию как аргумент без предварительного определения передаваемой функции. Функции, которые используются без названия известны как анонимные функции.

Давайте используем функцию *evaluate()*, чтобы исследовать, как работают анонимные функции. Для первого аргумента функции *evaluate()* мы напишем крошечную функцию, которая помещается на одной строке. А вторым аргументом будет значение входного параметра для этой анонимной функции:

```
> evaluate(function(x){x+1}, 20)
```

```
[1] 21
```

Результат выполнения данной анонимной функции очевиден, для значения входного аргумента *x = 20* она вычислила значение *x = x + 1*.

Попробуем использовать функцию *evaluate()* наряду с анонимной функцией, чтобы вернуть первый элемент вектора *c (8, 4, 0)*. Наша анонимная функция должна использовать только один входной аргумент, который должен быть переменной *x*. Первый элемент вектора обозначим *x[1]* и получим:

```
> evaluate(function(x){x[1]}, c(8, 4, -19))
[1] 8
```

Что необходимо предпринять для того, чтобы функция возвратила последний элемент вектора. Конечно, в данном конкретном случае достаточно вместо *x[1]* использовать *x[3]*, но в общем случае к последнему элементу вектора следует обращаться как *x[length(x)]*. Поэтому, возвращать последний элемент любого вектора будет функция:

```
> evaluate(function(x){x[length(x)]}, c(8, 4, -19))
[1] -19
```

В дальнейшем мы часто будем использовать функцию *paste()*. Эта функция допускает использование неопределенного количества входных аргументов, которые она объединяет в единую строку. Вот пример ее использования:

```
> paste("Программирование","на R","это интересно!")
[1] "Программирование на R это интересно!"
```

Посмотрим на описание этой функции:

```
> paste
function (..., sep = " ", collapse = NULL)
```

Сначала идет многоточие, а вслед за ним два аргумента, каждый из которых имеет значение по умолчанию *sep = " "* и *collapse = NULL*. Следует подчеркнуть, что в языке *R* является строгим правилом, что все аргументы функции, следующие за многоточием должны иметь значение по умолчанию.

Многоточие используется для задания произвольных входных аргументов. В частности для функции *paste()* входными аргументами могут служить любые объекты *R*, которые можно представить в формате

character. Если формат другой, то функция *paste()* предварительно выполняет преобразование, подобное *as.character()*. Наконец, если входным аргументом является вектор, то функция *paste()* обрабатывает последовательно каждый элемент. Приведем пример, в котором аргументами являются два вектора один числовой, другой строковый:

```
> nth <- paste(1:12, c("st", "nd", "rd", rep("th", 9)), sep=""); nth
 [1] "1st"  "2nd"  "3rd"  "4th"  "5th"  "6th"  "7th"  "8th"  "9th"  "10th" "11th" "12th"
```

В переменной *ntn* мы сформировали вектор окончаний английских порядковых числительных. Теперь используем его для указания порядкового значения каждого месяца года:

```
> paste(month.abb, "is the", nth, "month of the year.", sep = " ")
 [1] "Jan is the 1st month of the year."     "Feb is the 2nd month of the year."
 [3] "Mar is the 3rd month of the year."     "Apr is the 4th month of the year."
 [5] "May is the 5th month of the year."     "Jun is the 6th month of the year."
 [7] "Jul is the 7th month of the year."     "Aug is the 8th month of the year."
 [9] "Sep is the 9th month of the year."     "Oct is the 10th month of the year."
[11] "Nov is the 11th month of the year."    "Dec is the 12th month of the year."
```

Давайте напишем свою собственную модифицированную версию функции *paste()*. Например, потребуем, чтобы любая строка, сформированная нашей функцией *paste_our()* имела ограничители *Stop* и *Start.* Очевидно, функция *paste_our()* может быть реализована в виде:

```
> paste_our<- function(...){ paste("Start", ..., "Stop") }
```

и вот пример ее использования:

```
> paste_our("С добрым утром!")
[1] "Start С добрым утром! Stop"
```

Рассмотрим пример того, как «распаковать» аргументы функции, использующей многоточие для их размещения. Функция *paste_our()* будет находить поле многоточия в списке входных параметров, распаковывать аргументы из этого поля, назначать значения аргументов переменным и передавать последние функции *paste()*.

4.4. Циклы

Во многих языках программирования циклы служат базовыми строительными блоками, которые используются для любых повторяющихся задач. Однако в языке *R* чрезмерное или неправильное использование циклов

может привести к ощутимому падению производительности — и это при том, что способов написания циклов в этом языке необычайно много [9].

Рассмотрим особенности использования штатных циклов в *R*, а также познакомимся с функцией foreach из одноименного пакета, которая предлагает альтернативный подход. С одной стороны, *foreach* объединяет лучшее из штатной функциональности, с другой — позволяет с легкостью перейти от последовательных вычислений к параллельным с минимальными изменениями в коде.

Начнем с того, что часто оказывается неприятным сюрпризом для тех, кто переходит на *R* с классических языков программирования: если мы хотим написать цикл, то стоит перед этим на секунду задуматься. Дело в том, что в языках для работы с большим объемом данных циклы, как правило, уступают по эффективности специализированным функциям обработки данных.

Чтобы понять, насколько важна эта особенность, обратимся к цифрам. Допустим, у нас есть простая таблица *testDF* из двух столбцов *a* и *b*. Первый растет от 1 до 100 000, второй уменьшается со 100 000 до 1:

```
testDF <- data.frame(a = 1:100000, b = 100000:1)
```

Если мы хотим сформироватать третий столбец, который будет суммой первых двух, то много начинающих *R*-разработчиков могут написать код такого вида:

Пример 4.1. Использование цикла for

```
for(row in 1:nrow(testDF))
  testDF[row, 3] <- testDF[row, 1] + testDF[row, 2]
```

Подобный цикл занимает несколько десятков секунд, в зависимости от процессора, хотя тот же результат можно получить за тысячные доли секунды, воспользовавшись функцией для работы с таблицами из пакета *dplyr*:

Пример 4.2. Использование функции mutate()

```
testDF <- testDF %>% mutate(c = a + b)
```

Основная причина такой серьезной разницы в скорости заключается в потере времени на чтении и записи значений ячеек в таблице. Именно благодаря оптимизациям на этих этапах и выигрывают специальные функции.

Классические циклы

Язык R поддерживает основные классические операторы для написа-

ния циклов.

Оператор for — самый распространенный тип циклов. Синтаксис прост и знаком разработчикам на различных языках программирования. Цикл for выполняет переданную ему функцию для каждого элемента:

```
# печатаем числа от 21 до 99
        for(j in 21:99)
          print(j)

# печатаем все буквы из вектора LETTERS
        for(chars in LETTERS)
          print(chars)
```

Оператор while — тоже часто встречается в других языках программирования. В цикле while перед каждой итерацией проверяется логическое условие, и если оно соблюдается, то выполняется итерация цикла, если нет, то цикл завершается:

```
# печатаем числа от 21 до 99
        j = 21
        while(j<100)
          {j=j+1; print(j)}
```

Оператор repeat повторяется до тех пор, пока в явном виде не будет вызван оператор break:

```
# печатаем числа от 21 до 99
        j = 20
        repeat { j = j + 1; print(j);
                 if (j == 99) break}
```

Следует отметить, что циклы for, while и repeat всегда возвращают NULL, в качестве результирующего значения и в этом их отличие от следующей группы циклов.

Циклы на основе функции apply()

Функции для организации циклов — apply(), lapply(), sapply() и vapply() отличаются друг от друга тем, к чему цикл применяется и что возвращает [10].

Функция apply() применяется к матрицам:

```
apply(X, MARGIN, FUN, ...),
```

первый параметр (X) указывает исходную матрицу, второй параметр

(MARGIN) уточняет способ обхода матрицы (1 — по строкам, 2 — по столбцам, c(1,2) — по строкам и столбцам), третий параметр задает функцию FUN, которая будет вызвана для каждого элемента матрицы. Результаты всех вызовов будут объединены в один вектор или матрицу, которую функция apply() вернет в качестве результирующего значения.

Создадим матрицу m размером 3 x 3:

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

Посчитаем сумму ячеек для каждой строки и каждого столбца:

```
> apply(m, MARGIN = 1, FUN = sum)
[1] 12 15 18
> apply(m, MARGIN = 2, FUN = sum)
[1]  6 15 24
```

В этом примере использована системная функция sum, но можно использовать свою функцию.

Функции sapply(), lapply(),vapply()

Функция sapply() применяется аналогично apply(), но только к векторам и спискам. Она полезна при сложных итерационных вычислениях:

```
sapply(X, FUN, ..., simplify = TRUE),
```

первый параметр (*X*) указывает вектор или список, *FUN* – функция применяемая к *X*, *simplify = TRUE* — выводимый результат представляется в виде вектора или матрицы, в противном случае – в виде списка.

Другой распространенной функцией из семейства *apply* является *lapply()*:

```
lappy(X, FUN, …),
```

которая является версией функции sapply(), а именно:

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

Функция *lapply()* является полезной при работе со списками, позволяет применять различные функции к элементам списка (числовые функ-

ции можно применять только в том случае, если все элементы списка *X*

Функция *vapply()* аналогична *lapply()*. Дополнительно она добавляет проверку типов возвращаемого значения.

Если lapply выполняет функцию для каждого элемента в списке, то можно реализовать функционал, очень похожий на for, но с получением возвращаемого значения от всего цикла.

Реализуем пример 4.1 с использованием функций sapply() и vappy() и оценим эффективность их использования. Представим столбцы фрейма *testDF* в виде векторов:

```
a <- testDF$a
b <- testDF$b
```

Для функции *sapply()* сложение этих векторов и формирование столбца *c* выполним следующим образом:

```
Пример 4.3. Использование функции sapply()
result <- sapply(1:nrow(testDF), function(row) {a[row] + b[row]})
testDF$c <- result
```

Так как мы заранее знаем возвращаемый тип для каждой итерации, то можно заменить функцию *sapply()* на *vapply()*, указав в параметре *FUN.VALUE*, что каждая итерация возвращает одно целое число *integer(1)*:

```
Пример 4.4. Использование функции vapply()
result <- vapply(1:nrow(testDF), FUN.VALUE = integer(1),
                 function(row) {a[row] + b[row]})
testDF$c <- result
```

Используя материалы раздела 4.5, сравните скорости выполнения программ примеров 4.1 – 4.4 для числа строк во фрейме testDF от 0 до 200000 с интервалом 50000. Получаемые данные изобразите на графике.

Пакет foreach

Функция *foreach()* не является базовой для языка *R*. Соответствующий пакет необходимо установить и подключить перед ее вызовом:

```
install.packages("foreach")    # Установка пакета
library(foreach)               # Подключение пакета
```

Несмотря на то, что *foreach()* является сторонней функцией, она часто используется при формировании циклов по двум причинам:

- она возвращает значения, которые собираются из результатов каждой итерации, при этом можно определить свою функцию и реализовать

любую логику сбора финального значения цикла из результатов итераций;

- имеется возможность использовать многопоточность и запускать итерации параллельно.

Например, для чисел от 1 до 5 на каждом шаге цикла число возводится в квадрат. Результаты всех шагов записываются в переменную res в виде списка:

```
res = foreach(i=1:5) %do% (i^2)
```

Для управления формой представления результата циклической обработки удобно использовать опцию *combine*. Если мы хотим, чтобы результатом был не список, а вектор, то необходимо указать *"c"* (формирование вектора) в качестве значения опции *combine* (обратите внимание, что перед указанной опцией следует ставить точку):

```
> res = foreach(i=1:5,.combine="c") %do% (i^2); res
> [1]  1  4  9 16 25
```

Если в результате циклической обработки мы получаем на каждом шаге вектор, то для формирования результата в виде матрицы из этих векторов в качестве значения опции combine можно указать *"cbind"*:

```
> x <- foreach(i=1:4, .combine="cbind") %do% rnorm(4); x
           result.1    result.2    result.3    result.4
[1,] -0.20850006 -1.486203 -0.2704967  0.5673699
[2,]  1.51903988  0.437409  1.2537232  0.3196177
[3,]  0.08122457 -2.853031  0.2078843 -0.1328761
[4,] -0.45572878  1.466209 -2.3159257  0.9353287
```

Здесь мы сгенерировали четыре вектора из четырех случайных чисел и сформировали из них матрицу размерностью 4x4.

Мы можем также использовать операции "+" и "*" для формирования результата. Например, если в качестве значения опции *combine* задать оператор "+", тогда в переменной *res* будет сформирована сумма значений полученных на каждом шаге цикла:

```
> res = foreach(i=1:5,.combine="+") %do% (i^2); res
[1] 55
```

Наконец, в теле цикла можно использовать собственные функции для формирования результата циклической обработки.

Допустим, у нас есть функция, которая возвращает *data.frame*:

```
customFun <- function(param) {data.frame(param = param,
                result1 = sample(1:100, 1), result2 = sample(1:100, 1)) }
```

Если мы хотим вызвать эту функцию 10 раз и объединить результаты в один фрейм, то в *combine* для объединения можно использовать *"rbind"*. В результате мы получим фрейм *result*, объединяющий все результаты циклической обработки:

```
> result <- foreach(param = 1:10,.combine = "rbind") %do%  customFun(param); result
      param result1 result2
1         1      47      67
2         2      96      82
3         3      92      22
4         4      18      31
5         5      73      64
6         6      34      69
7         7      95       2
8         8      51      15
9         9      44      24
10       10      56      56
```

В *.combine* можно также использовать свою собственную функцию, причем с помощью дополнительных параметров можно оптимизировать производительность, если ваша функция умеет принимать больше чем два параметра за один раз (в документации *foreach* можно почитать описание параметров *.multicombine* и *.maxcombine*).

При этом стоит понимать, что для совсем простых операций, вроде нашего примера, функция *foreach()* приспособлена плохо — слишком много накладных расходов на инициализацию каждой транзакции. Функцию *foreach()* целесообразно использовать, когда каждая итерация занимает достаточно много времени и требуется агрегировать результаты выполнения всех итераций. Однако главное преимущество функции *foreach()* заключается в легкости перехода от последовательной обработки к параллельной. Фактически этот переход осуществляется заменой *%do%* на *%dopar%*, но при этом нужно учитывать следующие три особенности.

Во-первых, до вызова функции *foreach()* должен быть зарегистрирован *"parallel backend"*, то есть конкретная реализация параллельных процессов. В языке *R* есть несколько подобных реализаций: *doParallel, doSNOW* и *doMC* со своими особенностями. Рассмотрим одну из них:

```
>library(doParallel)          # Загружаем библиотеку в память
>cl <- makeCluster(8)         # Создаем «кластер» на восемь потоков
>registerDoParallel(cl)       # Регистрируем «кластер»
>system.time({foreach(i=1:8) %dopar% Sys.sleep(1)})
```

```
пользователь     система      прошло
        0.00        0.00        1.13
```

Нет никакой необходимости каждый раз перед *foreach* создавать, а затем удалять parallel backend. Как правило, он создается один раз в программе и используется всеми функциями, которые могут с ним работать.

Во-вторых, надо явно указать, какие пакеты необходимо загрузить в рабочие потоки с помощью параметра *packages*. Например, на каждой итерации *i* будем создавать файл *file_i.txt* с помощью пакета *readr*:

```
foreach(i=1:8, .packages = "readr") %dopar%
        write_csv(data.frame(id = "dopar"), paste0("file_", i, ".txt"))
```

В-третьих, вывод на консоль в параллельном потоке не отображается на экране. Иногда это может существенно усложнить отладку, поэтому обычно сложный код сначала пишут без параллельности, а потом заменяют *%do%* на *%dopar%.*

Следует напомнить, что параллельность вычислений не всегда означает увеличение скорости выполнения. Например, если на каждой итерации все процессорное время занимается на 100%, то, запустив эту операцию на этом же процессоре в несколько потоков, мы увеличим общее время выполнения, так как процессор будет переключаться между задачами. Кроме того, каждый поток создаст свою копию данных, и загрузка оперативной памяти резко возрастет. Этот пример подчеркивает, что надо заранее понимать, какого эффекта планируется достигнуть при использовании параллельной обработки.

Например, многие функции в R могут работать только в одном потоке, так что с ними можно использовать все ресурсы компьютера. Также есть отдельный класс задач, где итерации почти не используют процессорное время, но при этом блокируют дальнейшее выполнение программы. Например, вызов внешних сервисов и ожидание их ответа.

Возьмем внешний веб-сервис http://md5.jsontest.com, который считает хэш *MD5* для переданной строки. Напишем цикл, который для каждой строки из сформированного списка тестовых строк вызывает веб-сервис:

```
library(RCurl)
library(jsonlite)
string_list <- sapply(1:100, function(x) sprintf("string %2d", x)) #список тестовых строк
results_do <- foreach(str = string_list, .combine = c) %do%
{ out = getURL(url = sprintf("http://md5.jsontest.com/?text=%s", URLencode(str)),
                    postfields = "{}",
                    httpheader = "Content-Type: application/json;charset=utf-8",
                    verbose=FALSE)
checkResult <- fromJSON(out)
```

```
print(checkResult$md5)
}
```

Большую часть времени этого цикла займет обмен данными между веб-сервисом, поэтому если переписать его на параллельную обработку, то общее время выполнения может существенно сократиться:

```
library(doParallel)              # Загружаем библиотеку в память
cl <- makeCluster(8)             # Создаем «кластер» на восемь потоков
registerDoParallel(cl)           # Регистрируем «кластер»
string_list <- sapply(1:100, function(x) sprintf("string %2d", x))
results_dopar <- foreach(str = string_list,
                 .combine = c,
                 .packages = c("RCurl", "jsonlite")) %dopar%
                {out = getURL(url = sprintf("http://md5.jsontest.com/?text=%s",
                 URLencode(str)),
                  postfields = "{}",
                  httpheader = "Content-Type: application/json;charset=utf-8",
                  verbose=FALSE)
checkResult <- fromJSON(out)
print(checkResult$md5)
}
stopCluster(cl)                   # Останавливаем «кластер»
```

В заключение раздела можно отметить, что при работе с большим объемом данных циклы не всегда являются лучшим выбором. Использование специализированных функций для выборки, агрегации и трансформации данных чаще всего эффективнее классических циклов. Причем, основное отличие классических циклов *for, while* и *repeat* от группы функций на основе *apply* заключается в том, что последние возвращают значение.

Использование циклов *foreach* из одноименного пакета позволяет упростить написание циклов, гибко оперировать с возвращаемыми значениями в итерациях, а за счет подключения многопоточной обработки еще и намного увеличить производительность решения.

4.5. Время выполнения кода

Часто возникает необходимость оценить время выполнения функции или участка кода с целью оптимизации или выявления «узких» мест. Для измерения времени выполнения кода в языке *R* удобно использовать функцию *system.time()* из базового пакета *base*, а также специализированные функции из пакетов *rbenchmark* и *microbenchmark* [11].

Функция system.time()

Самым простым инструментом для измерения времени выполнения

кода является функция *system.time()* из пакета *base*. В качестве аргумента функция принимает выражения и возвращает время их выполнения. В основе функции *system.time()* используется функция *proc.time()*, которая показывает время, прошедшее с начала запуска *R*-сессии. Принцип работы функции *system.time()* очень простой: делаются замеры до и после выполнения выражения, затем первый замер вычитается из второго.

Измерим время выполнения функции *Sys.sleep(),* которая останавливает выполнение кода на заданный интервал времени (в секундах):

```
> system.time(Sys.sleep(2))
пользователь    система     прошло
       0.006      0.006      2.000
```

Как видим, выполнение данной операции заняло ровно две секунды.

Минимальный инетрвал времени, который может зафиксировать функция *system.time()* равен 1/1000 секунды. Если выражения выполянются быстрее, они не будут зафиксированы функцией *system.time()*. Например, оценим время вычисления среднего значения 10000 нормально распределенных значений:

```
x = rnorm(10^5)
> system.time(sum(x) / length(x))
пользователь    система     прошло
           0          0           0
```

Одним из способов обойти это ограничение может быть многократное повторение выражения и фиксация общего времени выполнения всех повторений:

```
> system.time(replicate(100, sum(x) / length(x)))
пользователь    система     прошло
       0.046      0.000      0.046
```

В этом примере с помощью функции *replicate()* мы повторили выражение *sum(x)/length(x)* 100 раз.

Тот же самый эффект можно получить и с помощью обычного цикла *for()*:

```
> system.time(for (i in seq_len(100)) sum(x) / length(x))
пользователь    система     прошло
       0.046      0.000      0.046
```

Вместо подобных решений можно использовать специальные пакеты, предназначенные для измерения производительности кода, в частности,

пакеты *rbenchmark* и *microbenchmark*. Основной принцип работы этих пакетов заключается в многократном выполнении выражений и расчёта ряда интегральных показателей, в частности, суммы, среднего значения или медианы времени выполнения всех попыток.

Пакет rbenchmark

Основа пакета *rbenchmark* — функция *benchmark()*. Данная функция работает следующим образом: указанные в качестве аргументов выражения выполняются заданное количество раз (по умолчанию 100) и вычисляется время, затраченное на выполнение всех попыток. В качестве аргументов функции *benchmark()* необходимо передать выражения или функции, а также количество повторений, передаваемых аргументом *replications* (анализ функции benchmark() показал, что, данная функция использует system.time() и replicate(), рассмотренные в предыдущем разделе).

Для примера возьмём три способа расчёта среднего арифметического значения для матрицы данных *x = replicate(10, rnorm(10^4))*, размерностью 10000x10. Представим эти способы в виде самостоятельных функций для удобства их вызова при работе с *benchmark()*:

```
1) apply_means <- function(x) {apply(x, 2, mean)}

2) loop_means <- function(x) {n.vars <- ncol(x)
        res <- double(n.vars)
        for (i in seq_len(n.vars))
        res[i] <- mean(x[, i])
        return(res) }

3) ColMeans(x)
```

Убедимся, что функции возвращают одинаковый результат. Сделать это можно с помощью функции *identical()*:

```
identical(apply_means(x), loop_means(x), colMeans(x))
#> [1] TRUE
```

Теперь, подключив пакет *rbenchmark*, мы можем сравнить время работы каждого из выбранных нами способов вычисления средних по столбцам:

```
> library(rbenchmark)
> benchmark(apply_means(x), loop_means(x), colMeans(x), replications = 100)
                test replications elapsed relative user.self sys.self user.child sys.child
1 apply_means(x)          100   0.487   10.146     0.358    0.128          0         0
3     colMeans(x)         100   0.048    1.000     0.048    0.000          0         0
2  loop_means(x)          100   0.309    6.438     0.261    0.047          0         0
```

Наиболее важны в результатах функции *benchmark()* столбцы *elapsed* и *relative*. Столбец *elapsed* показывает время в секундах, затраченное на выполнение соответствующей функции. Как видно из примера, самыми медленными оказались функции *apply_means()* и *loop_means(),* а самой быстрой функция *colMeans()*.

Показатель *relative* дает информацию о разнице во времени относительно самой быстрой функции (в нашем случае это *ColMeans()*), т.е. время самой быстрой функции берётся за единицу, и рассчитывается относительное время для остальных.

Для более удобного просмотра можно отфильтровать вывод функции *benchmark()* с помощью аргумента *columns*. Также может быть полезен аргумент *order*, позволяющий отсортировать вывод по любому из столбцов. Для примера зададим набор показателей, которые мы хотим включить в таблицу (пусть это *test, elapsed* и *relative*), и отсортируем выдачу по столбцу *elapsed* по возрастанию значений:

```
> benchmark(apply_means(x), loop_means(x), colMeans(x),
+ replications = 100, order = "relative",
+ columns = c("test", "elapsed", "relative"))
                test elapsed relative
3     colMeans(x)   0.050     1.00
2  loop_means(x)   0.313     6.26
1 apply_means(x)   0.425     8.50
```

Таким образом, сравнив несколько альтернатив решения задачи, можно сделать обоснованный выбор в пользу наиболее эффективного варианта.

Таким образом, несмотря на то, что пакет *rbenchmark* использует функцию *system.time()*, он преодолевает её ограничения (минимальный фиксируемый интервал времени в 1/1000 секунды и вариативность результатов при многократном выполении) путём многократных повторений и расчёте общего времени выполнения всех повторений.

Пакет microbenchmark

Функция *microbenchmark()* одноименного пакета работает сходным с функцией *benchmark()* образом, но предоставляет более гибкие средства по управлению процессом выполнения выражений и, в отличии от функции *benchmark()*, использует собственную реализацию измерения времени выполнения и организацию повторных испытаний. Пакет *microbenchmark* обеспечивает возможность:

- измерения времени выполнения выражения вплоть до наносекунд;
- контроля последовательности выполнения выражений;

- проведения предварительных испытаний («прогрева») до начала процесса измерений.

Также с помощью функции *microbenchmark()* можно получить исходную информацию о времени выполнения каждой попытки, что даёт достаточно широкие возможности по обработке и анализу полученных результатов.

В таблице ниже представлено время выполнения функций вычисления среднего значения из предыдущего примера, полученное с помощью функции *microbenchmark()*:

```
> library(microbenchmark)
> res <- microbenchmark(apply_means(x), loop_means(x), colMeans(x), times = 100)
> print(res, unit = "ms", order = "median")
Unit: milliseconds
              expr        min         lq       mean     median         uq        max neval
     colMeans(x) 0.470562 0.4778775 0.5052331 0.496539 0.5045515  1.307695   100
   loop_means(x) 2.454602 2.5125725 2.8582165 2.548990 2.7474840 14.433216   100
  apply_means(x) 3.205114 3.2960530 4.0908680 3.341933 3.5446620 14.581255   100
```

Все результаты представлены в виде описательных статистик, рассчитанных из времени выполнения каждой попытки. Наиболее информативный столбец - это столбец *median*, который показывает медиану времени выполнения выражения для всех попыток.

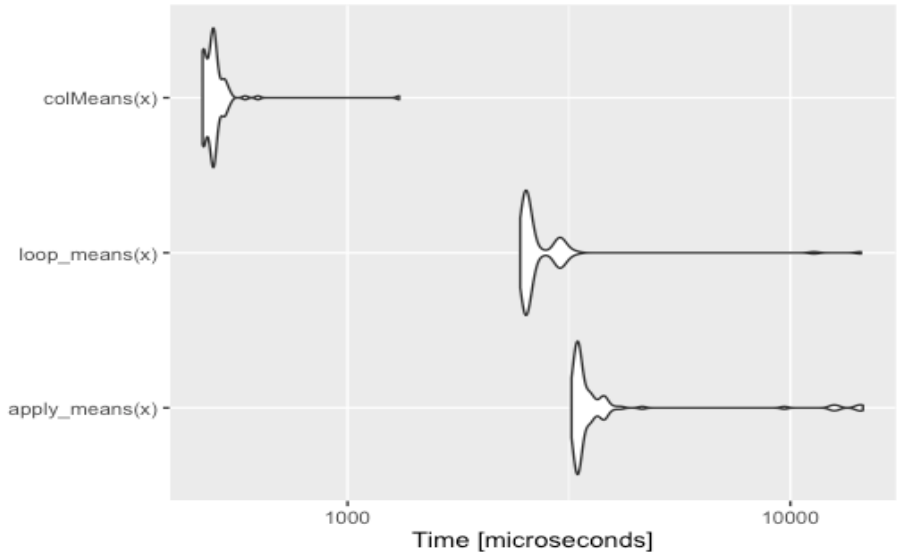
Вся полученная информация о попытках применения функций вычисления средних записана в отдельную переменную *res*. С помощью функции str() можно увидеть структуру переменной:

```
> str(res)
Classes ‘microbenchmark’ and 'data.frame': 300 obs. of  2 variables:
 $ expr: Factor w/ 3 levels "apply_means(x)",..: 2 1 1 1 1 1 2 3 2 3 ...
 $ time: num  2509645 3311244 3300025 3813254 3312101 ...
```

Переменная *res* представляет собой список и включает в себя две переменные: *expr* (выражение) и *time* (время выполнения). На основе этой информации и рассчитываются описательные статистики, приведённые в примере применения функции *microbenchmark()*. Наличие исходных данных о каждой попытке позволяет самостоятельно выбирать, рассчитывать и сравнивать предпочтитаемые показатели.

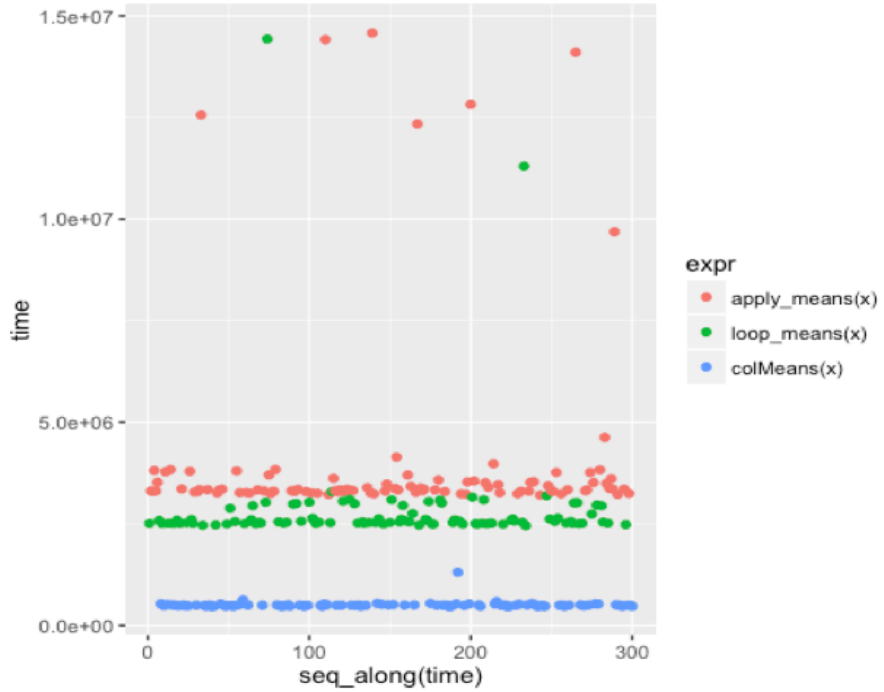
Помимо настройки формата вывода, выбора показателей, наличие информации о времени выполнения выражения в каждой попытке, пакет *microbenchmark* позволяет визуализировать результаты оценки времени выполнения выражения. Например, с помощью функции *autoplot()* из пакета *ggplot2*, можно получить следующий график:

```
> library(ggplot2)
> autoplot(res)
```



Ещё один довольно интересный способ графического представления результатов измерения скорости выполнения кода с помощью функции *qplot()* представлен ниже:

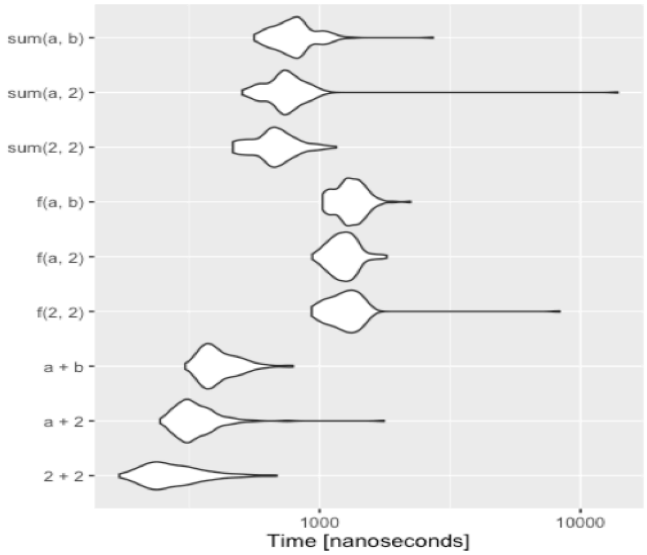
```
> qplot(y = time, data = res, colour = expr)
```



Таким образом, функция *microbenchmark()* является оптимальным инструментом для анализа проивзодительности кода. При этом минимально фиксируемый интервал времени — 1 наносекунла, что позволяет тести-

ровать даже небольшие, быстровыполняющиеся участки кода. Например, сравним время вычислений 2 + 2, выполненных различными способами:

```
> f <- function(a, b) {a + b}
> a <- 2
> b <- 2
> tpt <- microbenchmark(2+2,a+2,a+b,f(2,2),f(a,2),f(a,b),sum(2,2),sum(a,2),sum(a,b))
> autoplot(tpt)
```



5. Базовые функции

Настоящий раздел включает справочную информацию о базовых функциях, которые удобно использовать при программировании. Функции разбиты по тематическим разделам для облегчения поиска.

Напомним, что информацию о любой функции можно оперативно получить с помощью команды *?func_name*, где *func_name* - имя интересующей функции. С помощью команды *func_name* (без предваряющего знака *?*) можно получить описание и байткод функции.

5.1. Получение информации об объектах

- *help(тема)* - документация по *теме*;
- *help.search("тема")* - поск информаци по *"теме"* во всей встроенной справочной системе *R*;
- *apropos("выражение")* - выполняет поиск всех функций, в имени которых содержится *"выражение"*;
- *help.start()* - локально запускает *html*-версию встроенной справочной системы *R*;
- *str(a)* - отображает внутреннюю структуру объекта *a*;
- *summary(a)* - выводит обобщенную информацию об объекте *a*;
- *ls()* - выводит список всех объектов, находящихся в рабочей среде программы;
- *dir()* - отображает содержимое рабочей директории программы.

5.2. Ввод и сохранение данных

- *data(x)* - загружает таблицу данных *x*, сохраненую ранее при помощи команды *save()*;
- *library(x)* - загружает дополнительный пакет (библиотеку) *x*;
- *read.table(file)* - считывает файл с данными и создает из него таблицу данных (*dataframe*);
- *read.csv("filename", header = TRUE)* - то же, что и *read.table(),* - но с готовыми настройками для считывания *csv*-файлов;
- *read.delim("filename", header = TRUE)* - то же, что и *read.table(),* - но с готовыми настройками для считывания файлов, в которых - значения переменных разделены знаком табуляции (*tab-delimited files*);
- *save(file, ...)* - сохраняет указанные объекты *(...)* в двоичном файле - *XDR*-формата, с которым можно работать в любой операционной системе;
- *save.image(file)* - сохраняет все объекты, созданные в ходе работы, в виде специфичного для *R rda*-файла;

- *cat(..., file = "", sep = " ")* - превращает все указанные аргументы *(...)* в текст и сохраняет результат в виде файла, разделитель между - получаемыми текстовыми значениями задается при помощи аргумента *sep;*
- *print(a)* - выводит на экран объект *a* - это функция общего назначения, конкретный результат ее работы будет зависеть от класса объекта *a;*
- *format(x,...)* - позволяет отформатировать объект *x* так, что - он будет выглядеть "аккуратно" при выводе на экран;
- *write.table(x, file = "", row.names = TRUE, col.names = TRUE, sep =" ")* - сохраняет объект *x* в файл, предварительно преобразовав этот объект в таблицу данных (*dataframe*);
- *sink(file)* - выводит результы выполнения других команд в файл в режиме реального времени;
- sink() - прекращает действия предыдущей команды.

Для операций с данными, хранящимися в базах данных, существуют специальные - *R*-библиотеки: *RODBC, DBI, RMySQL, RPgSQL, ROracle*. Библиотеки *XML, hdf5, netCDF* могут пригодиться для работы с файлами других форматов.

5.3. Создание векторов и таблиц данных

- *c(...)* - функция общего назначения объединяет аргументы в один вектор определенного типа;
- *seq(from = , to = , by =)* - формирует последовательность числовых или других значений с заданным шагом;
- *rep(x, times =)* - повторяет *x* столько раз, сколько задано аргументом times;
- *data.frame(...)* - создает таблицу данных из поименованных или непоименованных аргументов;
- *list(...)* - создает список объектов;
- *array(x, dim =)* - создет массив данных из объекта *x*, аргумент *dim* используется для указания размерности массива;
- *matrix(x, nrow = , ncol =)* - создает матрицу из вектора *x*, количество строк задается аргументом *nrow*, количество столбцов - аргументом *ncol*. Если объект *x* не обдадает достаточной длиной, то его элементы при создании матрицы будут использованы повторно (*recycling*);
- *factor(x, levels =)* - превращает вектор *x* в фактор, число уровней фактора задается при помощи аргумента *levels*;
- *gl(n, k, length = n*k, labels = 1:n)* - создает фактор, где *n* - количество уровней фактора, *k* - число повторов для каждого уровня, *length* -

размер итогового объекта, *labels* - необязательный аргумент, который можно использовать для указания названий каждого уровня фактора;

- *rbind(...)* - функция построчно объединяет свои аргументы, создавая матрицу или таблицу данных;
- *cbind(...)* - функция, аналогичная предыдущей, отличие состоит лишь в том, что *cbind()* - объединяет свои аргументы в одну матрицу или таблицу данных по столбцам, а не по строкам.

Индексирование векторов

- *x[n]* - возварщает *n*-й элемент вектора *x;*
- *x[-n]* - возвращает все элементы вектора *x*, за исключением *n*-го;
- *x[1:n]* - возвращает первые *n* элементов вектора *x;*
- *x[-(1:n)]* - возвращает элементы вектора *x* с *n+1* до последнего;
- *x[c(1,4,2)]* - возвращает определенные элементы вектора *x* (здесь 1-й, 4-й и 2-й);
- *x["name"]* - возвращает элемент вектора *x* с именем *"name";*
- *x[x > 3]* - возвращает все элементы вектора *x* со значением *>3;*
- *x[x > 3 & x < 5]* - возвращает все элементы вектора x со значением *>3* и *<5;*
- *x[x %in% c("a","and","the")]* - возвращает только те элементы вектора *x*, которые указаны после оператора *%in%* (здесь из вектора *x* были бы извлечены текстовые значения - *"a", "and",* и *"the").*

Индексирование списков, матриц и таблиц данных

- *x[[n]]* - возвращает *n*-й элемент списка *x;*
- *x[["name"]]* - возвращает элемент списка или таблицы *x* с именем *"name";*
- *x$name* - команда, идентичная предыдушей.

5.4. Конвертация и тестирование объектов

Конвертация R-объектов

Следующие команды конвертируют объект *x* в объект соответствующего класса:

- *as.array(x)* - массив данных;
- *as.data.frame(x)* - таблица данных;
- *as.numeric(x)* - числовой вектор;
- *as.logical(x)* - логический вектор;
- *as.character(x)* - текстовый вектор.

Проверка типа объекта

Команды позволяют проверить, принадлежит ли объект *x* или его отдельные элементы к определенному типу:

- *is.na(x)* - отсутствующее значение;
- *is.null(x)* – ноль;
- *is.array(x)* - массив данных;
- *is.data.frame(x)* - таблица данных;
- *is.numeric(x)* - числовой вектор;
- *is.character(x)* - текстовый вектор.

Получение информации об объекте

- *length(x)* - возвращает число элементов, содержащихся в объекте *x;*
- *dim(x)* - возвращает размерность объекта *x*;
- *dimnames(x)* - возвращает или присваивает имена размерностей объекта *x*;
- *nrow(x)* - возвращает число строк в таблице или матрице *x*;
- *ncol(x)* - то же для столбцов;
- *class(x)* - возвращает или задает (например, *class(x) = "myclass")* класс объекта *x;*
- *unclass(x)* - удаляет атрибут класса у объекта *x*;
- *attr(x, which)* - возвращает или задает атрибут *which* объекта *x*;
- *attributes(obj)* - возвращает или задает список атрибутов объекта *obj*.

5.5. Извлечение данных и манипуляции с ними

- *which.max(x)* - возвращает порядковый номер элемента объекта *x* с максимальным значением;
- *which.min(x)* - возвращает порядковый номер элемента объекта *x* с минимальным значением;
- *rev(x)* - меняет порядок элементов объекта *x* на обратный;
- *sort(x)* - сортирует элементы объекта *x* по возрастанию;
- *rev(sort(x))* - сортирует элементы объекта *x* по убыванию;
- *cut(x, breaks)* - делит вектор *x* на равные интервалы, в качестве аргумента *breaks* может выступать либо необходимое число интервалов, либо вектор, - содержащий перечень точек разрыва;
- *match(x, y)* - ищет, какие значения объекта *y* совпадают со значениями обекта *x* и возвращает порядковые номера первых из совпадающих значений, например, если *y = c("a", "b", "c", "d", "e", "f")* и *x = c("d", "f")* - то результатом команды *match(x, y)* будет вектор [1] 4 6 ;
- *which(x == a)* - проверяет, какие элементы объекта *x* равны *a*, и возвращает вектор, содержащий порядковые номера этих элементов;

- *na.omit(x)* - исключает отсутствующие значения из объекта x, если x является матрицей или таблицей данных, то исключается каждая строка, содержащая хотя бы одно отсутствующее значение;
- *na.fail(x)* - возвращает сообщение об ошибке, если объект *x* содержит хотя бы одно осутствующее значение;
- *unique(x)* - если *x* - вектор или таблица данных, эта команда создаст соответствующий объект, в котором не будет повторяющихся значений;
- *table(x)* - возвращает таблицу с частотами встречаемости каждого значения *x*;
- *subset(x, ...)* - отфильтровывает и возвращает ту часть объекта *x*, которая соответствует определенному условию, например, команда *a = subset(x < 5)* - создаст вектор *a*, который будет содержать только те значения *x*, которые не превышают 5, если *x* - таблица данных, то можно использовать аргумент *select* для указания столбцов, которые необходимо извлечь из этой таблицы;
- *sample(x, size)* - производит квазислучайный отбор элементов из объекта *x* в количестве *size*; аргумент *replace = TRUE* позволяет осуществлять случайный отбор элементов с их возвратом в исходную совокупность *x;*
- *prop.table(x, margin=)* - рассчитывает маргинальные частоты таблиц; *apply(x, MARGIN, FUN = ...)* - возвращает вектор, массив или список значений, полученных путем - применения функции *FUN* к определенным элементам массива или матрицы *x*;
- *lapply(x, FUN = ...)* - возвращает список той же длины, что и *x*, при этом значения в новом списке будут результатом применения функции *FUN* к элементам исходного объекта *x*;
- *tapply(x, INDEX, FUN = ...)* - применяет функцию *FUN* к каждой совокупности значений *x*, созданной в соответствии с уровнями определенного фактора, перечень факторов указывается при помощи аргумента *INDEX*;
- *by(data, INDICES, FUN = ...)* - аналог *tapply(),* применяемый к таблицам данных;
- *merge(a, b)* - объединяет две таблицы данных (*a* и *b*) по общим столбцами или строкам;
- *aggregate(x, by = ..., FUN = ...)* - разбивает таблицу данных *x* на отдельные наборы данных, применяет к этим наборам определенную функцию *FUN* и возвращает результат в удобном для чтения формате, аргумент *by* задает список группирующих элементов (например, уровней факторов);

- *stack(x)* - преобразует данные, представленные в объекте *x* в виде отдельных столбцов, в таблицу данных (если *x* - список, то результатом будет один единственный столбец со всеми элементами *x*);
- *unstack(x)* - выполняет операцию, обратную действию функции s*tack()*;
- *reshape(x)* - преобразует таблицу данных из "широкого формата" (повторные измерения - какой-либо величины записаны в отдельных столбцах таблицы) в таблицу "узкого формата" (повторные измерения идут одно под одним в пределах одного столбца);
- *paste(..., sep = ...)* - конвертирует векторы в текстовые переменные и объединяет их в одно текстовое выражение, аргумент *sep* позволяет задать текстовое выражение, которое будет разделять значения объединяемых векторов (по умолчанию это пробел);
- *substr(x, start, stop)* - извлекает определенную часть из текстового вектора *x*, аргументы *start* и *stop* служат для указания позиции первого и последнего элемента извлекаемой части вектора *x*;
- *strsplit(x, split)* - разбивает текстовый вектор *x* в соответствии с шаблоном, заданным при помощи аргумента *split*;
- *grep(pattern, x)* - производит поиск частей текстового вектора *x*, которые совпадают с образцом, указанным при помощи аргумента *pattern*; *gsub(pattern, replacement, x)* - заменяет все части текстового вектора *x*, соответствующие шаблону - *pattern*, на выражение, заданное при помощи аргумента *replacement*;
- *tolower(x)* - преобразует все буквы текстового вектора *x* в строчные;
- *toupper(x)* - преобразует все буквы текстового вектора *x* в прописные;
- *match(x, table)* - выполняет поиск элементов в векторе *table*, которые совпадают со значениями из вектора *x*, и возвращает порядковые номера первых таких совпадений;
- *x %in% table* - команда, аналогичная предыдущей;
- *pmatch(x, table)* - выполняет поиск элементов в векторе *table*, которые частично совпадают с элементами вектора *x*, и возвращает порядковые номера первых таких совпадений;
- *nchar(x)* - возвращает количество знаков в текстовом векторе *x*.

5.6. Математические функции

Ниже приведен ряд математических функций, действие которых - должно быть понятно из их названий:

- *sin(x); cos(x); tan(x); asin(x); acos(x); atan(x); atan2(x); log(x); log10(x); exp(x);*
- *max(x)* - возвращает максимальное значение из числового вектора *x*;

- *min(x)* - возвращает минимальное значение из числового вектора *x*;
- *range(x)* - возвращает минимальное и максимальное значения из числового вектора *x*, т.е. выполняет команду *c(min(x), max(x))*;
- *sum(x)* - сумма всех элементов *x*;
- *prod(x)* - произведение всех элементов *x*;
- *mean(x)* - арифметическое среднее совокупности *x*;
- *median(x)* - медиана совокупности *x*;
- *quantile(x, probs = ...)* - рассчитывает выборочные квантили, соответствующие определенным вероятностям (по умолчанию это 0, 0.25, 0.5, 0.75, 1), при помощи - аргумента *probs* можно задать вектор с любыми вероятностями;
- *weighted.mean(x, w)* - средневзвешенное среднее по вектору *x*, аргумент *w* служит для указания весов;
- *rank(x)* - ранжирует элементы *x*;
- *var(x)* - дисперсия совокупности *x*;
- *cov(x)* - то же, что и *var()*, если *x* - матрица или таблица данных, то рассчитывается ковариационная матрица;
- *sd(x)* - стандартное отклонение совокупности *x*;
- cor(x) - возвращает корреляционную матрицу если *x* явялется матрицей или таблицей данных (результатом будет 1 если *x* является вектором);
- *var(x, y), cov(x, y)* - возвращает ковариации между *x* и *y*, или между всеми столбцами *x* и *y* - в случае, если *x* и *y* являются матрицами или таблицами данных;
- *cor(x, y)* - возвращает параметрический коэффициент корреляции Пирсона или корреляционную матрицу, если *x* и *y* являются матрицами или таблицами данных;
- *round(x, n)* - округляет *x* до *n* знаков после запятой;
- *log(x, base)* - рассчитывает логарифм *x* по основанию *base*;
- *scale(x)* - если *x* является матрицей, нормализует значения каждого столбца, т.е. вычитает от каждого значения среднее значение по столбцу и делит результат на стандартное отклонение по этому столбцу;
- *pmin(x, y)* - возвращает вектор с минимальными значениями из каждой пары *x[i], y[i]*;
- *pmax(x, y)* - команда, идентичная предыдущей, но для максимальных значений;
- *cumsum(x)* - возвращает вектор с кумулятивными суммами по вектору *x*;
- *cumprod(x)* - возвращает вектор с кумулятивными произведениями по вектору *x*;

- *cummin(x)* - возвращает вектор с кумулятивными минимумами по вектору *x*;
- *cummax(x)* - возвращает вектор с кумулятивными максимумами по вектору *x*;
- *union(x, y)* - объединяет элементы векторов *x* и *y*, результирующий вектор содержит только неповторяющиеся значения из обоих исходных векторов;
- *intersect(x, y)* - возвращает вектор только с теми значениями, которые встречаются одновременно в векторе *x* и *y*;
- *setdiff(x, y)* - возвращает вектор только с теми значениями вектора *x*, которые не встречаются в векторе *y*;
- *setequal(x, y)* - проверяет, содержат ли векторы *x* и *y* идентичные элементы (не обязательно - в одинаковых позициях), и возвращает соответствующее логическое значение;
- *Mod(x), abs(x)* - возвращают модуль *x*.

Многие математические функции принимают аргумент *na.rm = TRUE*, который позволяет игнорировать отсутствующие значения при выполнении вычислений.

Матрицы

- *t(x)* - выполняет траспонирование матрицы *x*;
- *diag(x)* - возвращает дагональ матрицы *x*;
- *%*%* - оператор умножения матриц;
- *solve(a, b)* - находит *x* в уравнении *a %*% x = b*, где *a* и *b* – матрицы;
- *solve(a)* - выполняет инверсию матрицы *a*;
- *rowsum(x)* - рассчитывает суммы по каждой строке матрицы *x* или другого - схожего по структуре объекта, более быстрой версией этой функции являетя *rowSums(x)*;
- *colsum(x), colSums(x)* - функции, аналогичные предыдущей, но работают со столбцами матриц;
- *rowMeans(x)* - рассчитытвает средние значения по каждой строке матрицы *x*;
- *colMeans(x)* - функция, аналогичная предыдущей, но работает со столбцами матриц.

Даты и время

- *as.Date(s), as.POSIXct(s)* - конвертируют вектор *s* в объект класса *Date* или *POSIXct* соответственно;
- *format(dt)* - конвертирует дату *dt* в текст; по умолчанию такой текст будет представлен - в виде “2001-02-21”, возможны и другие форматы, перечисленые ниже:

- %a, %A - сокращенное и полное названия дней
- %b, %B - сокращенное и полное названия месяцев
- %d - день месяца (01–31)
- %H - часы (00–23)
- %I - часы (01–12)
- %j - день года (001–366)
- %m - месяц (01–12)
- %M - минута (00–59)
- %p - AM/PM-представление времени суток
- %S - секунда (00–61)
- %U - неделя (00–53)
- %w - день недели (0–6, воскресенье имее позицию 0)
- %W - неделя (00–53)
- %y - год, без указания века (00–99)
- %Y - год, с указанием века

5.7. Построение графиков

- *plot(x)* - график значений вектора *x*, упорядоченных вдоль оси x;
- *plot(x, y)* - график зависимости *y* от *x*;
- *hist(x, breaks = ...)* - гистограмма частот значений переменной *x*, аргумент *breaks* можно использовать, чтобы изменить принятое по умолчанию количество столбцов;
- *barplot(x)* - столбчатая диаграмма значений вектора *x*, аргумент *horiz=TRUE* позволяет изобразить столбики горизонтально;
- *dotchart(x)* - если *x* таблица данных, то выполнение этой команды приведет к созданию диаграммы Кливленда;
- *pie(x)* - круговая диаграмма;
- boxplot(x) - график типа "ящик с усами";
- *sunflowerplot(x, y)* - то же, что и *plot(),* однако точки с одинаковыми координатами изображаются в виде "ромашек", количество лепестков у которых пропорционально количеству таких точек;
- *stripplot(x)* - изображает значения *x* вдоль единственной горизонтальной оси;
- *coplot(x˜y | z)* - график зависимости *y* от *x* для каждого интервала значений *z* (или для каждого уровня фактора, если z - фактор);
- *interaction.plot (f1, f2, y)* - если *f1* и *f2* факторы, эта функция создаст график со средними значениями *y* в соответствии со значениями *f1* (по оси *x*) и *f2* (по оси *y*);
- *matplot(x, y)* - график зависимости первого столбца значений *y* от первого столбца значений *x*, второго столбца *y* от второго столбца *x*, и т.д.;

- *fourfoldplot(x)* - изображает (в виде частей окружности) связь между двумя бинарными - переменными в разных совокупностях, объект *x* должен быть массивом размерностью *dim=c(2, 2, k)* или матрицей размером *dim=c(2, 2)* при *k = 1*;
- *assocplot(x)* - график Кохена-Френдли, который изображает то, насколько независимы значения в столбцах и строках таблицы сопряженности размером 2x2;
- *mosaicplot(x)* - мозаичный график остатков лог-линейной регрессии;
- *qqnorm(x)* - изображает квантили значений *x* против квантилей, которые можно было бы ожидать при условии, что *x* является нормально распределенной переменной;
- *qqplot(x, y)* - график зависимости квантилей *y* от квантилей *x*;
- *contour(x, y, z)* - выполняет интерполяцию данных и создает контурный график, *x* и *y* должны быть векторами, а *z* - матрицей, причем такой, что *dim(z) = c(length(x), length(y))*;
- *filled.contour(x, y, z)* - то же, что *contour()*, но заполняет области между контурами определенными цветами, легенда с расшифровкой цветов изображается автоматически;
- *image(x, y, z)* - изображает исходные данные в виде квадратов, цвет которых определяется - значениями *x* и *y*;
- *persp(x, y, z)* - то же, что и *image(),* но в виде трехмерного графика;
- *stars(x)* - если *x* матрица или таблица данных, то изображает график в виде "звезд" так, что каждая строка представлена "звездой", а столбцы задают длину - сегментов этих "звезд";
- *symbols(x, y, ...)* - изображает различные символы в соответствии с координатами, заданными *x* и *y* - (квадраты, круги, треугольники, и т.п.), размеры и цвета этих символов - задаются соответствующими аргументами функции;
- *termplot(mod.obj)* - изображает частные эффекты переменных из регрессионной модели *mod.obj*.

Многие графические функции управлются при помощи следующих аргументов:

- *add = FALSE* - если *TRUE*, то новый график будет прорисован поверх предыдущего (если он существует)
- *axes = TRUE* - если *FALSE*, оси графика и рамка вокруг него не будут нарисованы
- *type="p"* - задает тип графика: *"p"* - точки, *"l"* - линии, *"b"* - точки, соединенные линиями, *"o"* - то же, но точки представлены открытыми кружками, *"h"* - вертикальные линии, *"s"* - ступенчатые линии
- *xlim = , ylim =* - устанавливают верхний и нижний лимиты значений координатных осей, например: - *xlim=c(1, 10)* или *xlim=range(x)*

- *xlab = , ylab =* - задают названия соответствующих осей (в виде текстовых векторов)
- *main =* - главный заголовок графика (должен быть текстовым вектором)
- *sub =* - второстепенный заголовок график (изображается более мелким шрифтом)

Графические функции низкого уровня

- *points(x, y)* - добавляет к графику точки с координатами *x* и *y*, можно использовать аргумент *type = ...* (см. выше), чтобы установить определенный вид этих точек;
- *lines(x, y)* - то же, но для линий;
- *text(x, y, labels, ...)* - добавляет к графику текст, уканный при помощи аргумента *labels*, в место с координатами *x* и *y*;
- *mtext(text, side = 3, line = 0, ...)* - добавляет к графику текст, заданный аргументом *text*, сторона графика, куда будет добавлен тект, определяется параметром *side* (см. *axis()* ниже), аргумент *line* определяет, как близко к самому графику должен находиться добавляемый текст;
- *segments(x0, y0, x1, y1)* - изображает линию, проходящую из точки с координатами *(x0, y0)* в точку с координатами *(x1, y1)*;
- *arrows(x0, y0, x1, y1, angle= 30, code=2)* - то же, что и *segments()*, но для стрелок, головка стрелки будет изображена в точке с координатами *(x0, y0)*, если *code = 1*, в точке *(x1, y1)*, если *code = 2*, и с обеих сторон, если *code = 3*, аргумент *angle* - управляет тем, насколько остра головка стрелки;
- *abline(a, b)* - рисует линию, заданную уравнением $y = a + b*x$;
- *abline(h = y)* - рисует горизонтальную линию, отходящую от координаты *y*;
- *abline(v = x)* - рисует вертикальную линию, отходящую от координаты *x*;
- *abline(lm.obj)* - изображает регрессионную прямую, параметры которой хранятся в объекте *lm.mod*, который создается предварительно при помощи функции *lm()*;
- *rect(x1, y1, x2, y2)* - рисует прямоугольник, *(x1, y1)* - координата левого верхнего угла, *(x2, y2)* – правого нижнего;
- *polygon(x, y)* - рисует многоугольник, координаты которого хранятся в объектах *x* и *y*;
- *legend(x, y, legend)* - добавляет легенду в точке *(x, y)* с символами, которые задаются при помощи - аргумента *legend*;
- *title()* - добавляет к графику заголовок;
- *axis(side, vect)* - добавляет к графику ось внизу *(side = 1),* слева (*side = 2*), сверху (*side = 3*) или справа (*side = 4*);

- *rug(a)* - изображает данные из объекта *a* на оси *X* в виде небольших вертикальных линий;
- *locator(n, type = "n", ...)* - возвращает координаты *(x, y)* того места на графике, куда пользователь - кликнул мышкой *n* раз; также может нарисовать определенные символы (*type = "p")* или линии (*type = "l")* в этом месте в соотвтетствии с графическими параметрами, предварительно заданными при помощи - функции *parameters()*, по умолчанию ничего не рисуется *(type="n")*.

Управление графическими параметрами

Многие из перечисленных ниже параметров могут быть установлены глобально при помощи команды *par(...)* или локально в виде аргументов графических функций:

- *adj* - контролирует выравнивание текста (0 - по левому краю, 0.5 - по центру, 1 - по правому краю)
- *bg* - устанавливает цвет фона (например, bg = "red"), список 657 доступных цветов можно просмотреть при помощи команды ***colors()***
- *bty* - управляет типом рамки вокруг графика (доступные опции: "*o", "l", "7", "c", - "u", "]"* - рамка будет напоминать соответствующие символы), при *bty = "n"* - рамка нарисована не будет
- *cex* - управляет размером шрифта и символов, следующие параметры имеют аналогичное действие в отношении: чисел на координатных осях - *cex.axis*, меток осей - *cex.lab*, основного заголовка графика - *cex.main* и подзаголовка - *cex.sub*
- *col* - задает цвет символов и линий (в виде названий цветов *"red", "blue"* и т.п. - смотри *colors()* или в виде *"RRGGBB"* - смотрит *rgb(), hsv(), gray()* и *rainbow()*), так же, как и для *cex*, имеются варианты - *col.axis, col.lab, col.main, col.sub*
- *font* - целое число, задающее стиль текста (1: нормальный, 2: курсив, 3: жирный, 4: жирный курсив), как и для *cex*, имеются варианты *font.axis, font.lab, font.main, font.sub*
- *las* - целое число, задающее ориентацию меток осей (0: параллельно осям, 1: горизонтально, 2: перпендикулярно, 3: вертиткально)
- *lty* - управляет типом линий, может выражаться целым числом или просто соответствущим названием (1 или "solid" - сплошная; 2 или "dashed" - прерывистая; 3 или "dotted" - в виде точек, и т.д., до 6)
- *lwd* - число, задающее ширину линии (по умолчанию 1)
- *mar* - вектор из четырех чисел, определяющих пространство между осями - и границей графика, в виде c(bottom, left, top, right); - по умолчанию *mar = c(5.1, 4.1, 4.1, 2.1)*
- *mfcol* - вектор в виде *c(nr, nc)*, позволяющий разбить окно графического устройства на *nr* строк и *nc* столбцов, каждый новый график

будет потом последовательно нарисован в соответствующих ячейках получившихся столбцов
- *mfrow* - то же, но графики будут нарисованы прострочно
- *pch* - управляет типом символа, задается либо целым числом от 1 до 25, либо любым символом, заключенным в кавычки ""
- *ps* - целое число, определяющее размер (в пикселах) текста и символов
- *pty* - текстовое выражение, определяюще форму пространства, в котором будет - нарисован график, например, "s" - квадрат
- *tck* - число, определяющее длину насечек на координатных осях, при *tck* = 1 будет - нарисована координатная сетка
- *xaxt* - если *xaxt="n"*, ось *X* не будет нарисована (полезно использовать в связке с *axis(side = 1, ...)*)
- *yaxt* - то же для оси *Y* (полезно использовать в связке с *axis(side = 1, ...)*)

Lattice-графика
- *xyplot(y ~ x)* - график зависимости *y* от *x* (имеется большой набор опций)
- *barchart(y ~ x)* - столбчатая диаграмма
- *dotplot(y ~ x)* - диаграмма Кливленда
- *densityplot(~ x)* - график плотности распределения значений *x*
- *histogram(~ x)* - гистограмма значений *x*
- *bwplot(y ~ x)* - график типа "ящик с усами"
- *qqmath(~ x)* - аналог функции *qqnorm()* (см. выше)
- *stripplot(y ~ x)* - аналог функции *stripplot(x)* (см. выше)
- *qq(y ~ x)* - изображает квантили распределений *x* и *y* для визуального сравнения этих распределений, переменная *x* должна - быть числовой, переменная *y* - числовой, текстовой или фактором с двумя уровнями
- *levelplot(z ~ x*y | g1*g2)* - цветной график значений *z*, координаты которых заданы - переменными *x* и *y* (очевидно, что *x, y* и *z* должны иметь - одинаковую длину); *g1, g2...* (если присутствуют) - факторы или числовые переменные, чьи значения автоматически разбиваются на равномерные отрезки
- *wireframe(z ~ x*y | g1*g2)* - функция для построения трехмерных диаграмм рассеяния - и плоскостей, *z, x* и *y* - числовые векторы, *g1, g2...* - (если присутствуют) - факторы или числовые переменные, чьи значения автоматически разбиваются на равномерные отрезки
- *cloud(z ~ x*y | g1*g2)* - трехмерная диаграмма рассеяния

5.8. Статистические модели

- *nlm(f,p)* - минимизирует функцию *f*, используя алгоритм типа ньютоновского, *p* - вектор с исходными значениями параметров функции;
- *lm(formula)* - рассчитывает линейную регрессионную модель, формула модели *formula* обычно - представлена в виде *y ~ x1 + x2 + ...*, - где *y* - переменная-отклик, *x1, x2, ...* – предикторы;
- *glm(formula, family = ...)* - рассчитывает логистическую регрессионную модель, аргумент *family* служит для указания функции распределения остатков модели;
- *nls(formula)* - рассчитывает параметры нелинейной регрессионной модели по методу наименьших квадратов;
- *spline(x, y)* - выполняет интерполяцию точек с координатами *(x, y)* кубическим сплайном;
- *loess(formula)* - используя локальное сглаживание, выполняет подгонку полиномиальной плоскости к массиву точек, заданной одним - или несколькими предикторами.

Многие функции, рассчитывающие статистические модели, имеют - общие аргументы, в частности:

- *data = ...* - таблица данных, в которой хранятся переменные, указанные в - формуле функции
- *subset = ...* - указывает, какие именно значения переменных должны принимать - участие при расчете модели
- *na.action =* - указывает, как следует поступать с отсутствующими значениями; - например, для игнорирования таких значений применяется - *na.action = "na.omit"*

Для рассчитанных моделей часто применимы следующие функции - общего назначения:

- *predict(fit, ...)* - возвращает либо те значения, которые модель *fit* предсказывает на основе исходных данных, либо те значения, которые модель предсказывает для новых, заданных пользователем данных
- *df.residual(fit)* - возвращает число степеней свободы для остатков модели
- *coef(fit)* - возвращает рассчитанные коэффициенты модели (в ряде случаев, вместе с их стандартными ошибками)
- *residuals(fit)* - возвращает остатки модели
- *deviance(fit)* - возвращает девиату модели
- *fitted(fit)* - возвращает те значения, которые модель *fit* предсказывает на основе исходных данных

- *AIC(fit)* - возвращает информационный критерий Акайке для модели *fit*.

УПРАЖНЕНИЯ

Упражнение 1.

В каком из этих случаев будет получено значение *TRUE*:

a) isTRUE(!TRUE);
b) isTRUE(3);
c) isTRUE(5 != NA);
d) !isTRUE(4 < 3);
e) !isTRUE(8 != 5)

Упражнение 2.

В каком из этих случаев функция *identical()* не возвратит значение *TRUE*:

a) identical(sin(0),0)
b) identical(1 + 5, 6)
c) identical(paste("a","b"),"a b")
d) identical(NA,NaN)
e) identical(-21/0,-Inf)

Упражнение 3.

Вычислите значения функции *xor()*:

a) xor(4 >= 9, 8 != 8.0)
b) xor(!isTRUE(TRUE), 6 > -1)
c) xor(identical(xor, 'xor'), 7 == 7.0)
d) xor(isTRUE(5 != NA),identical("Анна","анна"))
e) xor(TRUE && c(TRUE, FALSE, TRUE,TRUE),FALSE)

Упражнение 4.

Файл *fuel.txt* является одним из нескольких файлов, которые функция *datafile()* из пакета *DAAG* размещает в рабочем каталоге. Для анализа файла, размещенного в рабочем каталоге можно использовать функцию *file.show()*. Исследуйте файл *fuel.txt* указанным способом. В качестве альтернативы используйте для анализа функцию *read.table()*.

Разместите файл *molclock1.txt* в рабочем директории. Определите среднее значение *AvRate*.

Упражнение 5.

Для каждого вида деревьев тропического леса в наборе данных *rainforest* из пакета *DAAG* подсчитать: (1) количество строк в которых значения для столбца *root* заполнены и не заполнены (содержат *NA*), (2) количество строк, в которых заполнены все столбцы.

Упражнение 6.

Для каждого столбца фрейма данных *Pima.tr2* из пакета *MASS* определить число незаполненных значений.

Упражнение 7.

Определите структуру наборов данных *tinting*, *possum* и *possumsites* из пакета *DAAG*.

Упражнение 8.

Определите среднее и диапазон значений для столбцов *height* и *weight* фрейма *women* (пакет *datasets*, который содержит указанный фрейм, загружается по умолчанию при старте *RStudio*).

Упражнение 9.

Определите среднее и диапазон значений для 100 случайных чисел нормального распределения со средним значением 0 и дисперсией 1. Повторите операцию несколько раз, каждый раз генерируя новый набор из 100 случайных чисел.

Упражнение 10.

Выделите все строки набора данных *rainforest* из пакета *DAAG* для всех типов деревьев за исключением *C.fraseri*.

Упражнение 11.

Извлеките следующие подмножества фрейма данных *ais* из пакета DAAG:

(a) данные для гребцов (*rowers*);

(b) данные для гребцов (*rowers*) и теннисистов (*tennis players*);

(c) данные для женских баскетбольных команд (*female basketabllers*) и гребцов (*rowers*).

Упражнение 12.

Используя данные о дереве *Acmena* из фрейма данных *reinforest* из пакета *DAAG*, постройте график зависимости *wood* (древесной биомассы) от *dbh* (диаметра дерева на высоте груди), используя как нетрансформированные шкалы, так и логарифмический масштаб.

Запишите уравнение, описывающее связь между *wood* и *dbh*.

Упражнение 13.

Фрейм данных *orings* из пакета *DAAG* предоставляет данные об авариях, которые произошли при запусках космических кораблей до катастрофического запуска «Челленджера» 28 января 1986 года. В диаграммы, используемые при принятии решения о продолжении запуска, перед запуском были включены только наблюдения из строк 1, 2, 4, 11, 13 и 18.

Добавьте новый столбец во фрейм *orings*, который идентифицирует строки, которые были включены в диаграммы перед запуском и постройте три графика зависимости *Total* и *Temperature*:

(a) используйте только те строки, которые были включены в графики перед запуском;

(b) используйте все строки;

(c) используйте все строки, выбирая разные символы или цвета, чтобы указать, были ли включены точки в диаграммы перед запуском.

Упражнение 14.

Используя фрейм данных *oddbooks* из пакета *DAAG*, исследуйте с помощью диаграмм взаимосвязи между:

(a) весом и объемом;

(b) плотностью и объемом;

(c) плотностью и площадью страницы.

Упражнение 15.

Изучите справочный материал для функций *dotplot()* и *stripplot()*. Сравните между собой графики и прокомментируйте отличия (фрейм данных *ant111b* из пакета *DAAG*):

```
with(ant111b, stripchart(harvwt ~ site))  # базовый график
library(lattice)
dotplot(site ~ harvwt, data=ant111b)
dotplot(harvwt ~ site, data=ant111b)
stripplot(harvwt ~ site, data=ant111b)
stripplot(site ~ harvwt, data=ant111b)
```

Упражнение 16.

Проверьте класс каждого из столбцов фрейма данных *cabbages* из пакета *MASS*. Постройте расположенные рядом графики зависимости *HeadWt* от *Date* для каждого из уровней *Cult*.

Найдите минимальное, среднее и максимальное значение *VitC* для каждого уровня *Cult* и добавьте их на построенные графики.

Упражнение 17.

Для фрейма данных *nsw74psid3* из пакета *DAAG* используйте функцию *stripplot()* для сравнения переменных *age* и *educ* на одном графике.

Упражнение 18.

Отсортируйте строки *Acmena smithii* во фрейме данных *rainforest* из пакета *DAAG* в порядке возрастания значений *dbh*.

Упражнение 19.

Создайте цикл *for*, который для заданного числового вектора выведет одно число на строку, вместе с квадратом и кубом этого числа.

Используте цикл *while* для достижения того же результата.

Покажите, как достичь того же результата без использования явного цикла.

Упражнение 20.

Следующая функция вычисляет среднее и стандартное отклонение числового вектора *x*:

```
meanANDsd <- function(x){
                av <- mean(x)
                sdev <- sd(x)
                c(mean=av, sd = sdev) # Функция возвращает этот вектор
                }
```

Измените функцию так, чтобы:

(a) по умолчанию использовалась функция *rnorm()* для генерации 20 случайных нормальных числа и возвращалось стандартное отклонение;

(b) если в *x* есть отсутствующие значения (*NA*), то среднее и стандартное отклонение рассчитывается для остальных значений.

Упражнение 21.

Функции, которые могут быть использованы для получения информации о фреймах данных, включают *str(), dim(), row.names()* и *names()*. Попробуйте каждую из этих функций для фреймов данных *allbacks, ant111b* и *tinting* (все из пакета *DAAG*).

Получите информацию о каждом столбце указанных фреймов данных, используя функцию *sapply()*.

Для столбцов фрейма tinting, которые являются факторами, используйте *table()*, чтобы табулировать число значений для каждого уровня.

Упражнение 22.

Составьте список наблюдений в каждом из различных районов во фрейме данных *rockArt* из пакета *DAAGxtras*. Создайте группу факторов, в

которой все районы с менее чем 5 наблюдениями сгруппированы вместе в категорию *other*.

Упражнение 23.

Изучая повторяющиеся случайные выборки из нормального распределения, можно получить представление о влиянии вариации выборки на распределение выборки.

Получить случайную выборку из 100 значений из нормального распределения (со средним значением 0 и стандартным отклонением 1), построить гистограмму распределения и наложить график плотности можно таким образом:

```
y <- rnorm(100)
hist(y, probability=TRUE)
lines(density(y))
```

Повторите несколько раз, вместо 100 значений выборки:

(a) возьмите 5 образцов размером 25, и постройте гистограмму с наложенным графиком плотности;

(b) возьмите 5 образцов размером 100, и постройте гистограмму с наложенным графиком плотности;

(c) возьмите 5 образцов размером 500, и постройте гистограмму с наложенным графиком плотности;

(d) возьмите 5 образцов размером 2000, и постройте гистограмму с наложенным графиком плотности.

Все 20 графиков должны быть расположены на одной странице. (Подсказка: используйте *par(mfrow = c(...))*, чтобы сгруппировать графики более точно, выполните настройки *par(mar = c (...), mgp = c (...)))*.

Упражнение 24.

Благодаря прямому интернет-подключению файлы могут быть прочитаны непосредственно с веб-страницы, например:

```
> webfolder <- "http://www.maths.anu.edu.au/~johnm/datasets/text/"
> webpage <- paste(webfolder, "molclock.txt", sep="")
> molclock <- read.table(url(webpage))
```

Используйте этот подход для ввода файла *travelbooks.txt*, который доступен на этой же веб-странице. Сколько книг, перечисленных в файле *travelbooks.txt* относятся к типу *RoadMaps*.

Упражнение 25.

Прочитайте данные из файла *bostonc.txt* в рабочий директорий. Для этого используйте функцию *datafile()* из пакета *DAAG*. Внимательно изучите содержимое файла. Загрузите в память только таблицу, размещенную в этом файле.

При использовании функции *read.table()*, нужно будет правильно подобрать значения параметров *sep, comment.char* и *skip*, задаваемых по умолчанию. Обратите внимание, что *\t* обозначает символ табуляции.

Упражнение 26.

Используйте функцию *read.csv()* для чтения файла *crx.data*, который доступен на веб-странице *http://mlearn.ics.uci.edu/databases/credit-screening*/. Ознакомьтесь с данными пункта 7 и 8 файла *crx.names*, чтобы узнать, какие столбцы в файле числовые, какие факторные (пункт 7). Убедитесь, что количество пропущенных значений в каждом столбце равно числу, указанному в пунке 8 файла *crx.names*.

Упражнение 27.

Напишите программу, которая обрабатывает произвольный числовой вектор и последовательно выводит на экран в каждой строке: значение компоненты вектора и рядом – значение компоненты вектора в квадрате. Кроме того, программа должна вычислить и вывести на экран среднее арифметическое и среднее геометрическое значение компонент заданного вектора.

Напишите программу с использованием оператора *for*, оператора *while* и без использования операторов цикла. Оцените и сравните производительность каждой реализации программы для векторов большой размерности.

Упражнение 28.

Исследуйте данные об алмазах, представленные в таблице *diamonds* из пакета *ggplot2*. Отыщите в таблице самый большой (по объему) и самый маленький алмаз. Укажите их стоимость.

Упражнение 29.

Используя фрейм *diamonds* из пакета *ggplot2*, определите все возможные сочетания качества огранки (*cut*), цвета (*color*) и чистоты (*clarity*) алмазов. Результат представьте в таблице со столбцами: № п/п, cut, color, clarity.

Упражнение 30.

Постройте функцию, вычисляющую значение *y = 1/(x+1)*sqrt(x)*. Используя функцию *integrate()*, найдите интеграл от функции *y* в интервале от *0* до *Inf*.

ИСПОЛЬЗОВАННЫЕ ИСТОЧНИКИ

1. http://cran.r-project.org/web /packages/
2. The Comprehensive R Archive Network (CRAN) - http://cran.r-project.org
3. http://dirk.eddelbuettel.com/cranberries/
4. RStudio—an open-source Integrated Development Environment (IDE) https://www.rstudio.com/
5. Wickham H. Advanced R, CRC Press, Taylor & Francis Group, 2015, p. 456.
6. The Art of R Programming http://heather.cs.ucdavis.edu/~matloff/132/NSPpart.pdf
7. http://plants.usda.gov/adv_search.html
8. https://github.com/swirldev/swirl_courses/blob/master/R_Programming_Alt/Looking_at_Data/plant-data.txt
9. Чистяков С. Программируем на языке R: как правильно писать циклы для обработки больших объемов данных, https://xakep.ru/2016/11/22/
10. https://www.datacamp.com/community/tutorials/r-tutorial-apply-family#gs.9hRldQg
11. http://r.psylab.info/blog/2015/04/23/time-measure

Филиппов Феликс Васильевич

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ R

ПРАКТИКУМ