Федеральное государственное образовательное бюджетное
учреждение высшего образования
«ФИНАНСОВЫЙ УНИВЕРСИТЕТ
ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»
(Финансовый университет)

Департамент анализа данных, принятия решений
и финансовых технологий

С.А. Зададаев

МАТЕМАТИКА НА ЯЗЫКЕ R

Учебник

Издательство Прометей

МОСКВА
2018

УДК 519.85
ББК 32.973
З-15

Рецензенты:

В.И. Соловьев, д.э.н., профессор, руководитель ДАДПРиФТ Финансового университета при Правительстве РФ

Д.В. Сошников, к.ф.-м.н., доцент НИУ ВШЭ, координатор академических программ Департамента стратегических технологий Майкрософт Россия

Зададаев, Сергей Алексеевич.

З-15 Математика на языке R: учебник / Финансовый университет при Правительстве РФ. – Москва: Прометей, 2018. – 324 с. (Учебники для вузов. Специальная литература)

ISBN 978-5-907003-59-0

Структурно учебник представляет собой 17 компьютерных практикумов по изучению и применению вычислительных возможностей языка R в решении базовых задач математического анализа и линейной алгебры и календарно соответствует программе дисциплины «Компьютерный практикум», читаемой в Финансовом университете при Правительстве РФ на первом курсе общеэкономических специальностей.

Содержательно в учебнике последовательно излагаются основы языка программирования R с постепенным углублением по мере продвижения по осваиваемым навыкам в применении к высшей математике первого курса. В конце учебника приведен глоссарий по операторам и библиотекам R для удобства последующего использования его в качестве справочного руководства по R. Для комфортного программирования на R практикумы ориентированы на популярную оболочку RStudio.

Учебник будет полезен всем студентам первых курсов, изучающих математический анализ и линейную алгебру, которые стремятся знать самые современные вычислительные технологии, а также тем, кто хочет научиться программировать на языке R и продолжать изучать его применение в статистическом анализе и анализе данных.

Учебник может быть интересен аспирантам, научным сотрудникам и преподавателям.

ISBN 978-5-907003-59-0

Оглавление

Вместо предисловия

Прежде, чем приступить к изучению и освоению материала этого учебника, важно правильно представлять себе специфику современного прикладного программирования. А она заключается в том, что нет ничего более быстро меняющегося, чем информационные технологии.

В процессе изучения вам самим наверняка захочется написать более совершенную функцию или процедуру, чем в обсуждаемом стандарте R. Сложно быстро изменить производственные технологии, но легко – программные.

Так что, приготовьтесь к постоянным усовершенствованиям пакета R, его библиотек и оболочки RStudio. Маловероятно, но возможно, что некоторые операции вообще не будут работать из-за их кардинального обновления. Это совершенная норма настоящего времени, и если вы хотите быть на олимпе вычислительных и визуализирующих возможностей, то решительно подключайте в свои союзники internet с его российскими и англоязычными ресурсами профессионалов, обсуждающих и творящих на языке R по всему миру.

Сама по себе семантика языка R достаточно проста для понимания и использования, и все-таки советуем на первых порах набирать программные коды вручную, а не копировать их из текста, если вы используете электронный вариант учебника. Такие коды выделены в тексте так:

```
# Здравствуй, мир! :)
# Я сделаю тебя лучше! ))
```

На сайте автора www.zadadaev.com есть форум и обратная связь. Всем удачи!

Компетенции дисциплины

В современной парадигме российского высшего образования принято указывать развиваемые той или иной дисциплиной профессиональные компетенции. Следуя этой традиции, кратко укажем: за что именно мы будем бороться при изучении материала данного учебника.

	Компетенция	**Знания, умения, владения**
1.	способность применять математические методы для решения стандартных теоретических и рикладных задач, интерпретировать полученные математические результаты	**Знать** вычислительные методики основных задач математического анализа и линейной алгебры; **Уметь** использовать компьютерные технологии при реализации математических методов и моделей для описания и анализа прикладных задач. **Владеть** навыками вычислительной работы в R и коммуникации с Excel.
2.	способность оформлять аналитические и отчетные материалы по результатам выполненной работы	**Знать** основные средства визуализации количественных данных в R **Уметь** использовать компьютерные технологии представления данных и графической визуализации результатов применения

	Компетенция	Знания, умения, владения
		математических методов и моделей для описания и анализа различных прикладных задач **Владеть** навыками работы в RStudio в части визуализации количественных данных.

Инструкция по установке пакетов R и RStudio

Пропустите этот раздел, если R и RSudio уже установлены на компьютере.

Для установки R и RStudio под операционную систему Windows воспользуйтесь следующими ниже шагами.

Необходимо:

1. Включить интернет на все время установочных действий.

2. По ссылке https://cran.r-project.org/ скачать загрузчик R «R-3.4.2-win» (или более новой версии).

3. Запустить сохраненный загрузчик R, приняв все его рекомендации и умолчания, дождаться окончания установки языка R.

4. Для Windows 7 (или ниже, например, XP) прописать путь к R в системе: ссылка примерно такая: Панель управления –> Система –> Дополнительные параметры системы –> Переменные среды –> В системных переменных выбрать переменную «path» и нажать «изменить» –> в конец строки «path» через разделитель «;» дописать путь к R (обычно это C:\Program Files\R – так устанавливается по умолчанию) –> нажать ок.

5. для Windows 10 прописывать путь к папке R не требуется.

6. По ссылке https://www.rstudio.com/products/rstudio/download/ скачать соответствующий вашей

операционной системе загрузчик RStudio (бесплатный вариант RStudio Desktop FREE) и запустить его.

Для установки под другие операционные системы действуйте аналогичным образом и помните: R не любит кириллицу в названии папок, категорически не переносит несколько точек в названии файлов и в ряде случаев отказывается грузить библиотеки, если за компьютером находится пользователь с ограниченными правами на текущую сессию.

Замечание

Часто для установки ряда библиотек, например, важной библиотеки library("xlsx") для коммуникаций R с Excel, на компьютер требуется установить Java. Это можно сделать, пройдя последовательно по следующим ниже ссылкам и приняв все умолчания:

https://java.com/ru/download/

и

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Не забудьте выбрать соответствующую операционную систему. После загрузки Java можно обычным образом загрузить пакет «xlsx», подробное описание которого приведено в последующих частях практикума.

Практикум 1.
Введение в R (RStudio)

Требования

Для работы с текущими и последующими учебно-методическими материалами по R необходимо иметь установленные на компьютере два пакета: интерпретатор языка программирования R и оболочку RStudio. Процедура официального бесплатного скачивания и установки этих пакетов была описана в инструкции выше.

Настоятельно советуем установить эти два пакета на своем личном компьютере (ноутбуке, планшете,...), т.к. вместе с десятками тысяч других специалистов по всему миру считаем это вычислительное средство одним из первых в линейке лидеров на данный момент времени.

Запуск RStudio

Для работы с R будем использовать удобную оболочку RStudio, которая уже сама будет осуществлять взаимодействие с интерпретатором языка R. Запустим RStudio, щелкнув по соответствующему ярлыку (см.рис. 1).

Замечание. Если текст картинок трудно читаем, используйте экранное увеличение с помощью вращения колесика мыши при нажатой клавише Ctrl (для учебника в электронном виде).

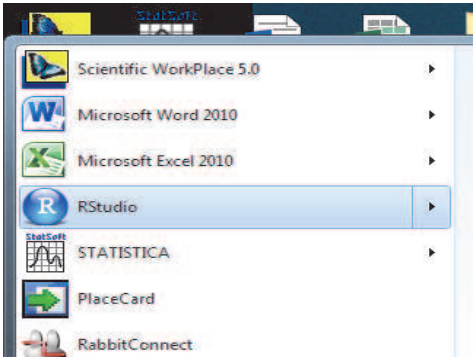


Рисунок 1

В результате мы увидим раскрывшееся окно программы. Создадим новый документ с будущим кодом (скриптом), выбрав мышью в верхнем левом пункте меню «R script» или нажав сочетание клавиш Ctrl+Shift+N (см. рис. 2):

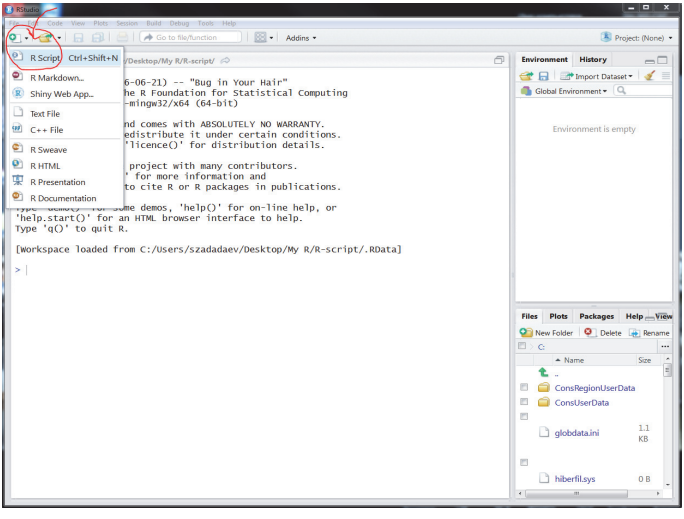


Рисунок 2

После этого область RStudio разобьётся на четыре окна и примет законченный вид (см. рис. 3):

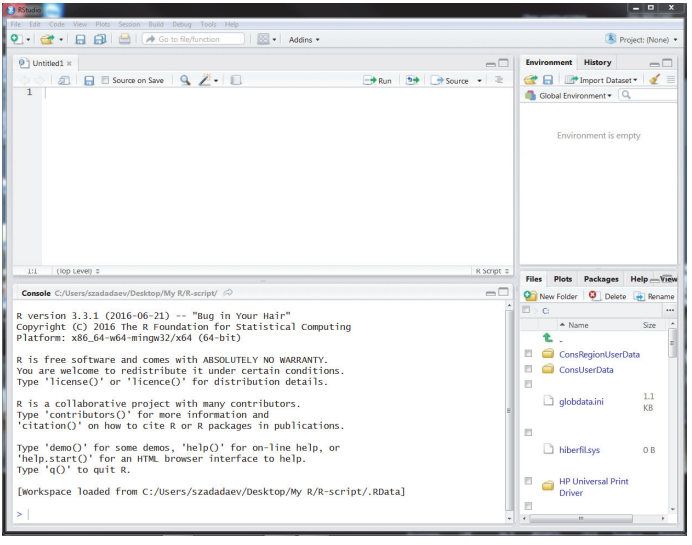


Рисунок 3

1. Левое верхнее: окно рабочего листа с будущим кодом на языке R. Здесь мы будем вводить команды R и их запускать.

2. Левое нижнее: консоль R – окно с пошаговой компиляцией команд R. На этом листе мы будем видеть результаты работы наших программ.

3. Правое верхнее: окно истории работы R и текущих значений объектов.

4. Правое нижнее: многофункциональное окно: навигация / графика / пакеты / справка...

Оператор комментария

Как шутят программисты: оператор комментария – самый частый оператор в любом языке программирования. Именно оставленные нами пояснения

делают текст программ в каком-то смысле интерактивным: ремарки напоминают нам о смысле введенных ранее команд.

В языке R таким оператором является символ решетки (хэш): #, т.е. компилятор R не будет воспринимать в качестве кода все, что написано после символа решетки «#» в текущей строке. Например, строка

```
# Здравствуй, мир! :)
```

послужит прекрасным заголовком, но само приветствие исполнено в R не будет.

Загрузка библиотек

Сейчас мы могли бы уже написать что-нибудь более содержательное в поле текста программы (левое верхнее окно), однако при первом запуске RStudio нам необходимо подгрузить из интернета актуальные библиотеки используемых процедур и функций в R. Это можно сделать двумя способами.

Первый способ: набрать специальную команду загрузки в R требуемого пакета. Например, желательной первичной загрузкой основной базы программных пакетов в R является загрузка библиотеки «Rcmdr». Введем в первой строке рабочего листа команду (можно просто скопировать текст через буфер обмена):

```
install.packages("Rcmdr")    # Загрузить из репозитория R
  # пакет "Rcmdr"
```

Важно знать, что в языке R различаются заглавные и прописные буквы, то есть символы «a» и «A» – разные! Здесь Rcmdr не равно rcmdr!

Если теперь после набора этой строки нажать Enter, то курсор перескочит на новую строку и ничего более не произойдет. Но если нажать сочетание Ctrl+Enter, то курсор также перескочит на новую строку, но код предыдущей строки при этом будет на-

правлен на компиляцию, что приведет к выполнению указанной команды. Можно будет заметить, как R подгружает из интернета необходимые модули, это занимает некоторое время. В итоге мы увидим сообщение об успешной установке (см. рис. 4):

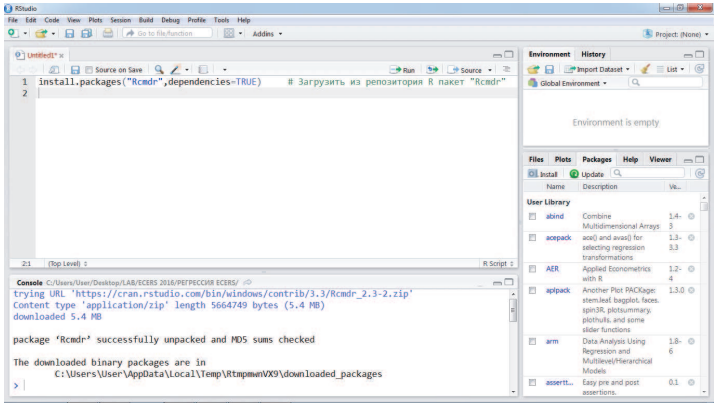


Рисунок 4

Замечание. На рисунке можно заметить, что команда install.packages выводится с параметром dependencies = TRUE, однако, с версии 3.4.2 данный параметр перестал быть актуальным. Здесь и далее его указывать не надо.

Такую установку требуется делать один раз для каждого компьютера. Если используются дополнительные библиотеки, то поступают аналогично.

Второй способ: вызвать интерактивное окно загрузчика библиотек, пройдя по меню Tools –> Install Packages... (см. рис. 5):

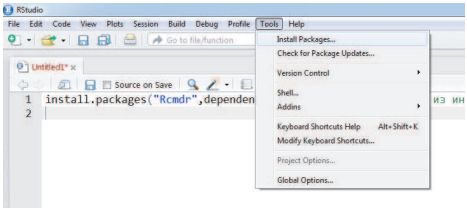


Рисунок 5

Замечание. На рисунке пункт меню Install Packages... находится в самом верху окна, однако, с версии старше 3.4.2 этот пункт будет уже вторым. Вообще говоря, надо привыкнуть к постоянным модификациям и улучшениям R и RStudio. Например, за год написания этого практикума R и RStudio были обновлены 4 раза. Такая частая модификация позволяет этому вычислительному средству оставаться чрезвычайно актуальным. Если после установки какого-либо пакета R выдаст сообщение, что пакет был собран под более свежую версию R – это не является ошибкой, но является намеком, что пора обновить пакет R и RStudio.

В появившемся окне остается только ввести необходимые имена пакетов (если несколько сразу, то через пробел) и нажать кнопку Install (см. рис. 6):

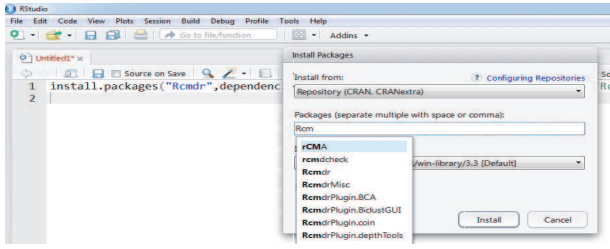


Рисунок 6

Однако, даже когда библиотека загружена на компьютер одним из описанных способов, обращение к ее процедурам и функциям в текущем сеансе (в текущем запуске RStudio) остается недоступным, пока мы не подключим, не активизируем в текущей сессии эту библиотеку командой library(name). Это сделано в R для экономии оперативной памяти.

Например, мы хотим использовать расширенные возможности алгебры матриц с помощью библиотеки «Matrix». Тогда мы должны один раз загрузить этот пакет на компьютер командой:

```
install.packages("Matrix")    # Загрузить из репозитория R
  # пакет "Matrix"
```

и всякий раз, когда собираемся использовать этот пакет, нам необходимо в начале сессии запускать команду активации:

```
library(Matrix)   # Активизировать загруженный в R
  # пакет "Matrix"
```

Задание 1. В R есть встроенные данные о зависимости скорости автомобилей от тормозного пути (исследование Ford, 1920 год). Эти данные хранятся в зарезервированной переменной под именем «cars». Для того, чтобы понять, как хранятся данные о скоростях и дистанциях тормозных путей в cars требуется запустить команду glimpse(cars), которая входит в библиотеку dplyr.

Решение. Фактически нам необходимо сделать три небольшие операции:

1. Загрузить на компьютер библиотеку dplyr (это очень удобная и популярная библиотека для манипуляций с различными данными):

```
install.packages("dplyr")    # Загрузить из репозитория R
  # пакет "dplyr"
```

2. Активизировать эту библиотеку в текущей сессии:

```
library(dplyr)   # Активация библиотеки "dplyr"
```

3. Вызвать соответствующую функцию из пакета:

```
glimpse(cars)   # Вызов функции glimpse, которая описы-
  # вает структуру данных
```

Далее мы выделяем эти три набранные строки левой клавишей мыши и запускаем код на компиляцию, щелкнув мышкой по клавише Run (или нажав Ctrl+Enter) (см. рис. 7).

В результате в окне консоли получим полную информацию о переменной cars. Мы вернемся к этому отчету несколько позже.

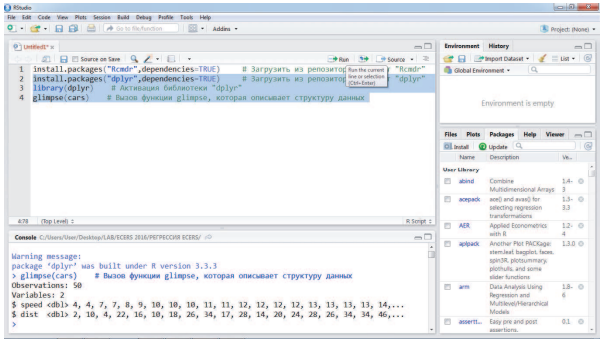


Рисунок 7

R – калькулятор

Перечислим основные математические функции, набрав которые в качестве кода, мы сразу получим вычисленные значения:

```
sin(2); cos(2); tan(2); asin(0.5); acos(0.5); atan(2); log(0.3);
log(1024, 2); exp(2); log10(4); sinh(2); cosh(2); tanh(2);
asinh(2); acosh(2); atanh(1/2);
```

Замечание. Здесь и далее условно в качестве аргументов функций взяты произвольные числа.

Обратим особое внимание, что в R принято использовать точку в качестве десятичного разделителя (в Excel, кстати, более заумно: в ячейках используют запятую, а в макросах – точку).

Расшифруем некоторые из приведенных функций:

```
exp(2)          # Экспонента от 2, e^2
log(1024, 2)    # Логарифм 1024 по основанию
2,              # log_2 1024
log(0.3)        # Натуральный логарифм числа 0.3,
                # ln 0.3
abs(-5)         # Модуль от -5, |–5|
```

```
atan2(0,-3)              # Угол между осью ox и вектором
  # (-3,0) / здесь x,y наоборот!
2**3; 2^3                # Возведение в степень 2 в 3, 2^3 /
  # возможны оба варианта
sqrt(4)                  # Арифметический корень из 4, √4
factorial(5)             # Факториал числа 5, 5!=1*2*3*4*5
choose(5, 3)             # Число сочетаний 3 из 5: C(5,3)
pi                       # Число π
exp(1)                   # Число e
```

Полезны также функции округления (представление чисел):

```
round(1.0023857, 6)      # Округляет число 1.0023857
  # до 6 знака после запятой включ.
signif(1.0023857, 6)     # Округляет число 1.0023857
  # до 6 значащих цифр включ.
trunc(-3.999)            # Отсечение дробной части -3.999
ceiling(-4.2)            # Округление до целого в мень-
  # шую сторону (по модулю)
floor(-4.2)              # Округление до целого в боль-
  # шую сторону (по модулю)
```

Задание 2. Вычислить с точностью не более четвертой цифры после запятой выражение

$$\frac{\ln^{-2}(\mathrm{tg}^2(\sqrt[3]{30.231}))}{\sin^{-1}(6!+|3.15-e^{4.002}|)}.$$

Решение. Разумеется, нам не следует сразу же бросаться набирать этого «крокодила» в строке. Надежнее всего разбить данное выражение на небольшие части, сохранить промежуточные вычисления в новых переменных и потом уже образовывать ответ.

Но для сохранения промежуточных ответов нам понадобится оператор присваивания, состоящий из двух символов, напоминающих стрелку влево:

<-

или напоминающий стрелку направо

->

в зависимости от направления от кого кому происходит присваивание (см. также рис. 8):

```
a <- tan(30.231^(1/3))^2                    # Аргумент лога-
  # рифма помещен в переменную a
b <-  log(a)^(-2)                           # Числитель поме-
  # щен в переменную b
d <- factorial(6) + abs(3.15 - exp(4.002))  # Аргумент
  # синуса помещен в переменную d
s <- sin(d)^(-1)                            # Знаменатель
  # помещен в переменную s
x <- b/s                                    # Значение всего
  # выражения помещено в переменную x
round(x, 4)                    # Ответ: округленное значение
```

Замечание. Всегда отделяйте оператор присваивания пробелом спереди и сзади.

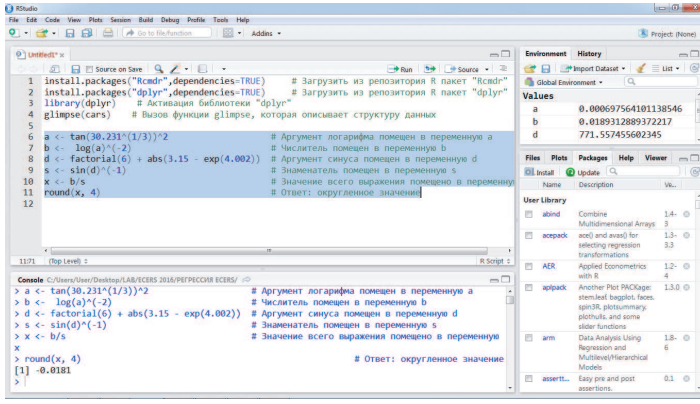


Рисунок 8

Здесь мы образовали новые переменные a,b,d,s и x, где в качестве символа присваивания использовали конструкцию, напоминающую стрелку: «<-». Не было бы ошибкой использовать и обычный знак равенства, но тем, кто хочет демонстрировать загра-

ничной публике высший класс следует использовать в этом контексте именно сочетание «<-» или в другую сторону «->» для случая, когда переменная находится справа от присваиваемого выражения. Вообще, в языке R есть достаточно строгое различие где следует употреблять оператор присваивания ->, где знак =, а где двойное равно ==, но об этом позже.

Обратите внимание, что значения наших промежуточных переменных отобразились в правом верхнем окне окружения. Если значений так много, что они не помещаются в данное окно, то узнать значение интересуемой переменной можно и по-другому. Как вариант просто написать ее на свободной строке и нажать Ctrl+Entr:

```
x                  # Вывести на экран консоли значение x
print(x)           # То же самое, но надежнее
```

либо указать ее через точку с запятой с основным выражением:

```
x <- b/s; x        # Значение всего выражения помещено
  # в переменную x
```

либо выделить мышкой именно эту переменную в тексте кода и также отправить на компиляцию Ctrl+Entr (или кликнуть по кнопке Run):

Но чем же оператор print(x) надежнее простого указания x? Ответ на этот вопрос дает следующий код с использованием объединяющих команды в одну группу операторных скобок {...}:

```
x <- 2; y <- 3          # Присваиваем переменным x и y
  # значения 2 и 3 соответственно
{x; y}                  # В этом объединении будет выведен
  # на консоль только последний "y"
{print(x); print(y)}    # В этом будут отображены обе пере-
  # менные
```

с результатом в консольном окне (левое нижнее) (рис. 9):

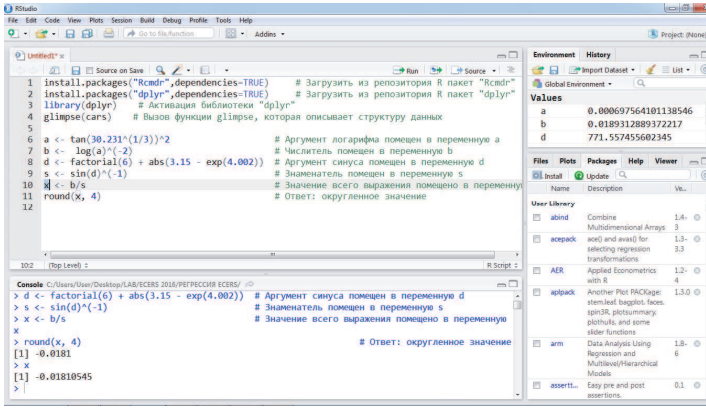


Рисунок 9

```
> x <- 2
> y <- 3
> {x; y}
[1] 3
> {print(x); print(y)}
[1] 2
[1] 3
```

Выводимая точность вычислений

Помимо округления числовых результатов может встать обратная необходимость в повышении точности. Следующая команда позволяет повысить выводимую в консоль точность *double*-арифметики до 22 используемых цифр/разрядов в записи числа (параметр *digits*):

```
options(digits=12)     # Установка максимального коли
  # чества используемых цифр на уровне 12
```

Для получения еще большей точности следует использовать специализированные пакеты, например, пакет «Rmpfr», справка по которому станет доступна из RStudio после стандартной установки пакета из

репозитория. Ближе к концу учебника мы подробно рассмотрим особое алгебраическое расширение double-арифметики.

Массивы чисел в R

Переменным в R можно присваивать не только отдельно взятые числа, но и массивы чисел. Самым простым примером может служить набор целых чисел от 0 до 10. Мы можем весь этот набор сохранить в массиве командой

```
m <- 0:10                    # Массив чисел 0,1,2,...,10
m                            # Вывести на экран содержимое m
```

Если теперь отправить этот код на компиляцию (Ctrl +Enter), то получим под именем «m» целый набор чисел. В случае, когда мы хотим обратиться к конкретному элементу этого массива, нам необходимо использовать оператор квадратных скобок [...]. Например, чтобы узнать 6-ое значение в массиве «m» надо написать:

```
m [6]          # 6-ой элемент массива m
```

а выражение

```
m [2:7]        # подмассив значений m с номерами от 2 до 7
```

образует подмножество значений массива m с номерами 2, 3, ... , 7, т.е. фактически новый массив m[2], m[3],..., m[7] (см. рис. 10).

Еще одним примером массива чисел может служить рассмотренный в задании №1 объект cars с данными о скоростях и тормозных путях. Напомним, что отчет в окне консоли о структуре объекта cars по функции glimpse(cars) имел вид:

```
Observations: 50
Variables: 2
```

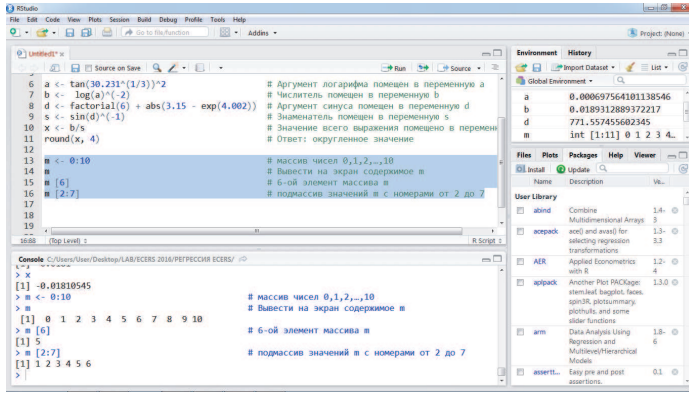


Рисунок 10

```
$ speed <dbl> 4, 4, 7, 7, 8, 9, 10, 10, 10, 11, 11,
12, 12, 12, 12, 13, 13, 13, 13, 14,...
$ dist  <dbl> 2, 10, 4, 22, 16, 10, 18, 26, 34, 17,
28, 14, 20, 24, 28, 26, 34, 34, 46,...
```

из которого следует, что переменная cars представляет собой массив данных (более точно – таблицу данных – тип data.frame, но сути это не меняет):

«Observations: 50» дословно означает «Наблюдений 50 штук» и, если представлять себе эти данные как таблицу, то речь идет о 50 строках;

«Variables: 2» дословно означает «Переменных 2 штуки» и фактически это – количество столбцов;

«$ speed <dbl> 4,4,...» означает, что первый столбец в переменной cars называется «speed» (данные о скоростях), он относится к типу «dbl». double – двойная точность – вещественные числа, модуль которых не превышает $1.7*10^{308}$, все что превосходит этот рубеж обозначается в R символом бесконечности «Inf» (не правда ли, сложно себе представить реальное количество чего-либо, выраженное единицей с 308 нулями);

«$ dist <dbl> 2, 10,...» то же, но с названием второго столбца как «dist».

Иными словами, данные о скорости и соответствующей дистанции сгруппированы в двух столбцах, а по строчкам соответствуют одному и тому же наблюдению (конкретной испытуемой машине).

В R это означает, что можно обратиться к различным наблюдениям, собранным в объекте cars, следующими способами:

```
cars            # Сам объект-массив, вывод на экран консоли
cars$speed   # Массив скоростей (вектор)
cars[ ,1]       # То же самое - значения всех строк из пер-
  # вого столбца, т.е. сам первый столбец
cars$dist      # Массив тормозных путей (вектор)
cars[ ,2]       # То же самое - значения всех строк из вто-
  # рого столбца, т.е. сам второй столбец
cars[1, ]       # Первое наблюдение (значения всех столб-
  # цов из 1-ой строки, т.е. первая строка)
cars[23,]       # 23-е наблюдение (значения всех столбцов
  # из 23-тьей строки, т.е. 23-я строка)
cars[6, 1]     # Скорость при 6-ом наблюдении
cars$speed[6]    # То же самое
cars[5, 2]          # Тормозной путь при 5-ом наблюдении
cars$dist[5]       # То же самое
```

Сравните построчечно выводы на консоли R, запустив каждую строку в отдельности Ctrl+Enter.

Задание 3. Вычислить среднее арифметическое скоростей первых пяти наблюдений в объекте cars.

Решение. В свободной строке составим код, вычисляющий среднее арифметическое первых пяти скоростей:

```
(cars$speed[1] +  cars$speed[2] + cars$speed[3] +
cars$speed[4] + cars$speed[5])/5
```

или короче

```
(cars[1,1] + cars[2,1] + cars[3,1] + cars[4, 1] + cars[5,1])/5
  # То же самое
```

или еще короче

```
mean(cars[1:5,1])          # То же самое
```

или так

```
mean(cars$speed[1:5])      # То же самое
```

Последние две команды вызывают встроенную функцию вычисления среднего арифметического mean для элементов первого столбца speed массива cars, взятых из строк с 1-ой по 5-ую включительно (см. рис. 11).

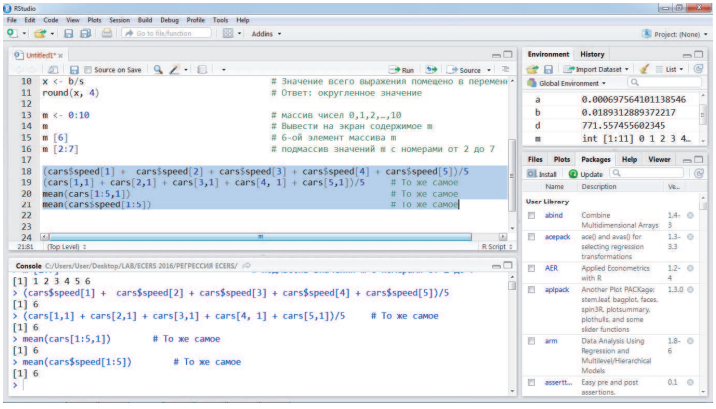


Рисунок 11

Вообще интересно, что если набрать предыдущее выражение, не указав конкретный диапазон наблюдений, то мы вычислим среднее арифметическое скоростей всех испытуемых машин (1920 г.)

```
mean(cars$speed)    # Средняя скорость всех испытуемых
 # машин
```

которая составит 15.4 mph, что эквивалентно 24.78 км/ч (сравните со средними скоростями сегодня).

В последующих разделах мы рассмотрим более подробно типы данных в R и работу с ними.

Некоторые дополнительные настройки RStudio

Пройдите по пунктам основного меню Tools –> Global Options.

Подменю General В появившемся диалоговом окне обязательно укажите во второй строчке путь к рабочей директории R (default working directory). Именно сюда будут записываться все ваши данные по умолчанию и именно отсюда R будет предлагать вам открыть существующие файлы.

Полезно поставить галочки в соответствующие места, регламентирующие автосохранение и пр. (см. рис. 12).

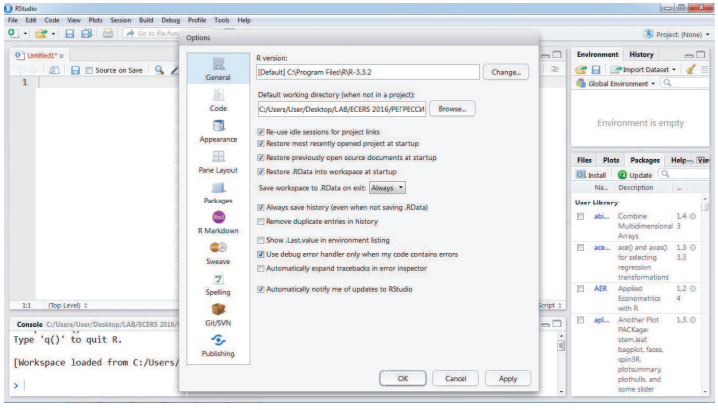


Рисунок 12

Подменю Appearance Данные настройки позволяют выбрать наиболее приятную цветовую палитру, сочетания шрифтов и масштаб отображения на экране. Смело выбирайте различные темы и экспериментируйте с Zoom и Editor font. Не забывайте по итогу нажимать клавишу Apply.

На рисунке 13 приведена настройка, используемая автором данной методички:

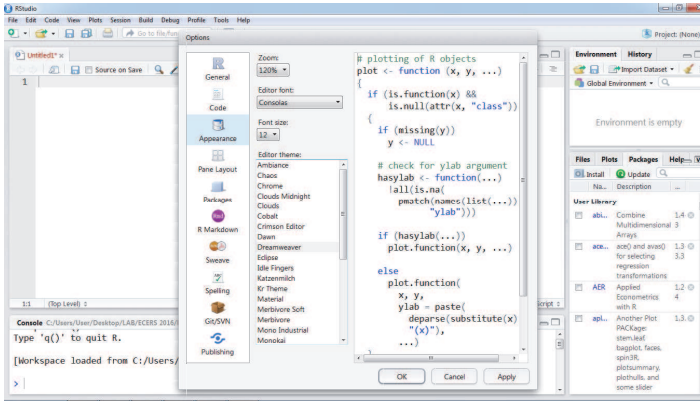


Рисунок 13

Замечание. При выборе Editor font не увлекайтесь – не все шрифты распознают кодировки CP1251 и UTF8!

Подменю Pane Layout Данный раздел позволяет настроить формат вывода информации в четыре окна RStudio, перенастроив решительно все, что можно.

Описание остальных разделов меню выходят за рамки данного практикума, но не менее полезны при углубленном использовании пакета.

Горячие клавиши в R

Ctrl + Shift + N – создать новый документ с будущим кодом программы

Ctrl + Enter – отправить на компиляцию (на выполнение) выделенный фрагмент кода или ту строку, где стоит курсор мыши (если при этом выделения нет).

Ctrl + L – очистить окно консоли (окно вывода результатов компиляции)

Ctrl + Shift + F10 – перезапуск сессии в R

Ctrl + Alt + R – отправить на компиляцию весь текст кода

Ctrl + Shift + C – исключить строку из кода (заремировать строку), повторное нажатие – обратно включить строку в код (снять заремирование).

Ctrl + O – открытие файла. Если файл читается некорректно, необходимо сменить кодировку. Для этого надо запустить в меню **File – Reopen with Encoding** (см. рис. 14):

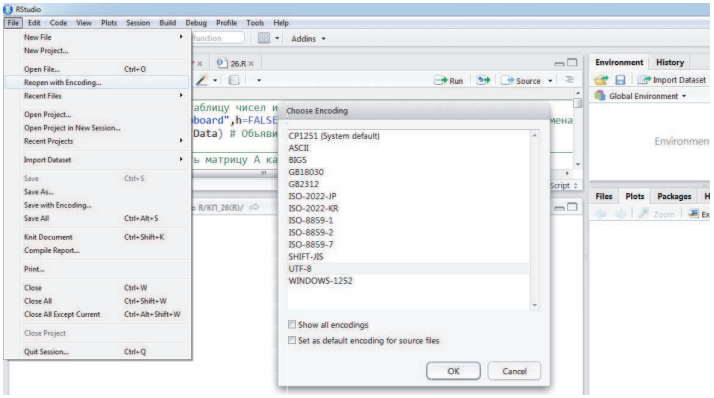


Рисунок 14

и выбрать **UTF-8** или **CP1251**. Лучше сохранять свои файлы в кодировке UTF-8.

F1 – открывает справку R о той функции или команде, перед которой стоит курсор мыши.

Tab – нажатие этой клавиши предложит возможные варианты продолжения текста, включая выбор параметров и переменных в аргументах набираемых команд. Нажимайте эту клавишу чаще.

Alt + «-» – подарок!

Забегая немного вперед

Полезные команды в R

```
version                 # посмотреть версию R
ls()                    # посмотреть какие объекты сейчас
  # находятся в памяти R
```

```
rm(x)                          # удалить объект x из пямяти R
rm(list=ls())                  # очистить всю память в R
install.packages("Rcmdr")      # загрузить из репозитория
  # R пакет "Rcmdr"
library("dplyr")               # активировать уже загру-
  # женную ранее библиотеку dplyr
```

Типичные ошибки в R

1. В любом коде на языке R различаются заглавные и строчные (прописные) буквы. Постарайтесь не путать x с X и select с Select.

2. Всякая открытая скобка должна заканчиваться закрытой, т.е. допустимы конструкции вида {....} и (...). Эту ошибку трудно распознать, если был случайно выделен фрагмент кода без последней закрывающей скобки. Пользователь увидит здесь следующую картину (рис. 15):

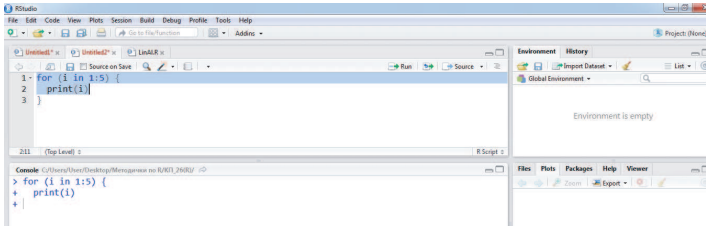


Рисунок 15

Даже пока не понимая смысл введенного кода, нетрудно заметить, что при выделении пользователь не захватил закрывающую фигурную скобку. В этом случае нижнее левое окно консоли (окно результата работы запущенного кода на R) будет заканчиваться сиволом (+).

```
> for (i in 1:5) {
+   print(i)
+
```

Это означает, что пользователем пока не введена корректная строка кода, и R ожидает дополнительного введения. Ощущение примерно такое, что R глухо завис и ни на что не реагирует, повторяя на любые действия свои (+).

Чтобы прервать это кажущееся зависание надо сделать два действия: щелкнуть левой клавишей мыши по левому нижнему окну консоли (активировав его) и нажать клавишу Esc. После этого консоль вернется в рабочее положение, а R прервет компиляцию значком приглашения «>»:

```
> for (i in 1:5) {
+   print(i)
+
>
```

3. Некорректное использование знаков (=), (==), (<-) и (->). Ошибки в этих случаях невероятно сложно диагностировать, поэтому лучше сразу проверить корректность использования этих команд:

– в качестве команды присвоения переменным значений объектов используем только символы (<-) или (->). Например, x <- y или 5 -> z, при этом стрелка указывает на то, кто кому присваивается. Допускается команда x -> y -> z Всегда отделяйте эти стрелки пробелами от переменных, чтобы не потерять возможный минус x <- -3 и избежать ложной интерпретации знака больше > и минуса –.

– в качестве объявления значения параметров в аргументах функций или команд используем только знак (=). Например, seq(from = 1, to = 5, by = 0.1)

– для проверки условий на равенство в операторах if, ifelse, и др. используется двойной знак равенства (==). Например, if (x==2) {print(2)}. Отметим, что «не равно» в R выглядит так: if (x!=2) { print(«Не 2»)}.

4. Кавычки в R не такие как в word!!! «...» – word, "..." – R

5. Десятичным разделителем в R является точка, а не запятая.

Задания для самостоятельной работы

1. Вычислить выражения с точностью в 6 значащих цифр

$$\log_{48.23}\left(2^{-3}+\frac{\sin^3\left(7!+C_{32}^{11}\right)}{\sqrt{1+\operatorname{arctg}\left(\dfrac{1}{1+0.2435}\right)}}\right).$$

Указание. В качестве проверки попробуйте сравнить ответ со значением, полученным в Excel или на калькуляторе.

2. Вычислить выражение с точностью в 3 цифры после запятой

$$\cos^{-1}\left(\frac{1}{\sqrt[3]{0.3532}}-\frac{\coth^3(12)\cdot e^{-1/4.8}}{\sqrt{\left|\log_{13.76}\left(\dfrac{256}{1809.43}\right)\right|+\operatorname{arcctg}\left(7^{-3}\right)}}\right);$$

Указание. Используйте корректные формулы, связывающие котангенс с тангенсом и арккотангенс с арктангенсом.

3. Вычислить среднее арифметическое значение длин тормозных путей для данных cars, выраженное в метрах. Использовать: в 1 футе 0,3048 метра. (Ответ: 13,1м.)

4. После загрузки библиотеки «ggplot2» в R становится доступной таблица данных под именем diamonds, в которой приведены статистические исследования алмазов. Определить сколько алмазов было исследовано и найти средний вес алмазов (в каратах).

Указание. Загрузить библиотеку «ggplot2» и использовать функцию glimps из пакета «dplyr», чтобы понять какую переменную из таблицы необходимо выбрать для дальнейших действий.

5. Проверить, действительно ли при очень малых значениях x функция $\sin x \approx x$. На какую, в таком случае, функцию будет похож $\cos x$?

Практикум 2. Программирование пользовательских функций в R (RStudio)

Задание математических функций

В языке R, как и в любом другом языке программирования, предусмотрена возможность создавать не только различные переменные, но и функции. Если рассматривать математические функции, то здесь объектом будет объявляться не конкретное число или массив, а сама формула или специальный алгоритм, по которым предполагаются дальнейшие вычисления. Это оказывается невероятно удобным при частой реализации одних и тех же действий.

Запустим RStudio, создадим новый лист программы (например, по сочетанию Ctrl+Shift+N) и наберем или скопируем следующий код:

```
f <- function(x) {(x-3)^2}      # Задание функции f(x)=(x-3)^2
```

Что произойдет, если мы отправим этот код на компиляцию (Ctrl+Enter) (см. рис. 16)?

На первый взгляд R никак не отреагировал на введенный текст, однако в действительности, в памяти R создалась функция под именем *f*, значения которой, согласно нашему коду в фигурных скобках {(x-3)^2}, зависят от одной переменной следующим образом:

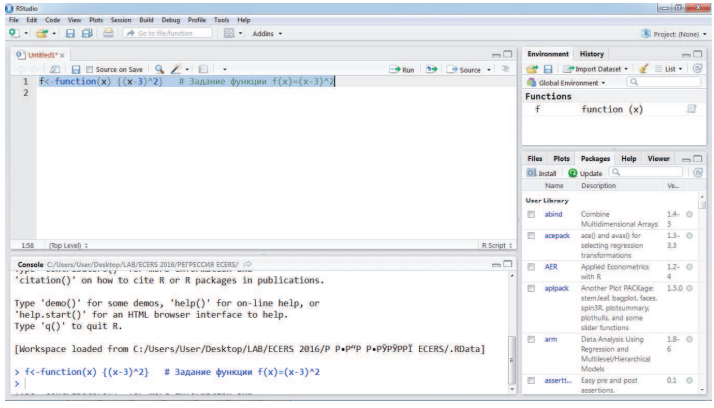


Рисунок 16

$$f(x)=(x-3)^2.$$

Обратите внимание на правое верхнее окно RStudio: в нем появилась только что созданная функция $f(x)$.

Это означает, что теперь R будет понимать обращение к этой функции для конкретных чисел. Например, вычислим значение этой функции в точке $x=5$, набрав (см. также рис. 17):

```
f(5)                    # Значение функции f  в точке 5
```

В качестве аргумента введенной функции f можно подставлять и целые массивы, тогда функция будет вычислена в каждой точке массива (см. также рис. 18):

```
m <- 0:5                # Массив m целых чисел 0,1,...4,5
f(m)                    # Массив значений функции f ,
  # вычисленной в точках из массива m
```

Как видим, строка внизу консоли R перечисляет все значения функции. Таким образом, можно легко задавать/генерировать конечные последовательности значений функции для заданных последовательностях значений аргументов.

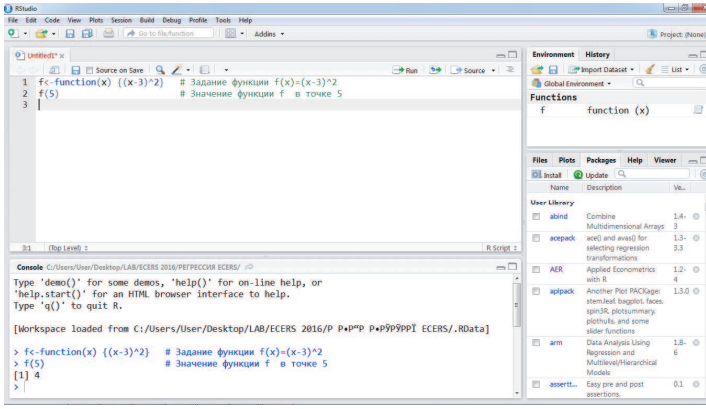


Рисунок 17

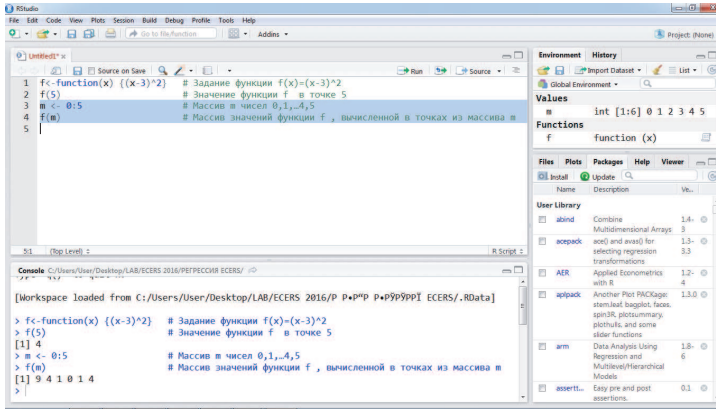


Рисунок 18

Задание 1. Объявить в R функцию

$$g(x,a,b)=\frac{x^2-a}{x-b},$$

где a и b – параметры (по умолчанию считать параметры равными единицам) и вычислить значения $g(2;1;1)$, $g(-3;5;1)$ и $g(6;9;3)$.

Решение. Наберем следующий код в окне программы:

```
g <- function(x , a = 1, b = 1) {(x^2-a)/(x-b)}        # Задание
 # функции g(x,a,b)
```

Если теперь в следующей строчке набрать

```
g(2)                # Вычисление функции  g(x =  2,a = 1, b = 1)
```

мы получим вычисленное значение функции $g(x)$ для $x = 2$, при этом опущенные нами параметры a и b будут приравнены к значениям по умолчанию, т.е. 1, объявленным ранее в аргументах function(x , a = 1, b = 1).

Вызов функции в формате:

```
g(-3, a = 5)      # Вычисление функции  g(x = -3, a = 5, b = 1)
```

даст нам значение функции g для $x = -3$. При этом параметр $a = 5$, а опущенное значение параметра b будет приравнено к значению по умолчанию, т.е. 1.

Естественно, возможно и полное обращение к нашей функции, с указанием всех ее аргументов (см. также рис. 19):

```
g(6, a = 9, b = 3)                          # Вычисление функции
g(x=6,a=9,b=3)
```

Замечание 1. При вызове функции $g(x,a,b)$ мы опускали название переменной, но не было бы ошибкой записать $g(x = 2)$. Вообще, в R можно опускать названия аргументов функции, если только помнить в каком порядке они перечисляются. Так, например, команда $g(2,5)$ будет эквивалента вызову функции $g(x = 2, a = 5, b = 1)$.

Замечание 2. Попробуйте объяснить результат NaN («невычисляемо», «не число»), если ввести команду:

```
g(3, 9, 3)        # Вычисление функции  g(x = 6, a = 9, b = 3)
```

Построение графиков функций

Используем возможности, описанные в предыдущем пункте, для построения графиков функций.

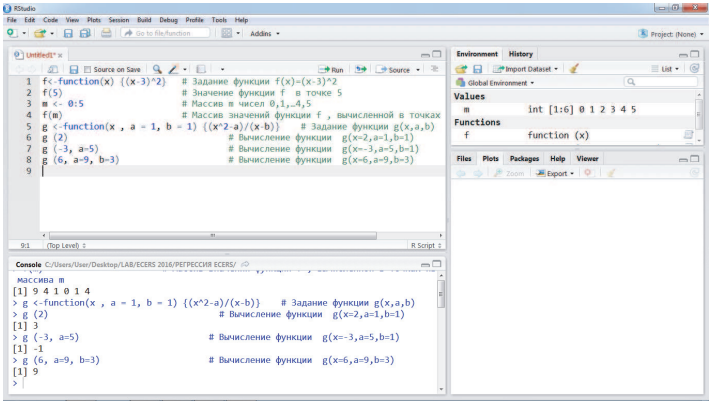


Рисунок 19

Задание 2. Построить график функции $f(x)=(x-3)^2$ на отрезке $[-5;8]$.

Решение. Данную функцию мы уже ранее объявили в R. Зададим последовательность аргументов x, пробегающих отрезок $[-5;8]$ с достаточно малым шагом, например, 0.05. Это можно сделать с помощью команды «seq»:

```
x <- seq(-5, 8, 0.05)      # Последовательность чисел от -5
 # до 8 с шагом 0.05, записанная в x
```

или в расширенном виде

```
x <- seq(from = -5, to = 8, by = 0.05)          # То же самое
```

Теперь остается вызвать стандартную функцию построения графика (см. также рис. 20):

```
plot(x, f(x), type = "l")                  # График функции f(x)
```

В качестве значения параметра типа кривой «type» выбрано значение «*l*» от слова line – соединение точек прямыми линиями. При написании кода на R здесь и далее мы будем иногда выделять эту литеру «l» курсивом: «*l*», чтобы не перепутать ее с единицей. При

копировании такого текста в RStudio форматирование курсивом пропадет.

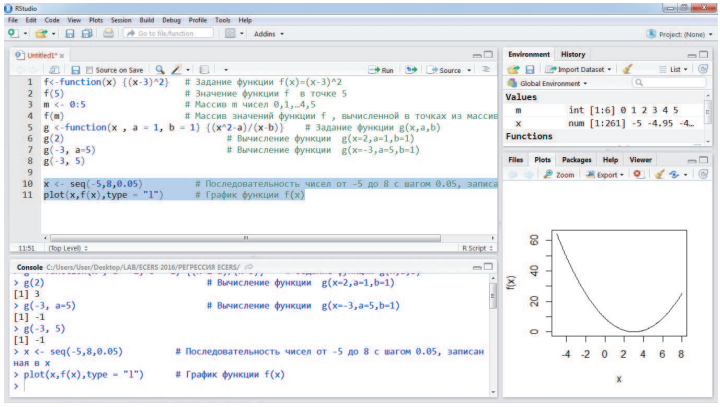


Рисунок 20

Функция seq имеет еще несколько полезных параметров. Например, часто бывает удобно задать последовательность от a до b не с помощью шага, а посредством указания общего количества точек в последовательности.

Следующая ниже строчка кода задает последовательность из 5 чисел, эквидистантно пробегающим диапазон от 21 до 36:

```
x <- seq(from = 21, to = 36, length.out = 5)
  # Последовательность из пяти чисел от 21 до 36
```

```
> x <- seq(from = 21, to = 36, length.out = 5)
#  Последовательность из пяти чисел от 21 до 36
> x
[1] 21.00 24.75 28.50 32.25 36.00
```

Задание 3. Построить графики функции $g(x,a,b)=\dfrac{x^2-a}{x-b}$ на отрезке $[-40;40]$ при различных значениях параметров:

$$a = 4,\ b = -5;$$
$$a = 4,\ b = 2;$$
$$a = 4,\ b = 5.$$

Решение. Функция g нами была уже объявлена. Зададим, как и раньше, последовательность аргументов

```
x <- seq(-40, 40, 0.05)          # Последовательность чисел
 # от -40 до 40 с шагом 0.05, записанная в x
```

и поочередно вызовем построение графиков соответствующих функций (см. также рис. 21):

```
plot(x,g(x,4,-5),type = "l", xlim = c(-50, 50), ylim =c(-50,50))
  # График функции g(x,a=4,b=-5)
plot(x,g(x,4,2),type = "l", xlim = c(-50, 50), ylim =c(-50,50))
 # График функции g(x,a=4,b=2)
plot(x,g(x,4,5),type = "l", xlim = c(-50, 50), ylim =c(-50,50))
 # График функции g(x,a=4,b=5)
```

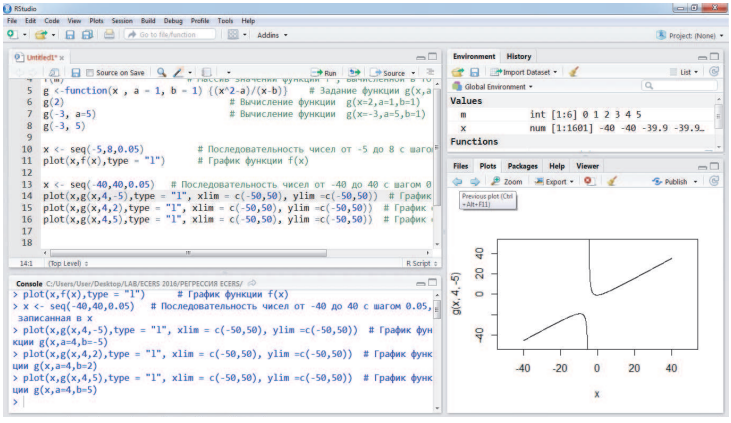


Рисунок 21

Для того, чтобы вернуться к предыдущему или последующему графику используйте в левой верхней части окна графика стрелки вперед и назад. Также, по-

лезно нажать кнопку Zoom, находящуюся чуть правее, для увеличения картинки.

Кстати, часто бывает необходимым разместить несколько графиков на одном рисунке. Достигается это заменой последующих plot на lines.

Например (см. рис. 22):

```
plot(x,g(x, 0, -5),type = "l", xlim = c(-50, 50), ylim =c(-50, 50),
ylab = "y")  # График g(x, 0, -5)
lines(x,g(x,20,-5), type = "l", xlim = c(-50, 50), ylim =c(-50, 50))
  # Добавлен график g(x, 20, -5)
lines(x,g(x,50,-5), type = "l", xlim = c(-50,50), ylim =c(-50,50))
  # Добавлен график g(x,50, -5)
```

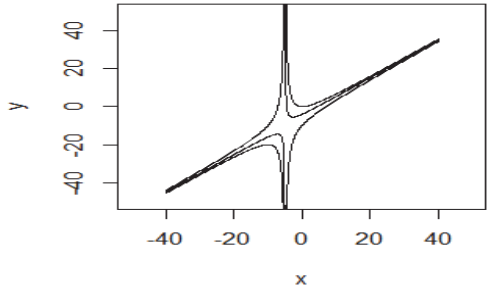


Рисунок 22

Как можно было заметить мы использовали дополнительные параметры в команде plot: xlim и ylim, устанавливающие границы изменения x и y на графике от -50 до 50.

При желании, можно разместить на рисунке привычные оси координат ox и oy, дополнительно используя следующую команду (см. также рис. 23):

```
abline(h = 0, v = 0, col = "gray50")
  # Нанесение на график линий ox и oy
```

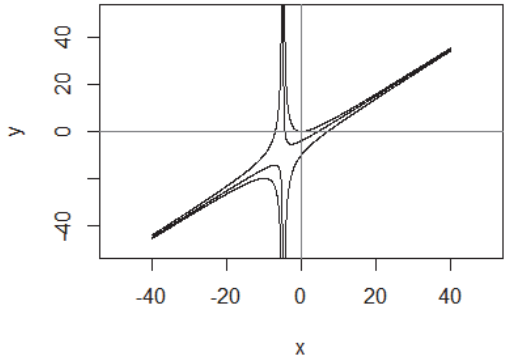


Рисунок 23

Полный перечень аргументов данной команды plot и их точное описание доступно по команде вызова справки:

```
?plot                    # Вызов справки по команде plot
```

или нажатием клавиши F1 при поставленном курсоре перед первой буквой интересующей нас команды. Попробуйте найти графический параметр *tck* для функции plot, который автоматически рисует координатную сетку, если задать его значение $tck = 1$.

Задание произвольных пользовательских функций

Обобщим создание математических функций на случай нескольких значений самой функции.

Задание 4. Запрограммировать функцию Power(x), возвращающую одновременно квадрат и куб своего аргумента x.

Решение. Введем на новых строчках следующий код:

```
Power <- function(x)  {    # Объявление имени функции
                            # Power
 P2 <- x^2                 # Вычисление квадрата в пере-
                           # менной P2
 P3 <- x^3              # Вычисление куба в переменной P3
 return(c(P2, P3))        # Возврат значений P2 и P3
}
Power(5)                  # Вызов функции Power для x = 5
```

Обратим внимание на то, что теперь фигурные скобки {...} после команды function(x) содержат не одно выражение, которое и считалось ранее по умолчанию значением нашей функции, а несколько команд. Во второй строчке мы вычисляем квадрат аргумента и записываем результат в переменную *P2*, а в третьей строке вычисляем куб и записываем в переменную *P3*. Следующей строкой мы должны указать компилятору R какое из вычисленных чисел мы хотим вернуть в качестве результата действия функции Power(x).

Здесь мы могли бы написать return(P2), и тогда функция вернула бы значение квадрата. Для случая return(P3) получили бы на выходе значение куба. Однако мы остановились на return(c(P2, P3)), и это означает, что будут выведены оба значения, т.к. результатом выполнения функции будет формирование массива из двух чисел: c(P2, P3). Замечание: вообще, оператор c(x,y,z,...) объединяет свои аргументы в вектор (массив), но об этом позднее.

В итоге после отработки набранного кода, действительно, получаем строку с квадратом и кубом для $x = 5$ (см. рис. 24).

Если бы мы присвоили значение данной функции какой-нибудь переменной, то эта переменная представляла бы собой обычный массив из двух значений. Например (см. также рис. 24 и 25),

```
Z <- Power(11)          # Вызов функции Power для x = 11
  # с сохранением в Z
```

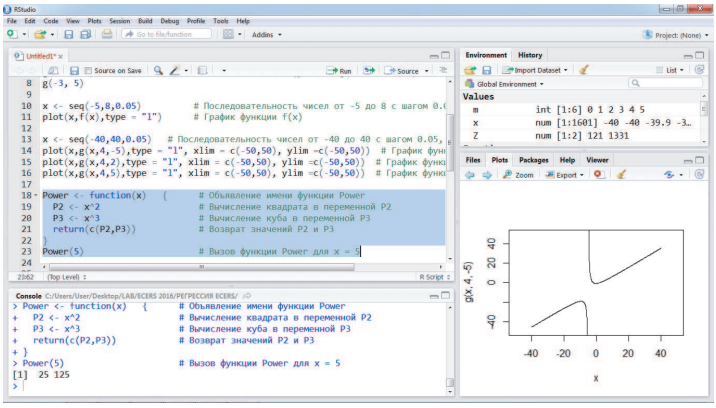


Рисунок 24

```
Z                    # Посмотреть Z (вывести на экран)
Z[1]                 # Посмотреть Z[1]
Z[2]                 # Посмотреть Z[2]
```

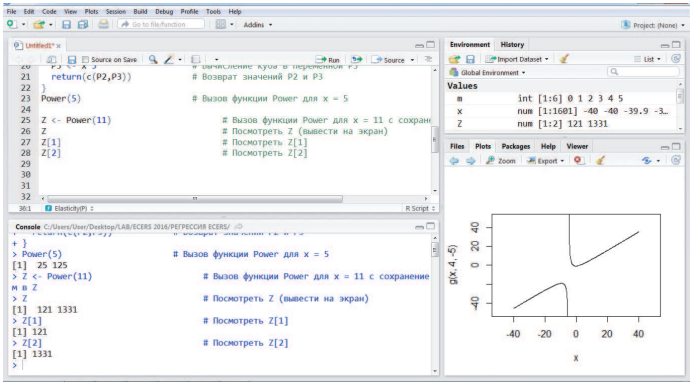


Рисунок 25

На практике часто стоит задача не только вычисления значения какого-либо признака, но и ответ на вопрос о качественном характере исследуемого явления. Например, при изучении зависимости спроса от

предложения нас больше интересует не значения самих зависимостей, а будет ли спрос в принципе эластичным или нет для конкретных цен.

Задание 5. Запрограммировать в R функцию, отвечающую на вопрос: будет ли спрос (Q) эластичным относительно цены предложения (P) для функции $Q(P)=\frac{1}{1+P^2}$. Здесь под спросом Q мы понимаем долю желающих приобрести товар по цене P.

Решение. Фактически нам необходимо вычислить предельную эластичность функции $Q(P)$ по переменной P по формуле

$$E_{Q,P}(P)=\frac{Q'(P)}{Q(P)}P$$

и выяснить превышает ли ее модуль единицу (случай эластичного спроса) или нет:

$$\left|E_{Q,P}(P)\right|>1.$$

В нашем случае

$$Q'(P)=\left(\frac{1}{1+P^2}\right)'=-\frac{2P}{\left(1+P^2\right)^2}.$$

Запишем следующий код в R:

```
Elasticity <- function(P) {        # Объявление имени функ-
  # ции Elasticity
 Q <- 1/(1+P^2)                    # Вычисление спроса Q по цене P
 D <- -2*P/(1+P^2)^2               # Вычисление производной Q по
  # цене P
 E <- D*P/Q                        # Вычисление эластичности E
 if (abs(E)>1) {Elasticity <- "Спрос эластичный"}
# Проверка условия эластичности
 else { Elasticity <- "Спрос неэластичный"}
# Проверка условия неэластичности
 return(Elasticity)                # Возвращение итогового значения
}
Elasticity(2)     # Вызов функции Elasticity для цены P=2
Elasticity(0.5)   # Вызов функции Elasticity для цены P=0.5
```

Здесь использован условный оператор if else. В синтаксисе языка R он имеет представление:

if (A) {...} else {...}

Если условие A выполняется, то производится группа операторов из первых скобок{...}, иначе – из последних. **Важно:** Если бы мы хотели записать условие «равно», то использовали бы двойной знак равенства if(abs(E) == 1)..., а если бы хотели «не равно», то соответственно if (abs(E) != 1)...

Обратите внимание, что значением нашей функции в данном примере является строка, а не число (рис. 26).

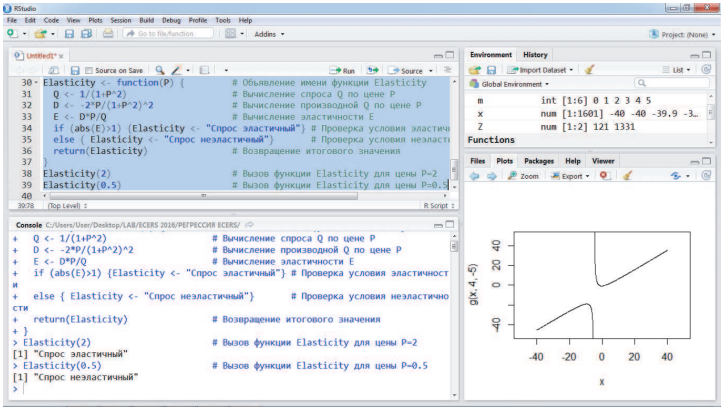


Рисунок 26

Замечание для отличников

Мы могли бы вывести в нашей функции все параметры, связанные с эластичностью: и функцию спроса, и значение ее эластичности, и словесный вывод об эластичности спроса. Делается это все той же командой return, но в качестве аргумента указывается объект специального типа list (список). Все, что мы хотим – это объединить результаты вычислений в один объект, назовем его «Result»:

```
Result <- list(Price=P, Function=Q, Elastic=E, Answer=
 Elasticity)                          # Объединение в Result
```

Здесь команда list объединит в одном объекте все что нам надо. Не забудем вставить этот код в строчку нашей функции перед командой return и исправить аргумент вывода Result в команде return:

```
return(Result)    # Возвращение итогового значения Result
```

Если теперь перезапустить все тело функции, то на выходе уже получим полный отчет об исследовании эластичности. (Кстати, если вообще не вводить команду return, то наша функция фактически превратится в процедуру, выполняющую ряд действий, но не возвращающую ничего в качестве объектов. Если же при этом мы совершали какие-либо вычисления, то в качестве возвращаемого значения будет определено последнее) (см. рис. 27).

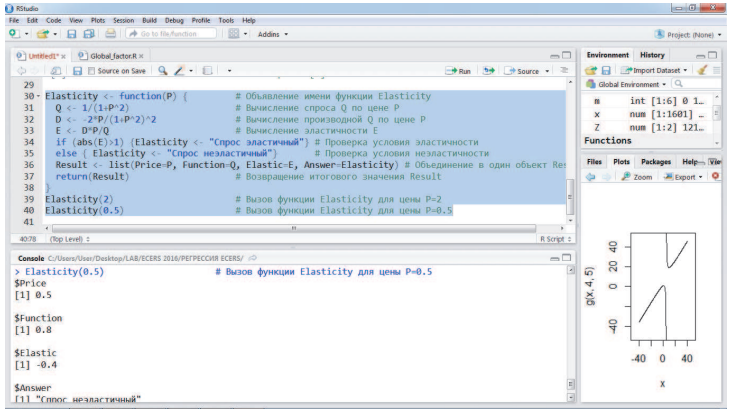


Рисунок 27

Обратите внимание, что в результате наша функция возвращает объект, имеющий в своем составе 4 компоненты: Price (цена), Function (спрос), Elastic (эластичность) и Answer (ответ будет ли спрос эластичным). Это означает, что если присвоить какой-либо переменной значение нашей функции для конкретной цены, скажем, переменной Today:

```
Today <- Elasticity(2)                    # Запись в переменную
  # Today значения Elasticity(2)
```

то в переменной Today будут по отдельности доступны следующие части (их называют полями) (см. также рис. 28):

```
Today$Price                    # Вывести на экран поле Price
  # переменной Today (цена)
Today$Function                 # Вывести на экран поле
  # Function переменной Today (спрос)
Today$Elastic                  # Вывести на экран поле
  # Elastic переменной Today (эластичность)
Today$Price                    # Вывести на экран поле Price
  # переменной Today (словесный ответ)
```

Можно представлять их себе как отдельные самостоятельные переменные, «зашитые» в объекте Today. Мы сталкивались с подобной семантикой в переменной cars, которая содержала в себе по сути два одномерных массива: cars$speed и cars$dist, которые являлись скоростями и длинами тормозных путей. Интересно, что сам язык R понимает переменную cars по типу и как таблицу (тип data.frame), и как список (тип list), и как двумерный массив чисел (тип vector).

Использование векторизованных процедур: оператор ifelse

Рассмотрим достаточно частый случай, когда функция представляет собой две ветви, заданные по-разному, например:

Задание 6. Запрограммировать функцию W(x), заданную системой:

$$W(x)=\begin{cases} x^2, & x \ge 0 \\ \sin(2x), & x < 0 \end{cases}$$

и построить ее график на отрезке $[-10; 5]$.

Решение. На первый взгляд, задача для нас уже не представляет проблемы. Действительно, исполь-

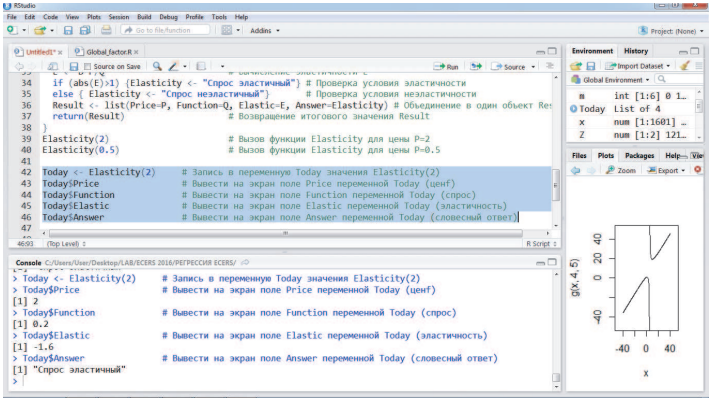


Рисунок 28

зуя оператор if () {} else {} можно получить вполне корректный код функции:

```
W <- function(x)  {                    # Объявление имени
  # функции W(x)
 if (x >= 0) {Res <- x^2} else {Res <- sin(2*x)}
# Вычисление правой и левой ветвей
 return(Res)
}
W(2)                  # Возвращает 2^2 - правая ветвь
W(-pi/4)              # Возвращает sin(-2pi/4) - левая ветвь
```

Однако, у этого кода есть существенный недостаток! Так написанная функция W(x) не будет воспринимать векторизованные обращения. Иными словами, привычным способом построить график этой функции нам не удастся: компилятор R выдаст ошибку на второй строчке кода при попытке подставить в функцию не число, а целый вектор (см. также рис. 29):

```
x <- seq(-10,5, length.out = 101)        # Разбиваем отре-
  # зок [-10; 5] на 100 частей
y <- W(x)                                # Вычисляем значения
  # функции W(x) во всех точках
```

```
plot(x, y, type = "l", lwd = 2, col = "blue")  # Строим гра-
  # фик y = W(x)
```

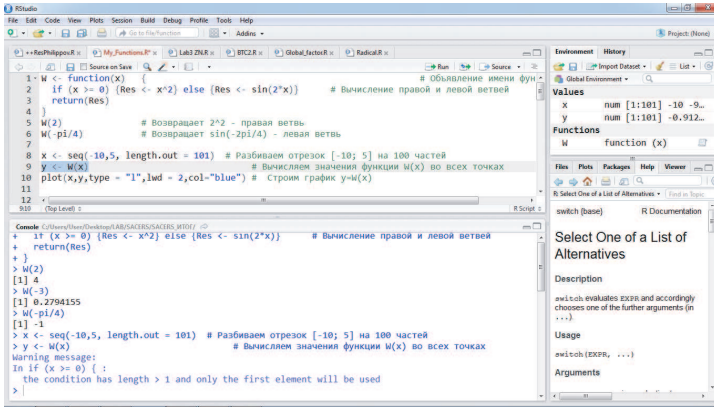


Рисунок 29

Дело в том, что в языке R условный оператор if () {} else {} не работает с векторизованными величинами. Здесь необходимо использовать специальные конструкции.

Самой простой и понятной является схема с привлечением специального оператора **ifelse**(A, expr1, expr2), который при выполнении условия A, реализует действия, составленные в expr1, а при невыполнении условия A, соответственно – в expr2. В нашем случае функция будет запрограммирована так:

```
W <- function(x)
{
 Res <- ifelse(x>=0, x^2, sin(2*x))
 return(Res)
}

x <- seq(-10, 5, length.out = 1001)        # Разбиваем отре-
  # зок [-10; 5] на 1000 частей
y <- W(x)                                  # Вычисляем зна-
```

```
  # чения функции W(x) во всех точках
plot(x, y, type = "l", lwd = 2, col="blue")     # Строим гра-
  # фик y=W(x)
```

Это позволяет построить график функции привычным образом с использованием векторизованных обращений к W(x) (см. рис. 30).

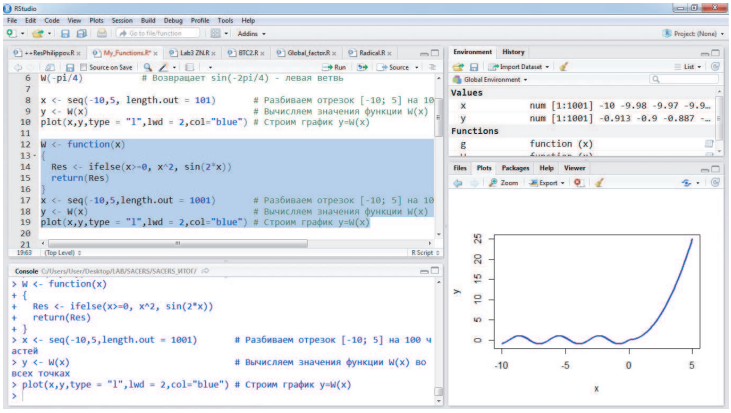


Рисунок 30

Использование векторизованных процедур: оператор [...]

Для построения функций, заданных по-разному на трех и более промежутках, вместо вложенных конструкций ifelse лучше использовать оператор квадратных скобок [...] – самый мощный для векторизованных процедур.

Задание 7*. Запрограммировать в R функцию S(x), заданную системой:

$$S(x)=\begin{cases}\sin(2x), & x<0\\ x^2, & 0\le x\le 2\\ 6-x, & 2<x\end{cases},$$

и построить ее график на отрезке $[-10; 5]$.

Решение. Введем следующий код, задающий функцию S(x) на трех промежутках:

```
S <- function(x)
{
 A <- (x<0)                  # Создаем массив из "TRUE/да"
   # и "FALSE/нет" для x < 0
 B <- (x>=0)&(x<=2)      # Создаем массив из "TRUE/да"
   # и "FALSE/нет" для (x >= 0) и (x <= 2)
 D <- (x>2)                  # Создаем массив из "TRUE/да"
   # и "FALSE/нет" для x > 2
 x[A] <- sin(2*x[A])  # Вычисляем левую ветвь функции
 x[B] <- x[B]^2          # Вычисляем середину функции
 x[D] <- 6 - x[D]        # Вычисляем правую ветвь функции
 Res <- x
 return(Res)
}
```

Идея здесь следующая: весь массив x аргументов функции разделить на три группы A, B и D, в которых функция задана соответственно одной из трех ветвей. Здесь A – это массив из последовательности «да» и «нет». Если в каком-либо месте стоит «да», то на этом месте подходящий аргумент функции для левой ветви, если нет – неподходящий. Аналогично для B и D.

В следующих трех строках значения соответствующих аргументов x[A], x[B] и x[D] переопределяются в значения самой функции.

Теперь можно построить график S(x) (см. рис. 31):

```
x <- seq(-10,5,length.out = 1001)       # Разбиваем отрезок
  # [-10; 5] на 1000 частей
y <- S(x)                                  # Вычисляем значения
  # функции S(x) во всех точках
plot(x, y, type = "l", lwd = 2, col="red")  # Строим график
  # y=S(x)
abline(h = 0, v = 0, col = "gray40")    # Рисуем оси координат
```

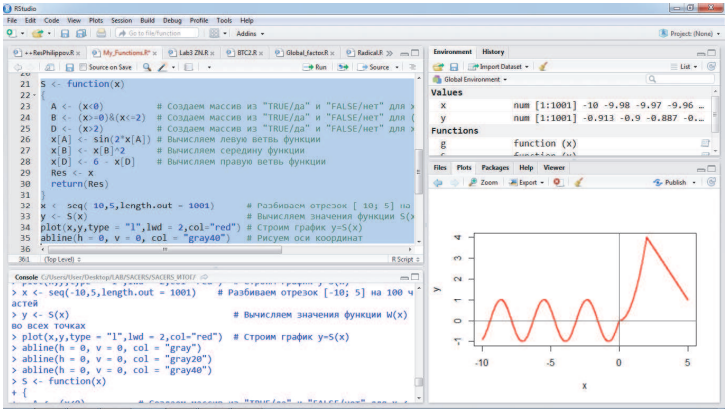


Рисунок 31

Задания для самостоятельной работы

1. Вычислить значения $\sin x$ для первых ста целых чисел: 1..100.

2. Построить график функции $\operatorname{sign} x$ на отрезке $[-2,2]$. *($\operatorname{sign} x$ – функция, возвращающая знак числа x, т.е. +1 для положительных и −1 для отрицательных, в нуле – ноль).* Указание: используйте встроенную в R функцию sign.

3. Объявить в R функцию $\operatorname{Separate}(x)$, которая возвращает два числа: целую и дробную части x. Построить их графики на отрезке $[-3;3]$.

4. Объявить в R функцию $\operatorname{Sink}(x)=\dfrac{\sin x}{x}$ и построить ее график на отрезке $[-20;20]$. Количество точек на отрезке взять равным 401. Будет ли данная функция непрерывна в нуле?

5. * Объявить в R функцию $h(x,y,a)=a+\dfrac{x}{y}$. Значение параметра a принять по умолчанию равным 3. При возникновении неопределенности типа деления

«ноль на ноль» функция должна возвращать сообщение об ошибке, а не «вылетать» с системным R-сообщением: «NaN». Проверить работоспособность функции на примерах $h(2,-2), h(6,3,2), h(0,0,5)$ и $h(-2,0,3)$.

Указание. В конструкциях оператора if (...) {...} else {...} полезно использовать логические операции: A&B (логическое «и») и A|B (логическое «или»).

Ответ:

```
h(2,-2)
[1] 2
> h(6,3,2)
[1] 4
> h(0,0,5)
[1] «Функция не определена»
> h(-2,0,3)
[1] -Inf
```

6. * Раскрасить в различные цвета три ветви функции S(x) в разобранном примере 7.

Приложение к практикуму 2: Библиотека Cairo

Можно было обратить внимание, что графики в R отображаются в менее качественном виде по сравнению с графиками, например, в Excel. Если нам необходимо использовать для отчетов, статей или иных публикаций безупречные по красоте графики – нам требуется воспользоваться библиотекой Cairo, которая автоматически превращает любой график в R в фотографическое чудо.

Загрузим библиотеку Cairo через меню Tools и активизируем командой

```
library(Cairo)                    # Для красивых графиков
```

Далее введем код программы, отображающей обычный график параболы и косинуса:

```
# Вычисляем y(x) и z(x)
x <- seq(-4, 4, length.out = 100)          # разбиваем отрезок
  # [-4, 4] на 100 точек
y <- x^2-4                    # Вычисляем значения параболы
z <- 7*cos(x*pi/4)          # Вычисляем значения косинуса

# Обычный график в R (RStudio)
plot(x, y, type = "l", col = "blue", lwd = 2)
lines(x, z, type = "l", col = "red", lwd = 3)
abline( h = 0, v = 0, lty = 2)
```

Теперь повторим графические команды, предварив их функцией Cairo... Только не торопитесь компилировать последнюю строчку кода dev.off(), чтобы не закрыть окно вывода графиков раньше времени.

```
# Сглаженный рисунок на экран
CairoWin(6, 6, bg="transparent")
    plot(x, y, type = "l", col = "blue", lwd = 2)
    lines(x, z, type = "l", col = "red", lwd = 3)
    abline( h = 0, v = 0, lty = 2)
dev.off()                              # Закрывает окно вывода
```

Далее приводим для сравнения оба рисунка, попробуйте угадать: кто есть кто (см. рис. 32).

Следующая модификация кода отправит идеальный рисунок в файл по указанному пути и имени:

```
# Сглаженный рисунок в pdf-файл
CairoPDF("C:\\Users\\Asus\\Desktop\\My_R\\TEMP\\
plot.pdf", 6, 6, bg="transparent")
    plot(x, y, type = "l", col = "blue", lwd = 2)
    lines(x, z, type = "l", col = "red", lwd = 3)
    abline( h = 0, v = 0, lty = 2)
dev.off()                            # Закрывает запись в файл
```

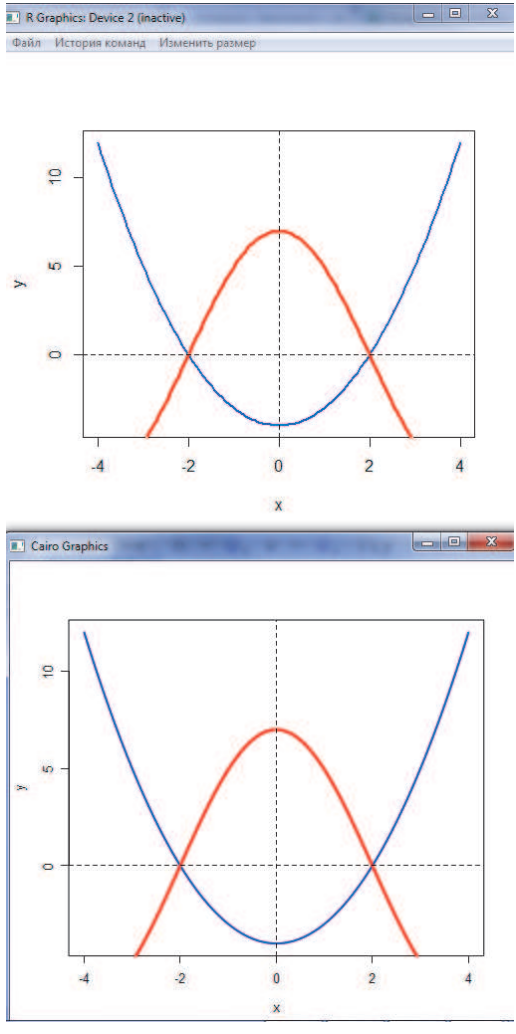

R Graphics: Device 2 (inactive)
Файл История команд Изменить размер
y
10
5
0
-4
-2
0
2
4
x
Cairo Graphics
y
10
5
0
-4
-2
0
2
4
x

Рисунок 32

Замечание. Обратите внимание как оформляется путь к файлу при обащении в команде – это общий стандарт в R.

Приложение к практикуму 2: Функция Radical

Обычная проблема для языков программирования – отказ базовой функции возведения в степень работать с отрицательными аргументами в дробных степенях, т.е. число $\sqrt[7]{-128} = (-128)^{1/7}$ так просто вычислить не удастся. Связано это с тем, что в абстрактном представлении дробь $1/7$ выглядит неотличимо от $2/14$, но для последней арифметический корень 14-ой степени не определен для отрицательных чисел. Именно поэтому, функция возведения в нецелую степень просто не определяется для отрицательных аргументов.

Ниже приведен код функции **Radical**, которая вычисляет радикалы (корни) произвольной целой степени. Для случая четной степени корня – функция возвращает положительное значение (арифметический корень). Функция поддерживает векторизованные обращения.

Запустите этот код, чтобы понять, как делать задание 6*.

```
# Radical(x, Power, err) - функция корень n-ой степени.
# P.S. При четных n - арифметический корень
# x - аргумент корня, Power - степень корня (по умолча-
# нию 3),
# err - допустимое отличие степени от целого числа  (по
# умолчанию 10^(-9))
# by  www.zadadaev.com

Radical <- function(x, Power=3, err = 10^(-9))
{
 if (abs(Power-round(Power, 0))>err) {paste("Ошибка! Сте-
```

```
  # пень корня не целое число:", Power)} else
 {
  A <- (x>=0)
  B <- (x<0)
  if (abs(Power/2-round(Power/2,0))>0.1)  # Нечетный
  # случай
  {
   x[A] <- x[A]^(1/Power)
   x[B] <- -(-x[B])^(1/Power)
   Res <- x
   return(x)
  }
  else    # Случай четной степени
  {
   x[A] <- x[A]^(1/Power)
   x[B] <- NA
   Res <- x
   return(x)
  }
 }
}

# Подключить к файлу можно командой:
# source("Radical.R") # При этом надо поместить файл
# Radical.R в рабочую директорию R


# Примеры обращения:
Radical(-8)             # Кубический корень из -8
Radical(34.09, Power = 7)   # Корень 7 степени из 34.09
Radical(64, 6)          # Арифметический корень 6-ой
  # степени из 64
Radical(-81, 2)         # Арифметический корень из -81
  # (не определен)
Radical(4, 8.03)          # Ошибочная степень корня 8.03

# Иллюстрации
```

```
x <- seq(-1.5, 1.5, length.out = 501)
plot(x,Radical(x, 1), type = "l", lwd = 2)
abline(h = 0, v = 0, col = "gray40")
lines(x, Radical(x, 2), type = "l", col="blue", lwd = 2)
lines(x, Radical(x, 3), type = "l", col="green", lwd = 2)
lines(x, Radical(x, 8), type = "l",col="red", lwd = 2)
lines(x, Radical(x, 15), = "l", col="orange", = 2)
```

Практикум 3. Исследование нулей и экстремумов функций (RStudio)

В данном разделе мы познакомимся с вычислительными методами определения нулей функции и ее экстремумов с помощью R. При этом некоторое внимание уделим тонкостям оформления графиков функций со всевозможными дополнительными построениями, точками и текстами. Ведь, как известно, всего один красочный и понятный график позволяет «выбить» из инвестора максимум вложений.

Успех современной вычислительной математики находится в тесной связи с двумя конкурирующими направлениями: первое из них – это использование математически эффективных быстрых алгоритмов приближенных вычислений; второе – наличие самих вычислительных мощностей (компьютеров). На сегодняшний день оказывается, что в рассматриваемых задачах поиска нулей функции или ее экстремумов прямой перебор миллионов вариантов происходит в очень короткое время (в пределах от нескольких секунд до десятка минут), что позволяет вытеснить специфические математические алгоритмы на второй план из-за ряда трудностей и ограничений последних.

Мы рассмотрим оба подхода, но начнем с самого простого и популярного аналога в Excel: «Подбор параметра» из анализа «что если» (см. рис. 33).

Аналог Excel «Подбор параметра»

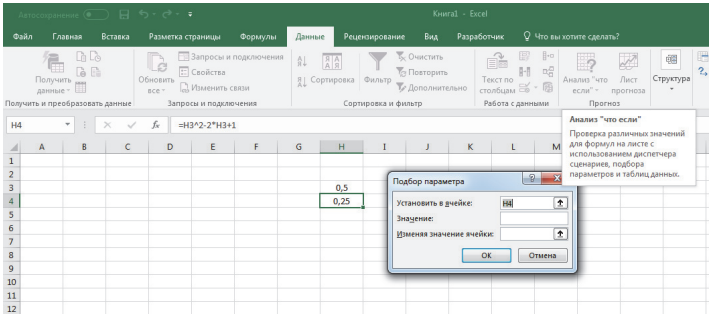


Рисунок 33

По сути здесь ставится задача о нахождении параметра x под заданное значение y_0 некоторого выражения $f(x)$, т.е. фактически ставится задача о приближенном решении уравнения

$$f(x)=y_0\,.$$

Задание 1. a) Объявить в R функцию

$$f(x)=\frac{x^3+3x^2-1}{x^2+1}$$

построить ее график на отрезке $[-4;2]$, нанести горизонтальным пунктиром уровень значения функции $y_0=2$ и приближенно отобразить соответствующую точку A на графике;

b) Найти соответствующее значение аргумента функции, т.е. приближенно решить уравнение

$$f(x)=2\,;$$

c) Аналогично предыдущим пунктам, сделать соответствующий рисунок и приближенно решить уравнение

$$f(x)=0.3\,.$$

Решение. Наберем следующий код в левом верхнем окне RStudio:

```
f <- function(x) {(x^3 + 3*x^2 - 1)/(x^2 + 1)}      # Задаем
  # функцию f(x)
dx <- 0.0001                  # Задаем точность по аргументу x
x <- seq(-4, 2, by = dx)                  # Разбиваем отрезок
  # [-4; 2] с шагом dx
plot(x, f(x), type = "l", lwd = 2, col = "blue", main =
"Подбор параметра для f(x) = 2")
abline(h = 0, v = 0, col = "gray40")    # Рисуем оси координат
```

Параметр lwd=2 в команде plot делает график несколько жирнее, а оператор *abline* выводит на последний рисунок горизонтальную линию $h = 0$ и вертикальную $v = 0$.

Добавим к рисунку горизонтальную линию уровня $y_0 = 2$ и сделаем ее зеленым пунктиром с помощью параметров col = «green» и lty = «84» (см. рис. 34):

```
abline(h = 2, col = "green", lty = "84")          # lty = "84":
  # 8 длина штриха, 4 - длина пробела
```

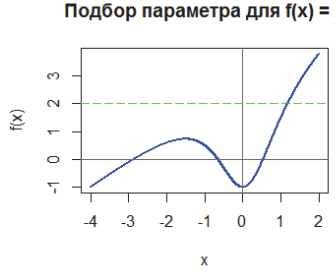


Рисунок 34

Теперь остается, посмотрев рисунок, примерно определить координаты точки A пересечения пунктира с графиком и отобразить ее на графике (с нескольких попыток получилось $x = 1.18$):

```
# Отмечаем точку A желтым  цветом с красной границей:
points(x = 1.18, y = f(1.18), col = "red", pch = 21, bg =
"yellow")
```

```
text(1.1,2.5,"A")                    # Подписываем рядом точку
  # литерой A
```

В итоге получаем безупречный рисунок 35:

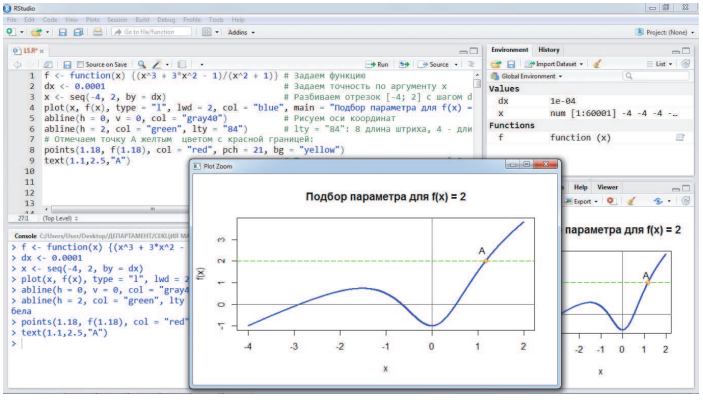


Рисунок 35

Задание a) выполнено.

Основной наш интерес в пункте b). Как максимально точно узнать абсциссу точки A, т.е. какое значение x является корнем уравнения $f(x)=2$? Запишем три, почти «магические», строчки кода и обсудим почему они дают искомый результат:

```
i <- which.min(abs(f(x)-2))      # Определяем какой номер i
  # у точки A среди разбиения отрезка
x[i]                  # Координата x точки A, т.е. наш корень
f(x[i])                          # Координата y точки A,
  # т.е. значение самой функции
```

```
> i <- which.min(abs(f(x)-2)) # Определяем номер i
у точки A среди разбиения отрезка
> x[i]                        # Координата x точки A,
т.е. наш корень, параметр
[1] 1.1746
```

```
> f(x[i])                    # Координата y точки A,
т.е. значение самой функции
[1] 2.000111
```

Итак, корень нашего уравнения $x = 1.1746$, при этом сама функции демонстрирует хорошую близость к 2: $f(x) = 2.000111$. Если мы хотим еще большей точности – нам необходимо уменьшить шаг dx и повторить процедуру еще раз.

Теперь очень важно все-таки понять: почему это работает! Сразу хотим отметить, что подобный прием приводит к очень быстрому результату, даже в оперировании с миллионами точек разбиения $\{x_i\}$.

Командой i <- which.min(abs(f(x)-2)) мы фактически определяем: какая по счету точка среди всех точек $\{x_i\}$ разбиения отрезка $[-4;2]$ соответствует минимуму функции

$$|f(x)-2|.$$

Действительно, все корни уравнения

$$f(x) = 2,$$

или эквивалентного ему

$$f(x) - 2 = 0,$$

соответствуют минимумам функции $|f(x)-2|$. Это хорошо видно из графиков функций (см. рис. 36–38):

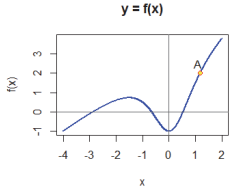


Рисунок 36

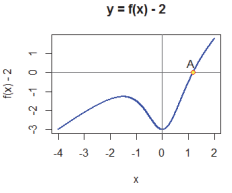


Рисунок 37

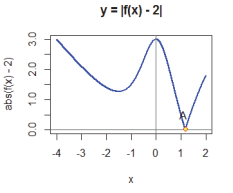


Рисунок 38

Таким образом, чтобы на некотором отрезке $[a;b]$ найти корень какого-либо уравнения

$$f(x) = y_0,$$

необходимо разбить весь отрезок на множество точек с малым шагом dx (это и будет точность самого кор-

ня x) и далее найти минимальное среди всех значений функции

$$\left|f(x_i)-y_0\right|$$

на выбранном разбиении $\{x_i\}$.

Решим пункт c). Проделаем аналогичные построения для графика, но уже с уровнем $y_0 = 0.3$:

```
plot(x, f(x), type = "l", lwd = 2, col = "blue", main = "f(x) = 0.3", col.main="red")
abline(h = 0, v = 0, col = "gray40")                    # Рисуем
  # оси координат
abline(h = 0.3, col = "green", lty = "84")              # lty = "84":
  # 8 - длина штриха, 4 - длина пробела
```

Однако, на рисунке теперь хорошо видны три корня – три пересечения графика функции с пунктирной линией уровня $y_0 = 0.3$ (см. рис. 39):

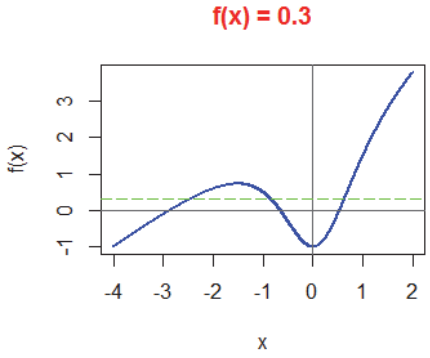


Рисунок 39

И, несмотря на то, что в теории функция $\left|f(x)-y_0\right|$, конечно, имеет три нулевых минимума (это и есть наши корни) (см. рис. 40):

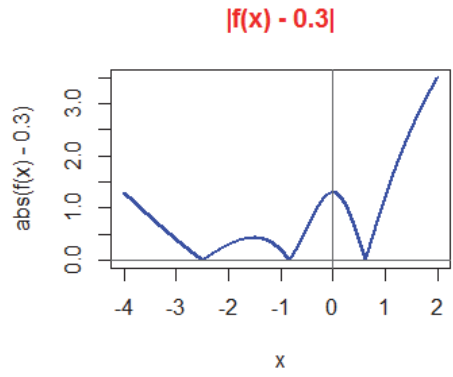


Рисунок 40

найти сразу все корни не всегда возможно, т.к. мы перебираем не все точки функции (их бесконечное несчетное множество), а лишь точки разбиения отрезка, заданные последовательностью x <- seq(-4, 2, by = dx). Тем самым, наши приближенные нули могут оказаться вовсе не нулями и, скорее всего, разными между собой. Функция which.min(abs(f(x)-yo)) выберет из них строго наименьший, и этот выбор нам всегда будет казаться случайным.

Именно поэтому, при приближенном решении уравнений необходимо сначала достаточно точно построить график функции, чтобы быть уверенными, что все корни хорошо «видны», и визуально определить окрестности, содержащие в себе ровно по одному корню. Кстати, применение «подбора параметра» в Excel без учета этой особенности может привести к непредсказуемым ответам, далеким от практического смысла.

Вернемся снова к нашему графику (рис. 41) и визуально определим все три окрестности для корней A, B и C уравнения $f(x)=0.3$. Этими окрестностями можно назвать $[-3;-2]$, $[-1;0]$ и $[0;1]$.

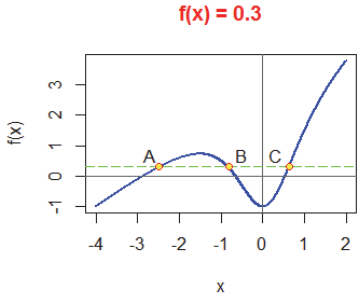


Рисунок 41

Теперь остается последовательно ввести для корня A:

```
x <- seq(-3, -2, by = dx)          # Разбиваем отрезок [-3; -2]
  # с шагом dx
i <- which.min(abs(f(x)-0.3))    # Определяем какой номер i
  # у точки A среди разбиения отрезка
x[i]                  # Координата x точки A, т.е. наш корень
f(x[i])               # Координата y точки A, т.е. значение
  # самой функции
```

с результатом:

```
> x[i]     # Координата x точки A, т.е. наш корень
[1] -2.4904
> f(x[i])   # Координата y точки A, т.е. значение
самой функции
[1] 0.2999942
```

для корня B:

```
x <- seq(-1, 0, by = dx)           # Разбиваем отрезок [-3;
  # -2] с шагом dx
i <- which.min(abs(f(x)-0.3))    # Определяем какой номер i
  # у точки A среди разбиения отрезка
x[i]                  # Координата x точки A, т.е. наш корень
f(x[i])               # Координата y точки A, т.е. значение
  # самой функции
```

с результатом:

```
> x[i]                          # Координата x точки A,
т.е. наш корень
[1] -0.8349
> f(x[i])                       # Координата y точки A,
т.е. значение самой функции
[1] 0.3000488
```

и для корня C:

```
x <- seq(0, 1, by = dx)            # Разбиваем отрезок [-3; -2]
  # с шагом dx
i <- which.min(abs(f(x)-0.3))  # Определяем какой номер i
  # у точки A среди разбиения отрезка
x[i]                  # Координата x точки A, т.е. наш корень
f(x[i])               # Координата y точки A, т.е. значение
  # самой функции
```

с результатом:

```
> x[i]          Координата x точки A, т.е. наш корень
[1] 0.6253
> f(x[i])      # Координата y точки A, т.е. значение
самой функции
[1] 0.3001385
```

Обратите внимание на получающиеся ответы для функции – они достаточно близки к заявленному уровню 0.3. Попробуйте улучшить точность для последнего корня C, выбрав dx = 0.0000001, что будет соответствовать 10 млн. точек разбиения отрезка.

Задание 2. Для функции из предыдущего задания

$$f(x)=\frac{x^3+3x^2-1}{x^2+1}$$

приближенно найти ее локальные экстремумы на отрезке $[-4;2]$ и изобразить ее наклонную асимптоту.

Решение. Представим рисунок для нашей функции по-другому:

```
f <- function(x) {(x^3 + 3*x^2 - 1)/(x^2 + 1)}  # Задаем
  # функцию
dx <- 0.00001                 # Задаем точность по аргументу x
x <- seq(-4, 2, by = dx)    # Разбиваем отрезок [-4; 2] с ша-
  # гом dx
plot(x, f(x), type = "l", lwd = 2, col = "blue", main = "Ло-
  # кальные экстремумы",
     xlab = "Значения x", ylab = "Значения f(x)")
abline(h = 0, v = 0, col = "gray40")       # Рисуем оси коор-
  # динат
```

и после нескольких попыток подбора точек-маркеров локальных экстремумов на графике:

```
# Отмечаем точки экстремумов желтым  цветом с красной
# границей:
points(x = c(-1.5,0), y = c(0.73,-1), col = "red", pch = 21,
bg = "yellow")
text(-1.5,1.2,"Max", col = "blue")              # Подписываем
  # точку словом Max
text(-0.6,-0.8,"min", col = "blue")             # Подписываем
  # точку словом min
```

получим вполне приемлемый для научной статьи рисунок 42:

Рисунок 42

Обратите внимание, что выделено две точки Max (-1.5; 0.73) и min(0, −1), следовательно, в команде

points(x = c(-1.5,0), y = c(0.73,-1), col = “red”, pch = 21, bg = “yellow”)

должны перечисляться сначала x – координаты этих пар точек: x = c(-1.5, 0), а затем y – координаты этих же пар точек: y = c(0.73, -1).

Теперь нам остается только выделить окрестности локальных экстремумов: $[-2;1]$ для max и $[-1;1]$ для min. Скрипт нахождения локальных экстремумов здесь такой: для точки Max

```
x <- seq(-2, -1, by = dx)          # Разбиваем отрезок [-2; -1]
  # с шагом dx
i <- which.max(f(x))               # Определяем какой номер
  # i у точки Max среди разбиения отрезка
x[i]                               # Координата x точки Max,
  # т.е. наш локальный максимум
f(x[i])                            # Координата y точки Max,
  # т.е. значение самой функции
```

с результатом:

```
> x[i]                             # Координата x точки
Max, т.е. наш локальный максимум
[1] -1.51275
> f(x[i])                          # Координата y точки
Max, т.е. значение самой функции
[1] 0.730882
```

для точки min

```
x <- seq(-1, 1, by = dx)           # Разбиваем отрезок [-4; 2]
  # с шагом dx
i <- which.min(f(x))               # Определяем какой номер
  # i у точки min среди разбиения отрезка
x[i]                               # Координата x точки min,
  # т.е. т.е. наш локальный минимум
f(x[i])                            # Координата y точки min,
  # т.е. значение самой функции
```

с результатом:

```
> x[i]          # Координата x точки min, т.е. наш
локальный минимум
[1] 0
> f(x[i])       # Координата y точки min, т.е. зна-
чение самой функции
[1] -1
```

Как видим, мы использовали здесь две функции which.max(f(x)) и which.min(f(x)) по прямому назначению – определение из множества значений функции $f(x_i)$ на разбиении $\{x_i\}$ соответственно максимального и минимального значений, вернее их порядковых номеров i.

Кстати, на всей области определения больше локальных экстремумов у функции нет. Чтобы в этом убедиться, требуется взглянуть на функцию издалека, выбрав отрезок пошире:

```
f <- function(x) {(x^3 + 3*x^2 - 1)/(x^2 + 1)}      # Задаем
  # функцию f(x)
dx <- 0.01                          # Задаем шаг по аргументу x
x <- seq(-10, 10, by = dx)          # Разбиваем отрезок [-4; 2]
  # с шагом dx
plot(x, f(x), type = "l", lwd = 2, col = "blue", main = "Ло-
кальные экстремумы",
     xlab = "Значения x", ylab = "Значения f(x)")
abline(h = 0, v = 0, col = "gray40")                # Рисуем оси
  # координат
```

На рисунке 43 хорошо видна наклонная асимптота.

Ее несложно получить делением многочленов в столбик:

$$\frac{x^3+3x^2-1}{x^2+1} = \underbrace{x+3}_{\text{это асимптота}} + \frac{-x+2}{x^2+1}.$$

Если теперь нанести на рисунок пунктиром уравнение наклонной асимптоты $y = x+3$ (см. рис. 43):

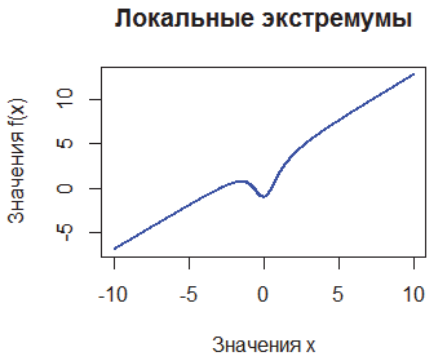


Рисунок 43

```
lines(x, x+3, col = "green", lty = "8424")      # Добавляем
  # сложным пунктиром "8424" асимптоту
```

то получим весьма наглядное представление о функции в целом (см. рис. 44):

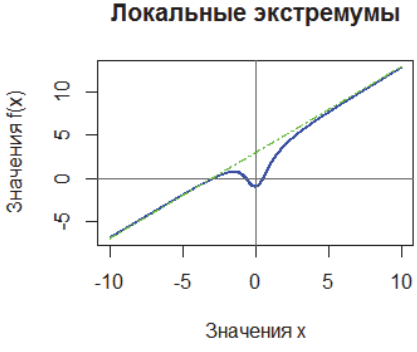


Рисунок 44

Замечание. Очень полезно получить этот график аналитически, исследуя первую и вторую производные.

Процедура поиска нулей функции: uniroot

Задание 3. Для параболы

$$f(x) = x^2 - 5x + 4$$

построить график в окрестности ее нулей, отметив их маркерами, и приближенно найти эти же корни с помощью процедуры uniroot.

Решение. Корнями данной параболы являются 1 и 4. Изобразим в R график параболы на отрезке $[-1;6]$, включающем оба корня:

```
f <- function(x) {x^2 - 5*x + 4}                    # Задаем параболу
dx <- 0.0001                                        # Задаем точность
  # по аргументу x
x <- seq(-1, 6, by = dx)                            # Разбиваем отре-
  # зок [-1; 6] с шагом dx
plot(x, f(x), type = "l", lwd = 2, col = "blue")    # Рисуем
  # f(x) жирной линией
abline(h = 0, v = 0, col = "gray40")                # Рисуем
  # оси координат
# Отмечаем точки корней желтым цветом с красной границей:
points(c(1,4), c(f(1), f(4)), col = "red", pch = 21, bg = "yellow")
```

Теперь в обязательном порядке остается только выделить окрестности каждого корня и применить процедуру *uniroot*. Для первого корня $x = 1$ выделяем окрестность $[0;2]$ и получаем:

```
# Нахождение корня f(x) в окрестности [0, 2]
dx <- 0.00000011                # Шаг dx (точность определе-
  # ния точки корня)
a <- 0; b <- 2                  # Задаем концы интервала, где
  # ищем корень
uniroot(f,c(a,b),tol=dx)        # Полная информация о най-
  # денном корне
```

или более подробно по каждому полю функции uniroot (см. рис. 45):

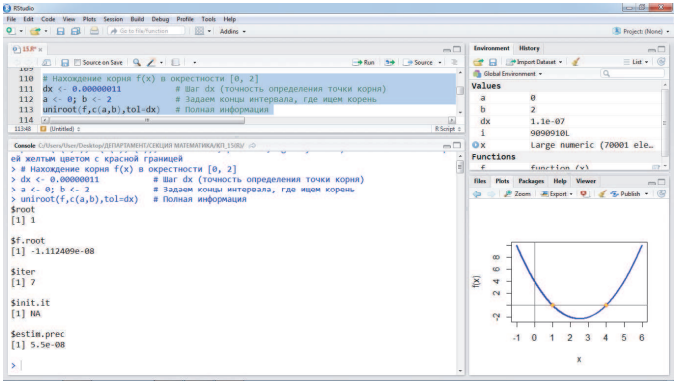


Рисунок 45

```
uniroot(f,c(a,b),tol=dx)$root          # Один найденный
  # корень
uniroot(f,c(a,b),tol=dx)$f.root        # Значение функции
  # в найденной точке
uniroot(f,c(a,b),tol=dx)$iter          # Число итераций
uniroot(f,c(a,b),tol=dx)$estim.prec    # Точность решения
```

с результатом:

```
> uniroot(f,c(a,b),tol=dx)$root          # Один най-
денный корень
[1] 1
> uniroot(f,c(a,b),tol=dx)$f.root        # Значение
функции в найденной точке
[1] -1.112409e-08
> uniroot(f,c(a,b),tol=dx)$iter          # Число ите-
раций
[1] 7
> uniroot(f,c(a,b),tol=dx)$estim.prec    # Точность
решения
[1] 5.5e-08
```

Аналогично для второго корня $x = 4$ выделяем окрестность $[3;5]$ и получаем:

```
dx <- 0.00000011                    # Шаг dx (точность
  # определения точки корня)
a <- 3; b <- 5                      # Задаем концы ин-
  # тервала, где ищем корень
uniroot(f,c(a,b),tol=dx)            # Полная информация
# или подробнее...
uniroot(f,c(a,b),tol=dx)$root       # Один найденный
  # корень
uniroot(f,c(a,b),tol=dx)$f.root     # Значение функции
  # в найденной точке
uniroot(f,c(a,b),tol=dx)$iter       # Число итераций
uniroot(f,c(a,b),tol=dx)$estim.prec    # Точность решения
```

с результатом:

```
> uniroot(f,c(a,b),tol=dx)$root              # Один
найденный корень
[1] 4
> uniroot(f,c(a,b),tol=dx)$f.root          # Значе-
ние функции в найденной точке
[1] -1.112408e-08
> uniroot(f,c(a,b),tol=dx)$iter              # Число
итераций
[1] 7
> uniroot(f,c(a,b),tol=dx)$estim.prec       # Точ-
ность решения
[1] 5.5e-08
```

При использовании *uniroot* важно знать, что выбираемый нами отрезок должен на своих концах давать разные знаки у функции, т.к. алгоритм нахождения корня (деление отрезка пополам) опирается на теорему о прохождении через ноль непрерывной функции с разными знаками на концах отрезка. Казалось

бы, если корень на отрезке один, то знаки на концах будут всегда разными. Это не так! И воспользоваться данной процедурой uniroot совершенно не получится в следующем характерном примере.

Задание 4. Для функции

$$f(x)=(1-x)\cdot(x-4)^2$$

построить график в окрестности нулей, отметив их маркерами, и приближенно найти эти же корни с помощью процедуры uniroot или каким-либо другим численным способом.

Решение. Корнями данного многочлена по-прежнему являются 1 и 4 с той лишь разницей, что теперь корень $x=4$ – кратный (повторяется).

Так же, как и в предыдущем задании, построим график функции на отрезке $[-1;6]$ (см. также рис. 46):

```
f <- function(x) {-(x-1)*(x-4)^2}          # Задаем функцию
dx <- 0.0001                               # Задаем точность
  # по аргументу x
x <- seq(-1, 6, by = dx)                   # Разбиваем отре-
  # зок [-1; 6] с шагом dx
plot(x, f(x), type = "l", lwd = 2, col = "blue")      # Рисуем
  # f(x) жирной линией
abline(h = 0, v = 0, col = "gray40")                  # Рисуем
  # оси координат
# Отмечаем точки корней желтым цветом с красной границей
points(c(1,4), c(f(1),f(4)), col = "red", pch = 21, bg = "yellow")
```

Нахождение первого корня не вызывает проблем:

```
# Нахождение корня f(x) в окрестности [0, 2]
dx <- 0.00000011          # Шаг dx (точность определения
  # точки корня)
a <- 0; b <- 2             # Задаем концы интервала, где
  # ищем корень
uniroot(f,c(a,b),tol=dx)   # Полная информация
```

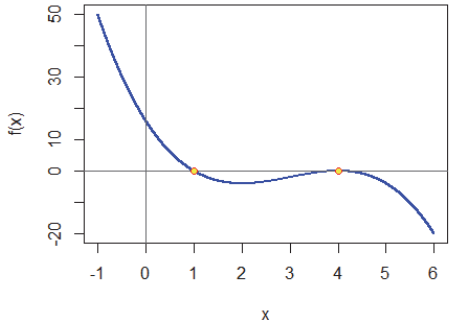


Рисунок 46

с результатом:

```
> uniroot(f,c(a,b),tol=dx)    # Полная информация
$root
[1] 1

$f.root
[1] -1.543515e-09

$iter
[1] 8

$init.it
[1] NA

$estim.prec
[1] 5.5e-08
```

Однако со вторым корнем ничего хорошего здесь не получится:

```
# Нахождение корня f(x) в окрестности [3, 5]
dx <- 0.00000011            # Шаг dx (точность определения
  # точки корня)
```

```
a <- 3; b <- 5                    # Задаем концы интер-
  # вала, где ищем корень
uniroot(f,c(a,b),tol=dx)          # Полная информация
```

с результатом:

```
    > uniroot(f,c(a,b),tol=dx)    # Полная инфор-
мация
Error in uniroot(f, c(a, b), tol = dx) :
  f() values at end points not of opposite sign
```

(дословно: значения функции на концах не противоположны по знаку).

Совершенно ясно, что это произошло из-за кратности корня, а еще точнее – из-за четной кратности корня.

Но мы можем поступить в таком случае уже известным нам «ломовым» приемом из первых примеров, правда потратив на вычисления на несколько секунд больше:

```
dx <- 0.00000011                  # Шаг dx (точность опре-
  # деления точки корня)
x <- seq(3, 5, by = dx)           # Выделили разбиение
  # окрестности первого корня
i <- which.min(abs(f(x)))         # Нашли номер мини-
  # мальной точки у |f(x)| на нашем отрезке
x[i]                              # Точка локального ми-
  # нимума |f|, т.е. корня для f
f(x[i])                           # Точность (насколько
  # значение f(x) отличается от нуля)
```

с отличным результатом:

```
> x[i]              # Точка локального минимума |f|,
т.е. корня для f
[1] 4
> f(x[i])             # Точность (насколько значение
f(x) отличается от нуля)
[1] -3e-16
```

Библиотека rootSolve

Еще одним вариантом решения данной проблемы является использование процедуры uniroot.all из библиотеки library(rootSolve) (только не забудьте предварительно загрузить rootSolve из репозитория):

```
library(rootSolve)          # Активируем библиотеку
uniroot.all(f,c(3,5))       # Находим корень на отрезке [3,5]
```

с результатом:

```
> uniroot.all(f,c(3,5))
[1] 4
```

Кстати, у этой процедуры есть еще один полезный параметр при вызове: n =..., который равен числу подразбиений отрезка, что позволяет поймать множество простых (некратных) корней. Например, для функции

$$g(x)=x(x-1)(x-2)(x-3)(x-4)$$

следующая процедура найдет все корни:

```
g <- function(x) {x*(x-1)*(x-2)*(x-3)*(x-4)}
uniroot.all(g,c(-2,7), n=10, tol= dx)
# n=10 - это на сколько частей подразбиваем отрезок
```

с результатом:

```
> g <- function(x) {x*(x-1)*(x-2)*(x-3)*(x-4)}
> uniroot.all(g,c(-2,7), n=10, tol= dx)  # n=10 -
это на сколько частей подразбиваем отрезок
[1]  -8.592321e-07    1.000000e+00    2.000006e+00
3.000006e+00  3.999971e+00
```

А для такой функции g:

$$g(x)=x(x-1)(x-2)(x-3)(x-4)^2$$

```
g <- function(x) {x*(x-1)*(x-2)*(x-3)*(x-4)^2}
uniroot.all(g,c(-2,7), n=10, tol= dx)  # n=10 - это на сколь-
  # ко частей подразбиваем отрезок
```

uniroot.all (в версиях более ранних, чем R 3.4.2) не заметит последний корень из-за его четной кратности:

```
> g <- function(x) {x*(x-1)*(x-2)*(x-3)*(x-4)^2}
> uniroot.all(g,c(-2,7), n=10, tol= dx)  # n=10 - это на сколько частей подразбиваем отрезок
[1]  -4.007415e-07    9.999810e-01    2.000001e+00
3.000002e+00
```

но если установлен R версии 3.4.2 и выше, то уже все корни будут определены корректно.

Процедура поиска экстремума функции: optimize

Как и в предыдущем пункте, специальные математические численные процедуры не могут применяться формально. Обратим внимание на особенности поиска локальных экстремумов с помощью optimize.

Задание 5. Для функции

$$f(x)=x(x-3)(x-5)$$

построить график в окрестности локальных экстремумов, отметив их маркерами, и приближенно найти их с помощью процедуры optimize.

Решение. Корнями данного кубического многочлена являются 0, 3 и 5, следовательно, точки локальных экстремумов (один максимум и один минимум) располагаются между этими корнями. Выберем чуть больший отрезок $[-1;6]$ и получим:

```
f <- function(x) {x*(x-3)*(x-5)}          # Задаем кубическую
  # параболу
dx <- 0.0001                  # Задаем точность по аргументу x
x <- seq(-1, 6, by = dx)              # Разбиваем отрезок [-1; 6]
  # с шагом dx
plot(x, f(x), type = "l", lwd = 2, col = "blue")
# Рисуем f(x) жирной линией
abline(h = 0, v = 0, col = "gray40")   # Рисуем оси координат
```

```
# Отмечаем точки корней желтым цветом с красной
  # границей:
points(c(4.12,1.2), c(f(4.12),f(1.2)), col = "red", pch = 21,
bg = "yellow")
text(1.2,12,"Max", col = "blue")     # Подписываем точку Max
text(4.1,-8,"min", col = "blue")     # Подписываем точку  min
```

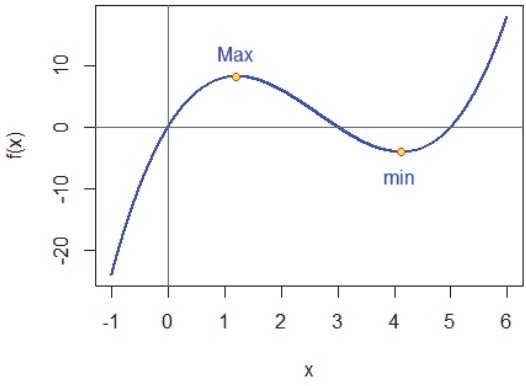


Рисунок 47

Для того, чтобы воспользоваться процедурой optimize необходимо выделить окрестность каждого из локальных экстремумов. Для точки локального максимума Max – это отрезок $[0;2]$ и код программы такой:

```
optimize(f, c(0,2), tol=0.000001, maximum=TRUE)
#  tol - точность
```

с результатом:

```
> optimize(f, c(0,2), tol=0.000001, maximum=TRUE)
# tol - точность
$maximum
[1] 1.2137

$objective
[1] 8.208821
```

При этом первое значение – точка локального максимума $x_{max} = 1.2137$, а второе значение – сам локальный максимум $f_{max} = 8.208821$. Заметим, что если сохранить результат поиска локального экстремума в какой-либо переменной, например, W:

```
W <- optimize(f, c(0,2), tol=0.000001, maximum=TRUE)
# tol - точность
```

то к точке локального максимума x_{max} и самому локальному максимуму f_{max} можно будет обратиться так:

```
W$maximum          # xmax
W$objective        # fmax
```

Для точки локального минимума min визуально выделяем содержащий ее отрезок $[3;5]$. Тогда код процедуры здесь следующий:

```
optimize(f, c(3,5), tol=0.000001)
  # По умолчанию – задача на минимум
  # или
optimize(f,c(3,5), tol=0.000001, maximum=FALSE)
  # Параметры указаны явно
```

с аналогичным по структуре результатом:

```
> optimize(f,c(3,5), tol=0.000001)      # По умолчанию – на минимум
$minimum
[1] 4.119633

$objective
[1] -4.060673
```

Задание выполнено.

А что произойдет, если наш выделенный отрезок не содержит внутри себя локальный экстремум? Тогда, к примеру, локально наибольшим значением на от-

резке фактически будет выступать одно из концевых значений функции. В этом случае процедура optimize вернет одно из концевых значений функции, причем совершенно не обязательно, что самое большее (для поиска локального максимума) или самое меньшее (для минимума).

Убедимся в этом, набрав следующий код:

```
# На отрезке [3,6] нет лок.max, но мы его ищем:
f(3); f(6)                  # Сравниваем значения функции на
  # концах отрезка 3 и 6
optimize(f,c(3,6), tol=0.000001, maximum=TRUE)
# Параметры указаны явно
```

В этом примере все оказалось корректно: optimize определяет наибольшую точку на отрезке:

```
> f(3); f(6)
# Сравниваем значения функции на концах отрезка 3
и 6
[1] 0
[1] 18
> optimize(f,c(3,6), tol=0.000001, maximum=TRUE)
# Параметры указаны явно
$maximum
[1] 5.999999

$objective
[1] 17.99998
```

Но в следующем примере optimize выделяет меньшую точку на конце отрезка

```
# На отрезке [2,6] нет лок.max, но мы его ищем:
f(2); f(6)    # Сравниваем значения функции на концах 2 и 6
# optimize(f,c(2,6), tol=0.000001, maximum=TRUE)
# Параметры указаны явно
```

```
> f(2); f(6)                    # Сравниваем значения
функции на концах отрезка 2 и 6
[1] 6
[1] 18
> optimize(f,c(2,6), tol=0.000001, maximum=TRUE)
# Параметры указаны явно
$maximum
[1] 2

$objective
[1] 5.999998
```

Происходит это потому, что используемый алгоритм поиска локального экстремума предполагает движение к искомой точке по касательной, проведенной к графику в некоторой произвольно выбранной точке. Алгоритм optimize выбирает в качестве первой итерации середину отрезка. В случае отрезка [3, 6] касательная возрастает и уводит алгоритм в правую границу отрезка: точку $x = 6$. В случае отрезка [2, 6] его центр смещается левее, «серединная» касательная убывает (т.е. возрастает влево) и уводит поиск максимума в левую границу отрезка $x = 2$ (см. рис. 48).

Замечание. Код процедуры, достраивающей пунктирные прямые:

```
x <- seq(3, 6, by = dx)
lines(x,6*x-31,col = "brown", lty = "44")
x <- seq(2, 5, by = dx)
lines(x,-4*x+11,col = "brown", lty = "44")
```

Таким образом, использовать optimize необходимо аккуратно, помня, что то, что она возвращает в качестве ответа может оказаться, действительно, локальным экстремумом, либо одним из концевых значений функции в случае ее монотонности. Всегда контролируйте процесс по графику или аналитическим представлениям о поведении функции.

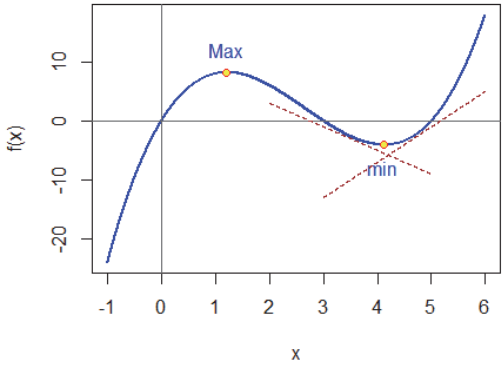


Рисунок 48

Процедура поиска экстремума функции: nlm

В литературе по R можно увидеть ссылки на модифицированный Ньютоновский алгоритм поиска локального минимума функции, реализованный в процедуре nlm. Для поиска максимума с помощью nlm просто домножают функцию на (−1).

Не вдаваясь в подробности, отметим, что использование данного метода требует первоначального выбора точки старта (подобно рассмотренной выше начальной касательной). Удивительно, но даже близкие к экстремумам значения совершенно не гарантируют сходимости метода. Прокомментируем это на нашем примере (см. рис. 49).

Определим точку локального минимума, выбрав старт в точке 4.5:

```
nlm(f, 4.5)    # Поиск минимума функции f со стартом в 4.5
```

с результатом:

```
> nlm(f,4.5)
$minimum
[1] -4.060673
```

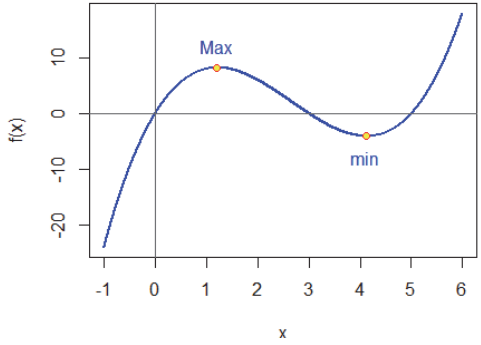


Рисунок 49

```
$estimate
[1] 4.119633

$gradient
[1] -5.394203e-07

$code
[1] 1

$iterations
[1] 2
```

Здесь немного по-другому: первое выводимое значение – это f_{min}, второе – x_{min}, третье – значение производной (должно быть близко к нулю), четвертое – код ошибки (1 – ошибки нет) и количество итераций в методе Ньютона.

Как видим, все в порядке, локальный минимум найден корректно. Однако, если выбрать начальное приближение чуть большим, например, 4.6:

```
nlm(f, 4.6)     # Поиск минимума функции f со стартом в 4.6
```

то неожиданно получим сваливание на левую ветвь функции фактически в минус бесконечность с кодом соответствующей ошибки:

```
> nlm(f,4.6)
$minimum
[1] -1.480914e+13

$estimate
[1] -24554.41

$gradient
[1] 1809147957

$code
[1] 5

$iterations
[1] 8
```

В заключении отметим, что прямой перебор значений функции в окрестности этой точки:

```
dx <- 0.00000011          # Шаг dx (точность определения
  # точки корня)
x <- seq(3, 5, by = dx)   # Выделили разбиение окрестно-
  # сти первого корня
i <- which.min(f(x))      # Нашли номер минимальной
  # точки на нашем отрезке
x[i]                      # Точка локального минимума
f(x[i])                   # Значение локального минимума
```

дал бы отличный результат:

```
> x[i]                  # Точка локального минимума
[1] 4.119633
> f(x[i])               # Значение локального минимума
[1] -4.060673
```

Задания для самостоятельной работы

1. Для функции

$$f(x)=x^3-6x+9$$

a) Построить график в окрестности нулей;

b) Нанести маркерами и подписать буквами корни функции, ее локальные экстремумы и точки перегиба;

c) Вычислить приближенно координаты всех изображенных точек;

d) Сравнить результаты с теоретически точными, получив их аналитически.

2. Для функции

$$f(x)=\frac{1.3x^4-3x^3+2x-0.2}{x^4+3}$$

a) Построить график в окрестности локальных экстремумов;

b) Нанести маркерами и подписать буквами корни функции и ее локальные экстремумы;

c) Вычислить приближенно координаты всех изображенных точек;

d) Нанести на рисунок пунктирной линией асимптоту графика.

3. На отрезке $[-3.5;3.5]$ приближенно найти корни, точки локальных экстремумов и точки перегиба для функции

$$f(x)=x^4-2x^3-8x^2+18x-9.$$

4. Написать код процедуры, изображающей следующий рисунок:

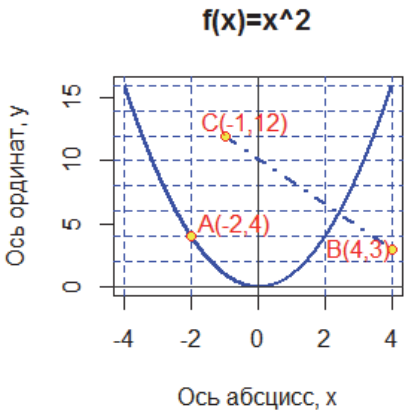


Рисунок 50

Практикум 4. Численное нахождение определенного и несобственного интеграла в R (RStudio)

Как обычно, запустим RStudio и создадим новый документ с будущим кодом по сочетанию клавиш Ctrl+Shift+N.

Приближенное вычисление определенных интегралов в R

Рассмотрим следующий пример вычисления определенного интеграла.

Задание 1. Вычислить определенный интеграл

$$\int_{o}^{\ln 4} e^{-x} dx,$$

используя соответствующие процедуры приближенного решения в среде R.

Решение. Первое, что нам требуется сделать, – это объявить в R подынтегральную функцию

$$f(x) = e^{-x}$$

с помощью уже знакомого кода:

```
f <-function(x) {exp(-x)}   # Объявление функции f(x)=exp(-x)
```

Не забываем после набора каждой строки нажимать Ctrl+Enter, чтобы функция была обработана компилятором R.

Далее нам требуется вызвать специальную базовую процедуру численного интегрирования в R по правилу:

```
integrate(f, 0, log(4))          # Вычисление определенного
  # интеграла от f(x) по области [0, ln4]
```

или в развернутом виде:

```
integrate(f=f, lower = 0, upper = log(4))      # То же самое
```

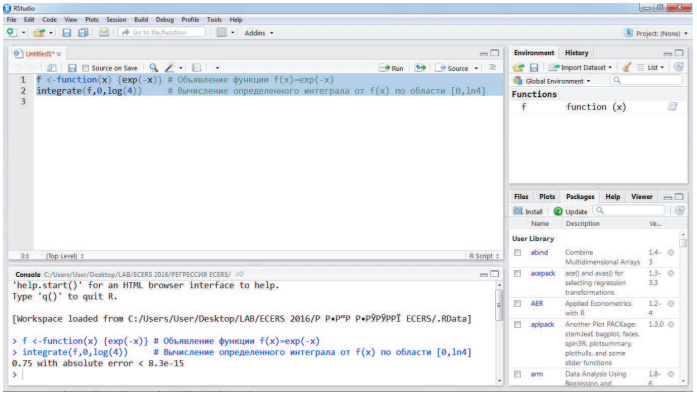


Рисунок 51

Обратим внимание, что ответ численного интегрирования в R выводится с автоматическим указанием ошибки расчетов. В данном примере она составила менее 8.3×10^{-15}. Вывод функции integrate, строго говоря, не является числом – мы видим целую фразу:

```
0.75 with absolute error < 8.3e-15,
```

в которой указан и ответ, и точность. Однако это не всегда бывает удобным.

Если мы хотим использовать полученное значение интеграла для дальнейших вычислений, то необ-

ходимо дополнительно указать поле $value, хранящее в себе само значение определенного интеграла:

```
integrate(f, 0, log(4))$value          # Значение определен-
  # ного интеграла от f(x) по области [0, ln4]
```

Вообще, функция вычисления определенного интеграла возвращает довольно сложный объект типа list, по которому доступны следующие поля значений:

```
integrate(f, 0, log(4))$value          # Значение определен-
  # ного интеграла ...
integrate(f, 0, log(4))$abs.error      # Оценка модуля абсо-
  # лютной погрешности
integrate(f, 0, log(4))$message        # OK или сообщения
  # об ошибках
```

и другие *(подробная информация о классе integrate* доступна по клавише *F1)* (см. рис. 52).

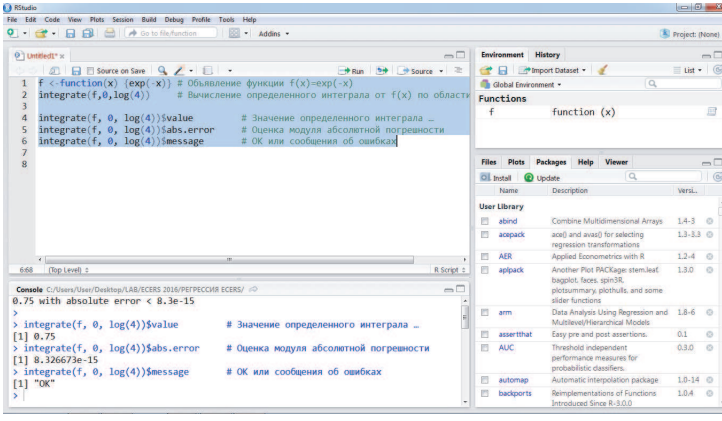


Рисунок 52

Задание 2. Вычислить определенный интеграл

$$\int_1^2 \ln x dx,$$

точно и приближенно. Сравнить оценку модуля абсолютной ошибки в R с реальным расхождением ответов.

Решение. Точное (аналитическое) вычисление определенного интеграла от $\ln x$ получается применением формулы интегрирования по частям. Последовательность преобразований здесь такова:

$$\int_1^2 \ln x dx = x\ln x\Big|_1^2 - \int_1^2 x d(\ln x) = x\ln x\Big|_1^2 - \int_1^2 x\frac{1}{x}dx =$$

$$= \Big(x\ln x - x\Big)\Big|_1^2 = 2\ln 2 - 2 - (1\ln 1 - 1) = 2\ln 2 - 1.$$

Теперь составим программный код, вычисляющий точное и приближенное значение интеграла в R:

```
g<-function(x) {log(x)}              # Объявление функции
  # g(x)=ln(x)
Integral_exactly <- 2*log(2) - 1     # Точное значение интеграла ...
Integral_numerical <- integrate(g, 1, 2)$value
  # Приближенное значение интеграла ...
```

Здесь мы ввели подынтегральную функцию натурального логарифма g(x); образовали новую переменную Integral_exactly, в которую поместили точное значение интеграла; образовали новую переменную Integral_numerical, которой присвоили приближенное значение нашего интеграла.

Теперь остается образовать модуль разности полученных значений интеграла и сравнить его с выдаваемой в R оценкой ошибки приближения (см. также рис. 53):

```
abs(Integral_exactly - Integral_numerical)        # Модуль
  # истинной ошибки
integrate(g, 1, 2)$abs.error                      # Оценка
  # модуля абсолютной погрешности
```

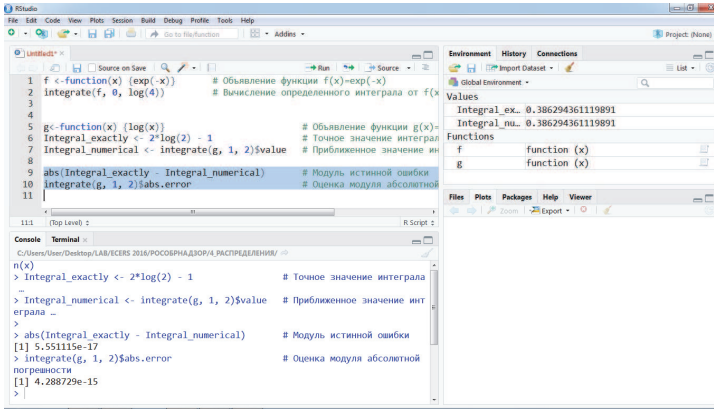


Рисунок 53

Действительно, обнаруживаем, что фактическое отклонение ответов даже на два порядка меньше, чем оценочное.

Замечание. Строго говоря, мы немного лукавим, когда сравниваем с «точным» ответом $2\ln 2 - 1$, т.к. число $\ln 2$ – иррациональное и само по себе в виде десятичной дроби может быть представлено лишь с некоторой точностью. Впрочем, сути обсуждаемых отношений между точным и приближенным значениями определенных интегралов это не меняет.

Приближенное вычисление несобственных интегралов в R

До сих пор при численном вычислении определенных интегралов в R мы особо не обращали внимание на подынтегральную функцию в вопросе ограниченности на области интегрирования. Например, несобственный интеграл

$$\int_0^1 \frac{1}{2\sqrt{x}} dx$$

вполне мог показаться нам определенным и по «предельному» действию формулы Ньютона-Лебница:

$$\int_0^1 \frac{1}{2\sqrt{x}}dx = \sqrt{x}\Big|_0^1 = \sqrt{1} - \sqrt{0} = 1,$$

и по семантике численного взятия определенных интегралов в R (см. рис. 54):

```
h <-function(x) {1/(2*x^0.5)}                    # Объявление функции
  # h(x)=1/(2*x^0.5)

integrate(h, 0, 1)$value                         # Приближенное зна-
  # чение интеграла ...
```

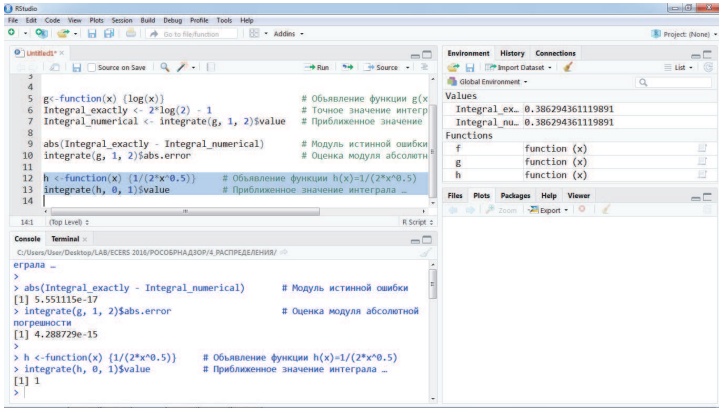


Рисунок 54

Мы и в правду получили верный ответ, но только не для определенного интеграла, а для несобственного, т.к. подынтегральная функция

$$h(x) = \frac{1}{2\sqrt{x}}$$

обращается в бесконечность на нижнем конце интервала интегрирования

$$x = 0.$$

Вывод прост: классу integrate в R безразличен тип интеграла: определенный или несобственный. Однако, в случаях с несобственными интегралами требуется действовать на порядок более внимательно, т.к. алгоритмы численной оценки в R вынуждены заменять бесконечные пределы интегрирования конечными пределами и, если мы не выделим особую точку на одном из концов отрезка интегрирования, то R может просто не заметить особенности или их сократить друг с другом и выдать достаточно отвлеченный результат.

Задание 3. Вычислить несобственный интеграл

$$\int_{-1}^{1} \frac{1}{x} dx.$$

Решение. Совершенной ошибкой будет составить такой, естественный на первый взгляд, код в R:

```
W<-function(x) {1/x}              # Объявление функции
  # W(x)=1/x
integrate(W, -1, 1)       # Приближенное значение интегра-
  # ла ...
```

Сообщение об ошибке (неограниченности функции) в этом случае выдаст и R (см. рис. 55).

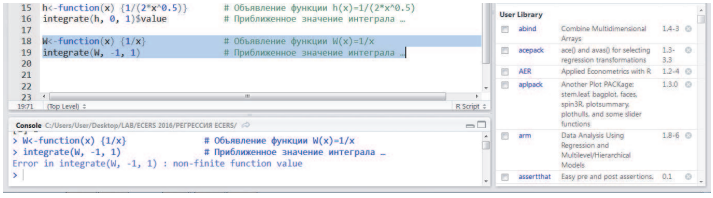


Рисунок 55

Кстати, это уже является некоторым симптомом к расходимости интеграла.

Правильным же здесь было бы изначально представить несобственный интеграл в виде:

$$\int_{-1}^{1}\frac{1}{x}dx=\int_{-1}^{0}\frac{1}{x}dx+\int_{0}^{1}\frac{1}{x}dx,$$

в котором особая точка $x=0$ *(где функция обращается в бесконечность)* выделена в качестве граничной точки обоих интегралов. Напомним, что несобственный интеграл в этом случае будет сходиться, только если оба интеграла в правой части сходятся. Вычисление, например, первого из интегралов

```
integrate(W, -1, 0)     # Приближенное значение интеграла
...
```

приводит к отказу алгоритма из-за достижения предела в возможном количестве подразбиений для удержания точности вычислений. Иными словами, такое сообщение – повод для серьезных раздумий по поводу сходимости несобственного интеграла. В нашем случае, конечно, интеграл расходится

$$\int_{-1}^{0}\frac{1}{x}dx=\ln|x|\Big|_{-1}^{0}=\ln 0-\ln 1=-\infty.$$

Замечание. Если подстановки в формулу Ньютона-Лейбница при корректном выделении особой точки не приводит к неопределенности, то результат ее совпадает с точным значением несобственного интеграла

Задание 4. Вычислить несобственный интеграл

$$\int_{0}^{+\infty}\frac{1}{1+x^2}dx.$$

Решение. В этом случае «несобственность» интеграла обусловлена неограниченностью функции из-за неограниченности самой области интегрирования. Фактически здесь особой точкой является символ $+\infty$. В языке R этот символ имеет представление как «Inf» (см. также рис. 56):

```
S <-function(x) {1/(1+x^2)}                # Объявление функ-
  # ции S(x)= 1/(1+x^2)
integrate(S, 0, Inf)                       # Приближенное зна-
  # чение интеграла от 0 до +Inf
```

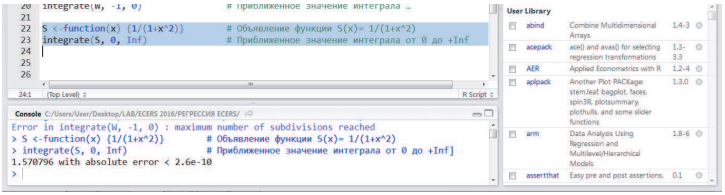


Рисунок 56

Сравните этот ответ с теоретическим

$$\int_0^{+\infty} \frac{1}{1+x^2}dx = \operatorname{arctg}x\Big|_0^{+\infty} = \operatorname{arctg}(+\infty) - \operatorname{arctg}0 = \frac{\pi}{2}.$$

Задание 5. Вычислить несобственный интеграл

$$\int_1^{+\infty} \frac{1}{\sin x + \sqrt{x}}dx.$$

Решение. Абсолютно аналогично предыдущему примеру имеем:

```
V <-function(x) {1/(sin(x)+x^0.5)}
# Объявление функции V(x)= 1/(sin(x)+x^0.5)
```

```
integrate(V, 1, Inf)
# Приближенное значение интеграла от 1 до +Inf
```

После компиляции этого кода мы увидим почти явное указание на расходимость интеграла (см. рис. 57).

```
Error in integrate(V, 1, Inf) : the integral is probably divergent
```

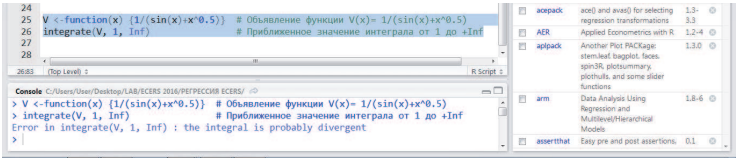


Рисунок 57

Однако, даже в этом случае и, вообще, при любом явном отказе R требуется дополнительное исследование на расходимость интеграла. Здесь рассуждения такие: при $x \to +\infty$

$$\frac{1}{\sin x + \sqrt{x}} \sim \frac{1}{\sqrt{x}};$$

$$\int_1^{+\infty} \frac{1}{\sqrt{x}} dx = 2\sqrt{x}\Big|_1^{+\infty} = 2\sqrt{+\infty} - 2\sqrt{1} = +\infty.$$

Задания для самостоятельной работы

1. Вычислить определенный интеграл

$$\int_0^1 \sqrt[3]{x-2}\,dx,$$

точно и приближенно. Сравнить оценку модуля абсолютной ошибки в R с реальным расхождением ответов. *Замечание.* Внимательно изучите ошибку, выдаваемую R, которая совершенно не связана с вычислением интеграла, и найдите пути ее преодоления! Подсказка:

используйте пользовательскую функцию Radical, описанную в предыдущем практикуме.

2. Вычислить определенный интеграл

$$\int_0^{\pi/2} x^2 \cos x dx,$$

точно и приближенно. Сравнить оценку модуля абсолютной ошибки в R с реальным расхождением ответов.

3. Приближенно вычислить с указанием оценки абсолютной ошибки или доказать расходимость:

$$\int_0^{+\infty} \cos x dx$$

$$\int_0^{+\infty} x^4 e^{-x^2} dx$$

$$\int_1^{+\infty} \frac{\ln x}{x^2} dx$$

$$\int_0^{\pi} \frac{\sin x}{x} dx$$

$$\int_0^4 \frac{dx}{x^3 - x^2}$$

4. Найти геометрическую площадь фигуры, ограниченной

a) параболой $y = 4 - x^2$ и осью абсцисс.

b) функцией $y = \frac{1}{\sqrt[3]{x}}$, прямой $x = 1$ и осью ординат. *Замечание.* Ответ в этом примере не равен 1,5.

Практикум 5. Построение поверхностей и линий уровня в R (RStudio)

Одним из основных преимуществ использования языка R являются его широчайшие возможности графического представления данных. Рассмотрению содержания основных графических библиотек в R можно посвятить не один учебник – настолько обширна программная база, разработанная для этих задач. Мы познакомимся в данном практическом занятии с минимально необходимым набором команд и процедур, которые позволяют легко и наглядно визуализировать функции двух переменных, их линии уровня, а также изображать произвольные поверхности.

Построение графиков функций двух переменных

Задание 1. Построить график гиперболического параболоида на множестве D

$$D = \{(x,y) \in \mathbf{R}^2 \mid -10 \le x \le 10; -10 \le y \le 10\}$$

Решение. Заметим, что множество D является квадратом, а гиперболическим параболоидом называют функцию вида:

$$f(x,y) = x^2 - y^2,$$

график которой напоминает седло. Убедимся в этом, построив эту красивую поверхность.

Базовая идея программирования графического представления поверхностей в R следующая:

1. объявляем функцию $f(x,y)$;

2. разбиваем области изменения переменных $x \in [-10;10]$ и $y \in [-10;10]$ на достаточно мелкие части, т.е. представляем каждую переменную как набор последовательно меняющихся чисел $x_1, x_2, \ldots x_i, \ldots x_n$ и $y_1, y_2, \ldots y_j, \ldots y_m$ от −10 до +10 с каким-либо небольшим шагом (например, 0.5 или мельче);

3. далее во всех полученных всевозможных точках (x_i, y_j) вычисляем значение нашей функции $z_{ij} = f(x_i, y_j)$.

4. и отправляем полученные массивы чисел x, y и z в специальную команду *persp* построения двумерного графика (поверхности) в R:

Код этой небольшой процедуры выглядит так (см. также рис. 58):

```
f <- function(x, y) {x^2 - y^2}     # Объявляем функцию f
x <- seq(-10, 10, by=0.5)           # Задаем последователь-
  # ность значений x
y <- seq(-10, 10, by=0.5)           # Задаем последователь-
  # ность значений y
z <- outer(x, y, f)                 # Вычисляем значение
  # функции во всех парах (x,y)
persp(x, y, z, theta=30, phi=10, col="green",
ticktype="detailed",
    main="Гиперболический параболоид")
# График поверхности f(x,y)
```

Для увеличения рисунка полезно кликнуть левой клавишей мыши по надписи Zoom в левой верхней части окна с графиком. Для копирования рисунка в буфер обмена необходимо кликнуть по следующей за Zoom надписи Export.

В использованной команде *persp* мы указали наиболее важные параметры вызова:

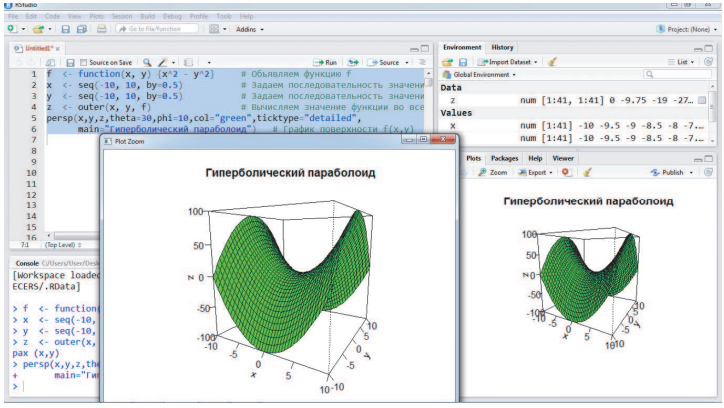


Рисунок 58

theta=30 – угол поворота картинки влево-вправо (*azimuthal direction*) относительно нашего фронтального взгляда на рисунок;

phi=10 – угол поворота картинки вверх-вниз (*colatitude*) относительно нашего фронтального взгляда на рисунок;

col="green" – параметр, отвечающий за выбор цвета поверхности;

ticktype="detailed" – параметр, задающий тип отображения осей

main="Гиперболический параболоид" – отвечает за название к рисунку.

Детальная информация о параметрах вызова процедуры *persp* доступна по универсальной команде вызова справки:

```
?persp          # Вызов справки по persp
```

Замечание. Как бы внимательно мы не смотрели на поверхность гиперболического параболоида нам трудно поверить в то, что она целиком составлена лишь из прямых линий. И тем не менее, это факт.

Линии уровня функции

Задание **2.** Построить график параболоида $f(x,y)=x^2+y^2$ на множестве D

$$D=\{(x,y)\in \mathrm{R}^2 \mid -10\le x\le 10; -10\le y\le 10\}$$

и изобразить его линии уровня.

Решение. Построение поверхности параболоида выглядит аналогично предыдущим рассуждениям. Единственная модификация кода состоит в объединении нескольких операций:

```
x <- y <- seq(-10, 10, by=0.5)          # Задаем последова-
  # тельность значений x и y
z <- outer(x, y, function(x,y) {x^2 + y^2})     # Вычисляем
  # значение функции во всех парах (x,y)
persp(x, y, z, theta=30, phi=25, col="yellow",
ticktype="simple", main="Параболоид z = x^2 + y^2")
# График поверхности f(x,y)
```

Как видим: одинаковые присвоения x и y можно объединить в одну строчку, а функцию $f(x,y)$ можно заявить локально внутри вычислительной процедуры *outer*, не объявляя в R название самой функции.

Обратите внимание на значение параметра **ticktype=«simple»** и связанные с этим изменения рисунка (изображение осей в этом случае достаточно аскетично) (см. рис. 59).

Для выполнения второй части задания (изображение линий уровня функции) воспользуемся встроенной функцией *contour* (контур). Вообще, линией уровня $C=Const$ на плоскости аргументов функции (x,y) называют такое геометрической место точек, для которых значение функции не изменяется, оставаясь равным заданному уровню C, т.е. во всех точках линии уровня C выполняется равенство $f(x,y)=C$. Кстати, для непрерывных функций линия уровня – это непрерывная кривая, которую иногда также называют изолинией функции.

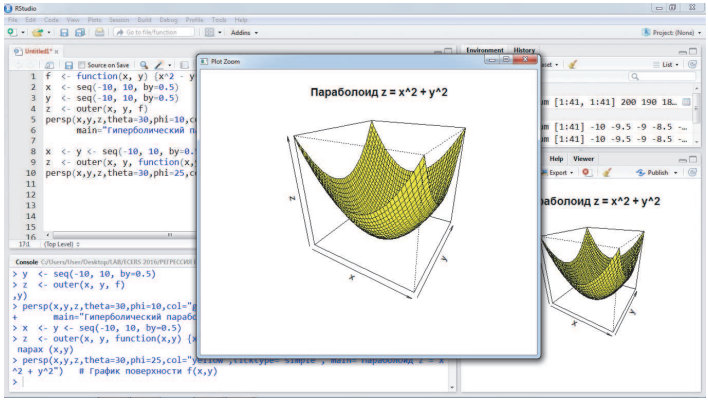


Рисунок 59

Добавка к предыдущему коду здесь такова:

```
contour(x, y, z, nlevels=10, main="Линии уровня f(x,y)=C")
# Линии уровня f(x,y)=Const
```

Параметр **nlevels=10** отвечает за количество отображаемых изолиий, которые в данном случае являются окружностями (см. рис. 60):

$$f(x,y) = x^2 + y^2 = Const\ .$$

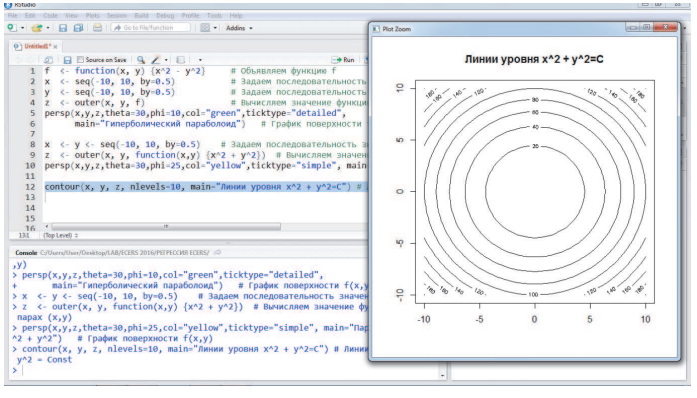


Рисунок 60

Задание 3. Построить линии уровня для функции в окрестности начала координат.

Решение. Первые три строчки кода здесь вполне стандартны:

```
x <- y <-  -50:50                    # x и y задаем как по-
  # следовательности от -50 до 50 с шагом 1
f <- function(x,y) {x^3/8 - y^3/27}  # Задаем функцию f
z <- outer(x, y, f)                  # Вычисляем f во всех
  # точках объявленной сетки (x,y)
```

Здесь мы выбрали в качестве окрестности начала координат квадрат со сторонами по $x \in [-50; 50]$ и по $y \in [-50; 50]$, а разбиение взяли с шагом 1.

Далее введем и отработаем (по сочетанию Ctrl+Enter) отдельно каждую (!) из приводимых ниже модификаций изображения линий уровня.

```
#1 Практически идеально
contour(x,y,z, nlev = 20, method = "edge", vfont = c("sans
serif", "plain"))
#2 Упрощенно, но  с хитрыми подписями осей
contour(x, y, z, ylim = c(-50, 50), method = "simple",
labcex = 1,
     xlab = quote(x[1]), ylab = quote(x[2]))
#3 Ненавязчиво хорошо
contour(x, y, z, ylim = c(-50, 50), nlev = 20, lty = 2, method
= "simple",
     main = "20 levels; \"simple\" labelling method")
#4 Цветовая схема:
image(x, y, z)
contour(x, y, z, col = "pink", add = TRUE, method =
"edge",
     vfont = c("sans serif", "plain"))
```

Внимательно изучите отличия в полученных рисунках и обратите внимание на последнюю цветовую схему изображения линий уровня, содержащую две строки кода (см. рис.61):

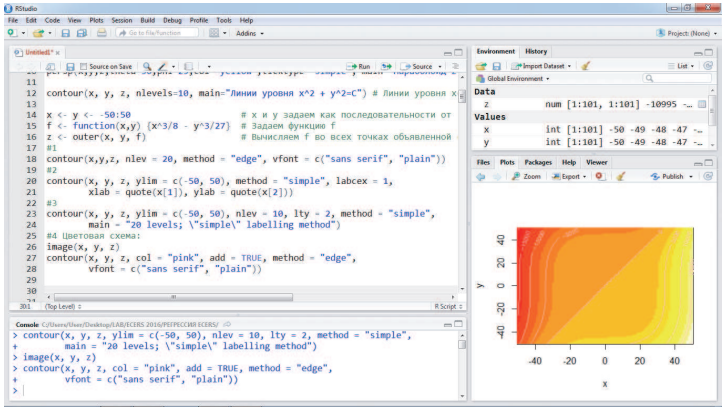


Рисунок 61

Замечание. Укажем еще одну альтернативу цветовому изображению изолиний функции. Абсолютно аналогично вместо последних двух строчек цветовой схемы можно попробовать набрать другую интересную альтернативу

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

Теперь попробуйте самостоятельно разобраться со значениями используемых параметров в *contour*, сравнивая рисунки между собой (см. рис. 62).

Построение поверхностей

Для того, чтобы освоить использование высших достижений визуализации в R, необходимо понять, на первый взгляд, достаточно тонкое различие между графиком функции двух переменных $f(x,y)$ и поверхностью. Это не всегда одно и то же. Конечно, все рассмотренные нами примеры подтверждают, что графиком функции $f(x,y)$ является какая-либо поверхность, и это правильно. Но вот обратное утверждение не всегда верно, т.е. не всякая поверхность соответствует какой-либо двумерной функции $z = f(x,y)$. Это

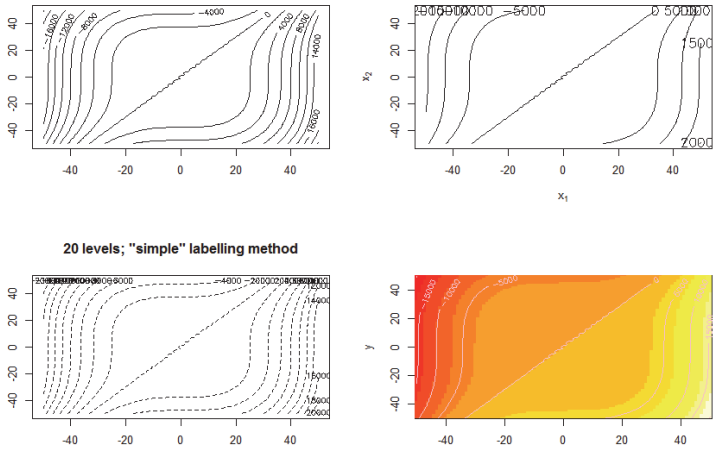


Рисунок 62

связано с тем, что поверхность может иметь самопересечения; неоднозначно соответствующие точкам плоскости (x,y) изгибы; наложения и прочие особенности. Понятие поверхности более ёмкое и вполне конкретное математическое понятие.

Будем представлять себе поверхность как геометрическое место точек в трехмерном евклидовом пространстве $\mathbf{R}^3$, координаты которых (x,y,z) задаются тремя кусочно-непрерывными функциями двух переменных (u,v):

$$\begin{cases} x = x(u,v), \\ y = y(u,v), \\ z = z(u,v). \end{cases} \qquad (1)$$

на некотором множестве изменения переменных $(u,v) \in G \subset \mathbf{R}^2$. Такой способ задания поверхности часто называют параметрическим, а сами переменные (u,v) – параметрами.

Совершенно ясно, что любая двумерная функция $z = f(x,y)$ определяет собой поверхность, если в каче-

стве параметров (u,v) использовать сами переменные (x,y), т.е.

$$\begin{cases} x = u, \\ y = v, \\ z = f(u,v). \end{cases} \qquad (2)$$

Если эта «продвинутая» идея представления поверхностей нам понятна, то мы можем использовать для построения графиков двумерных функций более совершенные инструменты в R.

Задание 4. Построить график функции $f(x,y) = (x^2 + y^2)(2 + \sin x)$ в окрестности начала координат с использованием пакета plot3D.

Решение. В первую очередь однократно подгрузим пакет plot3D с помощью команды

```
install.packages("plot3D")
```

и активизируем его командой

```
require(plot3D)              # То же, что и library(plot3D), но
  # активно в пределах этого листа
```

Далее основная идея здесь следующая:

1. Ввести на множестве параметров $(u,v) \in G \subset \mathbf{R}^2$ прямоугольную сеть (u_i, v_j), подобно тому как мы это делали для (x,y). ;

2. Описать (x,y,z) через функции параметров для всех пар (u_i, v_j);

3. Вызвать специальную команду цветового 3D построения поверхности, обязательно указав в специальном параметре colvar значения какой из координат (x,y,z) нужно отображать цветом.

Введем следующий код на R, реализующий данные шаги:

```
f <- function(x,y) {(x^2 + y^2)*(2 + sin(x))}     # Объявляем
  # функцию f(x,y)
```

```
M <- mesh(seq(-5, 5, length.out = 100), seq(-5, 5, length.
out = 100))                          # Разбиваем сеть (ui, vj)
u <- M$x     # Вводим разбиение параметра u (это набор ui)
v <- M$y     # Вводим разбиение параметра v (это набор vj)

x <- u     # Вводим функцию для координаты x поверхности
y <- v     # Вводим функцию для координаты y поверхности
z <- f(u,v)       # Вводим функцию для координаты z поверх-
  # ности

surf3D(x, y, z, colvar = z, phi = 20, bty = "b2",theta = 50,
    lighting = TRUE, ltheta = 40, colkey = TRUE, box =
TRUE)     # Построение поверхности
```

Здесь первая строка обычным образом вводит нашу функцию $f(x,y)$. Вторая строка представляет собой специальную команду *mesh*. В ее аргументе через запятую мы указали две одинаковые последовательности от -5 до $+5$ и содержащие по 100 точек с равным интервалом (в прошлый раз мы указывали вместо этого длину шага «*by = 0.5*»). Эти последовательности и есть наборы параметров u_i и v_j.

В следующих двух строках мы фактически объявляем символы наших параметров (u,v). *Замечание.* Однако, есть существенное отличие от предыдущих примеров. Дело в том, что команда *mesh* присваивает каждому из параметров u <- M$x и v <- M$y не столько свои наборы, сколько всевозможные свои наборы при различных значениях оставшегося параметра. Иными словами, после команды u <- M$x переменная u будет представлять собой двумерный массив значений параметра u во всех точках сети (u_i,v_j). Аналогично с v <- M$y. Но если это сложно для понимания, можно двигаться дальше без особых проблем.

Следующие три строки реализуют систему задания поверхности как графика функции $f(x,y)$ (см. формулу (2)).

Последняя команда собственно и рисует поверхность. Здесь важнейшим является параметр **colvar!** В нашем случае **colvar = z**, что указывает процедуре *surf3D*, что цветом надо выделять значения координаты z, которая отвечает за функцию $f(x,y)$ (см. рис. 63):

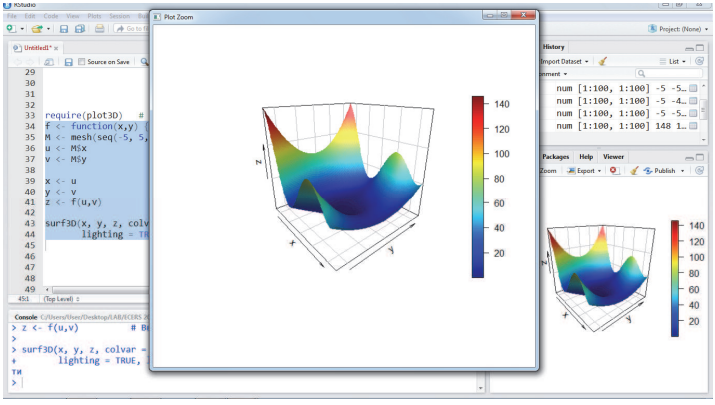


Рисунок 63

Конечно, построение таких удачных ракурсов требует терпения в подборе углов **phi** и **theta**. Попробуйте поизменять остальные параметры, чтобы понять их смысл. Также всегда доступна справка по команде

```
?surf3D
```

Задание 5. Построить график производственной функции Кобба-Дугласа $Q(L,K) = L^{P_1} \cdot K^{P_2}$ для значений параметров $P_1 = 0.85$ и $P_2 = 0.15$ и в окрестности начала координат с использованием пакета plot3D.

Решение. Пакет plot3D нами уже активирован. Перейдем к программированию поверхности для заданной функции. Учитывая, что функция Кобба-Дугласа определена для неотрицательных L и P, выберем разбиение отрезка [0; 50] на 100 точек для каждой из переменных. Тогда получим (см. также рис. 64):

```
M <- mesh(seq(0, 50, length.out = 100), seq(0, 50, length.
out = 100))   # Создаем сеть (ui,vj)
u <- M$x      # Объявляем значения параметра u
v <- M$y      # Объявляем значения параметра v

x <- u    # Вводим функцию для координаты x поверхности
y <- v    # Вводим функцию для координаты y поверхности
z <- x^0.85*y^0.15  # Вводим функцию для координаты z
  # поверхности

# Строим поверхности:
surf3D(x, y, z, colvar = z, phi = 30, bty = "b2",theta = 90,
    lighting = TRUE, ltheta = 40, colkey = TRUE, box =
TRUE)
surf3D(x, y, z, colvar = z, phi = 40, bty = "b2",theta = 90,
    lighting = TRUE, ltheta = 40, colkey = TRUE, box =
TRUE)
surf3D(x, y, z, colvar = z, phi = 50, bty = "b2",theta = 90,
    lighting = TRUE, ltheta = 40, colkey = TRUE, box =
TRUE)
```

Последовательно рассмотрите все три варианта ракурсов поверхностей, чтобы принять наилучший.

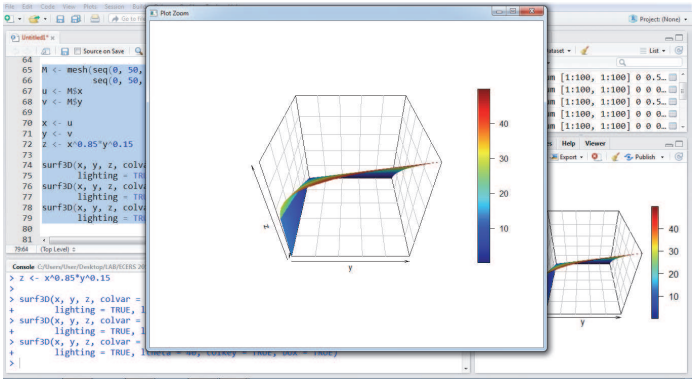


Рисунок 64

Задание 6. Построить график параболоида на множестве

$$D = \{(x,y) \in \mathbf{R}^2 \mid x^2 + y^2 \le 25^2\}.$$

Решение. Область D представляет собой круг радиуса 25 с центром в начале координат. Такие области удобно описывать переходя к полярным координатам $(\rho;\varphi)$ на плоскости (x,y). При этом

$$\begin{cases} x = \rho\cos\varphi \\ y = \rho\sin\varphi \end{cases}$$

Теперь необходимо понять: для того, чтобы был описан круг радиуса 25, необходимо, чтобы $0 \le \rho \le 25$ и $0 \le \varphi \le 2\pi$. Таким образом, разбиением для ρ будет seq(0, 25, length.out = 100), а для φ – seq(0, 2*pi, length.out = 100).

Сама функция $f(x,y)$ в новых переменных будет равна

$$f(\rho,\varphi) = (\rho\cos\varphi)^2 + (\rho\sin\varphi)^2 = \rho^2$$

Тогда в итоге получим (см. также рис. 65):

```
M <- mesh(seq(0, 50, length.out = 100), seq(0, 2*pi, length.
out = 100)) # Задаем сеть парам-ов
r <- M$x    # Объявляем имена параметров
fi <- M$y   #...

x <- r*cos(fi)  # Формулы связи координаты x с параме-
  # трами полярной системы координат
y <- r*sin(fi)  # Формулы связи координаты y с параме-
  # трами полярной системы координат
z <- r^2        # Заданная функция в полярной системе
  # координат

surf3D(x, y, z, colvar = z, phi = 35, bty = "b2",theta =
60, lighting = TRUE, ltheta = 40, colkey = TRUE, box =
TRUE)       # Поверхность параболоида на круге
```

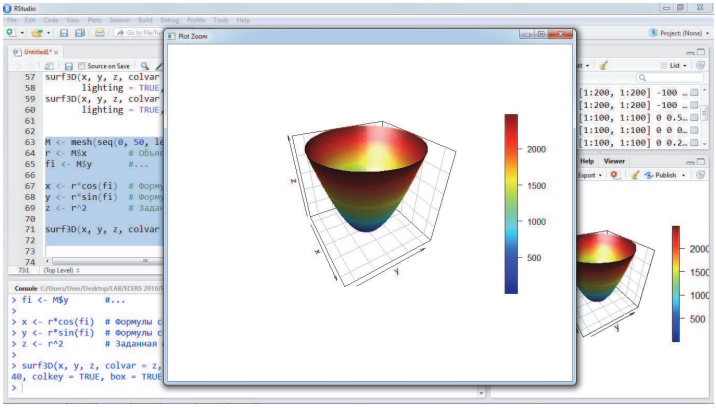


Рисунок 65

Задание 7. Реализовать в R коды двух примеров (воронка и спираль) (см. рис. 66 и 67)

```
# Воронка
M <- mesh(seq(0, 1, length.out = 100), seq(0, 2*pi, length.
out = 100))   # Задаем сеть параметров
r <- M$x      # Объявляем имена параметров
fi <- M$y     #...

x <- r*cos(fi)
y <- r*sin(fi)
z <- sqrt(r)

surf3D(x, y, z, colvar = z, phi = 35, bty = "b2",theta =
60, lighting = TRUE, ltheta = 40, colkey = TRUE, box =
TRUE)         # Поверхность

# Спираль
M <- mesh(seq(0, 4, length.out = 100), seq(0, 6*pi, length.
out = 100))   # Задаем сеть параметров
r <- M$x      # Объявляем имена параметров
fi <- M$y     #...
x <- r*cos(fi)
```

```
y <- r*sin(fi)
z <- sqrt(r) + fi/3

surf3D(x, y, z, colvar = z, phi = 25, bty = "b2",theta =
60, lighting = TRUE, ltheta = 40, colkey = TRUE, box =
TRUE)        # Поверхность
```

и ответить на вопрос: как выглядят формулы для функций из приведенных примеров в переменных x и y, т.е. требуется записать выражения для $f(x, y)$ воронки и спирали.

Решение.

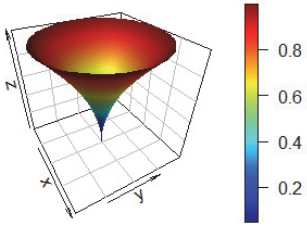


Рисунок 66

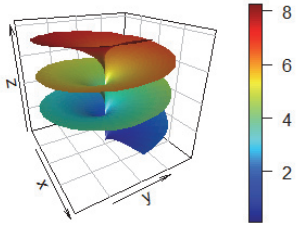


Рисунок 67

Получить формулу для воронки совсем несложно, учитывая что в коде было указано z <- sqrt(r):

$$f(x,y)=\sqrt{\rho}=\sqrt[4]{x^2+y^2}.$$

А вот формулы для спирали в виде однозначной функции $f(x,y)$ не существует, т.к. витки спирали делают поверхность неоднозначно проецируемой на плоскость (x,y). Например, учитывая, что в коде z <- sqrt(r) + fi, половинка одного из витков может быть представлена как

$$f(x,y)=\sqrt{\rho}+\varphi=\sqrt[4]{x^2+y^2}+\operatorname{arctg}\left(\frac{y}{x}\right).$$

Следующая половинка витка будет

$$f(x,y)=\sqrt[4]{x^2+y^2}+\operatorname{arctg}\left(\frac{y}{x}\right)+\pi$$

и т.д.

В заключении советуем отработать следующий ниже код (реализованный в терминах сферической системы координат) и поизучать различные виды отображения поверхностей при разных значениях управляющих параметров (см. рис. 68).

```
require(plot3D)
M <- mesh(seq(0, pi, length.out = 100), seq(0, 2*pi, length.
out = 100))
phi <- M$x
theta <- M$y
r <- sin(4*phi)^7 + cos(2*phi)^3 + sin(6*theta)^2 + cos(6*theta)^4

x <- r * sin(phi) * cos(theta)
y <- r * cos(phi)
z <- r * sin(phi) * sin(theta)

surf3D(x, y, z, colvar = z, phi = 25, bty = "b2",theta = 60,
    lighting = TRUE, ltheta = 40, colkey = TRUE, box = TRUE)

surf3D(x, y, z, colvar = y, colkey = FALSE, shade = 0.5,
    box = FALSE, theta = 60)

surf3D(x, y, z, colvar = y, colkey = FALSE, box = FALSE,
    theta = 60, facets = FALSE)

surf3D(x, y, z, colvar = y, colkey = FALSE, box = FALSE,
    theta = 60, border = "black", xlim = range(x)*0.8,
    ylim = range(y)*0.8, zlim = range(z)*0.8)

surf3D(x, y, z, box = FALSE,
    theta = 60, col = "lightblue", shade = 0.9)
```

Задания для самостоятельной работы

1. Постройте график функции $f(x,y)=ye^{-x^2}$ в квадрате $[-5;5]\times[-5;5]$. *Указание:* используйте процедуру persp. Оформите результат в word.

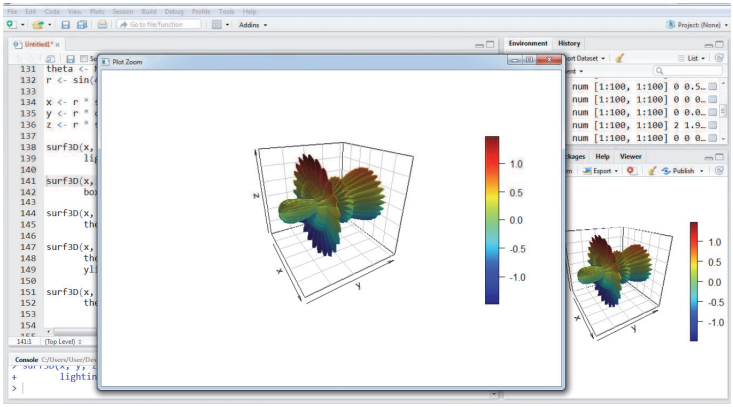


Рисунок 68

2. Постройте линии уровня для производственной функции Кобба-Дугласа из разобранного задания 5. Оформите результат в word.

3. Постройте поверхность, заданную функцией $f(x,y)=x^3-3600x-50y^2$. Выберете несколько удачных ракурсов. Указание: используйте разбиение на отрезках [−100,100] и пакет plot3D. Оформите результат в word.

4. * Придумайте функцию, графиком которой была бы поверхность, похожая на холмистую местность.

5. ** Постройте лист Мёбиуса. Указание: Найти (internet) параметрические формулы, задающие лист Мёбиуса, и реализовать их в пакете plot3D.

6. *** Постройте поверхность шара. Указание: используйте последний пример в заключении со сферической системой координат.

Практикум 6. Символьное дифференцирование в R (RStudio)

В этом пособии изучаются возможностями языка R в оперировании с символьными, т.е. аналитически точными, вычислениями. Будем рассматривать встроенные в R библиотеки, позволяющие точно вычислять производные функции и, связанные с ними, построения градиента и гессиана.

Заметим, что из общих соображений использовать для символьного дифференцирования пользовательские функции в R не представляется возможным, т.к. они, вообще говоря, не обязаны содержать в себе локализованную в единственном выражении формулу, задающую функцию.

Для символьного дифференцирования нам в первую очередь понадобится освоить специальный тип данных в R, представляющий собой аналитическое выражение какой-либо суперпозиции зарезервированных базовых математических функций языка R. Такой тип получил свое название от английского *expression* – выражение.

Тип expression (выражение)

Запустим RStudio, создадим новый файл с будущим кодом программы (Ctrl+Shift+N) и, выбрав

в меню File–Save As..., сохраним его под именем, скажем, Derived.R.

Задание 1. Объявить функцию

$$f(x) = 5x^2 + 8\sin x$$

в качестве выражения (тип *expression*) в R и вычислить значение этого выражения в точке

$$x = 2\,.$$

Решение. Введем следующий код в левом верхнем окне программы

```
f <- expression(5*x^2+8*sin(x))     # Объявляем выраже-
  # ние f(x)
f                                   # Выводим на экран значение f
```

В результате выполнения этого кода в R будет сформирована переменная f, типом которой будет являться выражение – *expression* (см. рис. 69):

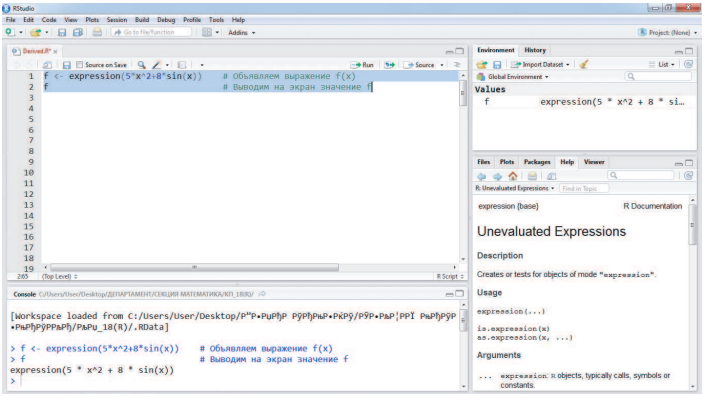


Рисунок 69

Обратите внимание на строку отчета компилятора (внизу):

```
> f                             # Выводим на экран значение f
expression(5 * x^2 + 8 * sin(x))
```

Здесь сообщается, что под именем f заявлено выражение. Как видим, по способу задания *f* не является пользовательской функцией, однако, по сути, такие выражения (*expression*) можно использовать как функции. Например, чтобы вычислить значение нашего выражения *f* в какой-либо точке, необходимо ввести зарезервированную команду:

```
eval(f)                # Вычисляем значение выражения f(x)
```

Однако, если перед этой командой не определить значение аргумента *x*, то мы получим сообщение об ошибке:

```
> eval(f)          # Вычисляем значение выражения f(x)
Error in eval(f) : object 'x' not found
```

Корректным же будет предварительное задание аргумента до вызова функции eval:

```
x <- 2                 # Задаем значение аргумента x
eval(f)                # Вычисляем значение выражения f(x)
```

В итоге получаем правильную последовательность оперирования с выражениями *(expression)* (см. рис. 70):

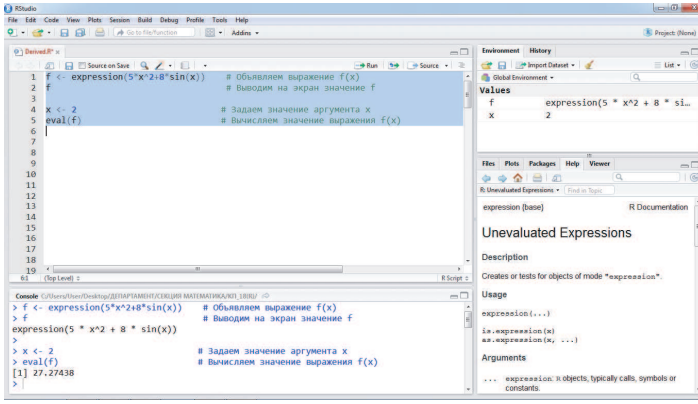


Рисунок 70

Основной оператор символьного дифференцирования D(f, «x»)

Задание 2. Найти точное выражение для производной функции

$$f(x) = 5x^2 + 8\sin x$$

и вычислить ее значение в точке $x = 2$.

Решение. На предыдущем шаге мы уже объявили функцию $f(x)$ как выражение (*expression*) в R. Теперь нам лишь остается применить к ней базовую функцию символьного дифференцирования в R:

```
fx <- D(f, "x")              # Первая производная от выраже-
  # ния f по переменной x
fx
```

Здесь под именем fx будет содержаться выражение для первой производной функции f по переменной x. Если отправить введенную строку на компиляцию (Ctrl+Enter), то в отчете получим:

```
> fx <- D(f, «x»)    # Первая производная от выра-
жения f по переменной x
> fx                 # Выводим на экран выражение
для производной fx
5 * (2 * x) + 8 * cos(x)
```

Переменная fx содержит в себе аналитически точное выражение для производной. Правда, стоит отметить, что тип этого объекта уже не столько *expression*, сколько чуть более общий *language* (т.е. лексическое выражение). Операционно такое расширение типа абсолютно ничего не меняет. Мы так же можем вычислить значение производной в заданной точке (см. также рис. 71):

```
x <- 2                          # Задаем значение аргумента x
eval(fx)                        # Вычисляем значение производной
  # fx в точке x
```

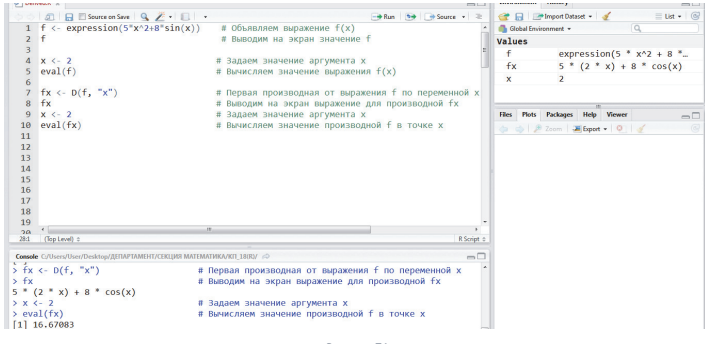


Рисунок 71

Более того, мы можем вычислить вторую производную, применив ту же функцию D(...) к полученной ранее первой производной fx, пользуясь тем, что fx все же относится и к типу *expression*.

Задание 3. Найти точное выражение для второй и третьей производной функции

$$f(x)=5x^2+8\sin x$$

и вычислить их значение в точке $x=2$.

Решение. Нам остается еще несколько раз вызвать функцию D(...) символьного дифференцирования от полученных результатов предшествующих производных (см. также рис. 72):

```
fxx <- D(fx, "x")       # Вторая производная - производная
  # от первой производной выражения f
fxx                     # Выводим на экран выражение для
  # производной fxx
```

```
eval(fxx)                # Вычисляем значение производной
  # fx в точке x
fxxx <- D(fxx, "x")  # Третья производная - производная
  # от второй производной к f
fxxx                     # Выводим на экран выражение для
  # производной fxx
eval(fxxx)               # Вычисляем значение производной
  # fx в точке x
```

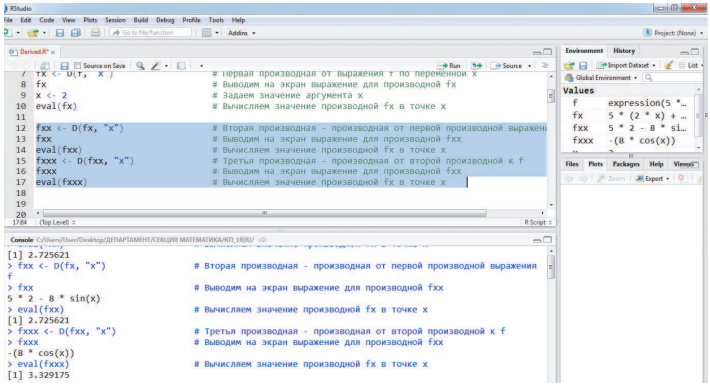


Рисунок 72

Теперь нам ничего не мешает вычислить произвольные частные производные функции нескольких переменных.

Задание 4. Для функции

$$g(x,y,z)=3x^2yz^3+e^x\ln z-\sqrt[3]{y}$$

найти точное выражение для частной производной третьего порядка $\dfrac{\partial^3 g}{\partial x\partial y\partial z}$ и вычислить ее значение в точке $M(-1;2;e)$.

Решение. Объявим выражение (*expression*) для заданной функции g, вызовем последовательно несколько раз «оператор» D(...) дифференцирования в R, зададим точку M и вычислим значение полученной частной производной:

```
g <- expression(3*x^2*y*z^3+exp(x)*log(z)-y^(1/3))  # Объ-
  # являем выражение g(x,y,z)
gx <- D(g, "x")                  # Первая производная от выра-
  # жения g по переменной x
gxy <- D(gx, "y")                # Производная от производной
  # gx по переменной y
gxyz <- D(gxy, "z")              # Производная от производной
  # gxy по переменной z
x <- -1; y <- 2; z <- exp(1)     # Задаем точку M(-1,2,e)
gxyz                             # Выводим на экран выраже-
  # ние для производной gxyz
eval(gxyz)                       # Выводим на экран значение
  # для производной gxyz в точке
```

В результате отправки на компиляцию получаем исчерпывающий ответ (см. рис. 73):

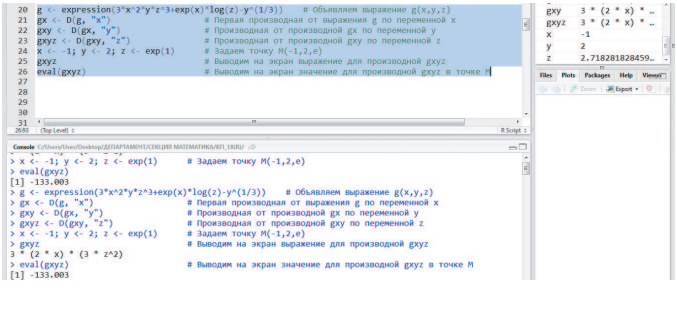


Рисунок 73

Замечание для отличников. Прежде, чем переходить к рассмотрению функций, вычисляющих градиент и гессиан, отметим, что можно изначально

вместо операции *expression* использовать операцию *substitute*, которая будет так же объявлять наши функции выражениями, но только в общем лексическом смысле. Иными словами, следующий код абсолютно эквивалентен предыдущим, но кое в чем даже выигрывает:

```
f <- substitute(5*x^2+8*sin(x))          # Объявляем выра-
  # жение f(x)
fx <- D(f, "x")                  # Первая производная от выра-
  # жения f по переменной x
f                                # Выводим на экран выражение f
fx                               # Выводим на экран выражение
                                   # fx
x <-2                            # Задаем значение аргумента x
eval(f)                          # Выводим на экран значение f
eval(fx)                         # Выводим на экран значение fx
```

Сравните отчеты компилятора между собой в третьей и четвертой строках этого кода и кода, где использовалась бы вместо *substitute* функция *expression*:

```
f <- expression(5*x^2+8*sin(x))          # Объявляем выра-
  # жение f(x)
fx <- D(f, "x")                  # Первая производная от выраже-
  # ния f по переменной x
f                                # Выводим на экран выражение f
fx                               # Выводим на экран выражение fx
x <-2                            # Задаем значение аргумента x
eval(f)                          # Выводим на экран значение f
eval(fx)                         # Выводим на экран значение fx
```

Базовые функции deriv и deriv3

Задание 5. Для функции

$$h(x,y)=x\mathrm{tg}^3 y$$

найти значение градиента в точке $M\left(3;\frac{\pi}{4}\right)$.

Решение. Напомним, что градиентом функции называется вектор, составленный из первых производных функции, т.е. в нашем случае двух переменных $h(x,y)$:

$$\text{grad}h=\left(h'_x;h'_y\right),$$

причем, направление этого вектора (градиента) на плоскости переменных указывает направление максимального роста функции $h(x,y)$, а длина – соответствует значению производной по направлению максимального роста.

Составим следующий код на языке R для вычисления значения градиента в заданной точке:

```
h <- expression(x*(tan(y))^3); h                    # Объявляем
  # выражение h(x,y)
Grad <- deriv(h, c("x","y"), func = TRUE)  # Задаем функ-
  # цию градиента h
Grad(3,pi/4)                                  # Вычисляем h и grad(h)
  # в точке (3; pi/4)
```

и, после запуска на компиляцию, получим вычисленное в заданной точке значение как самой функции, так и координат вектора градиента (см. рис. 74):

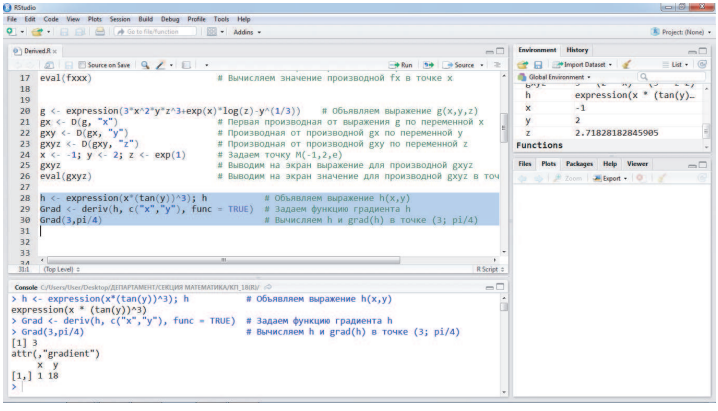


Рисунок 74

Здесь следует отметить, что действие функции *deriv* отличается от рассмотренной выше *D(…)*, и отличается принципиально: если ранее D(f, «x») возвращало нам аналитическое выражение для производной (тип *expression*), то сейчас команда deriv(h, c(«x»,»y»), func = TRUE) возвращает обычную функцию (тип *function*), вычисляющую градиент в точке. Чтобы лучше понять эту разницу, запустим команду:

```
Grad                    # Убеждаемся, что Grad - это функция
```

В результате чего, получим сообщение компилятора о том, что имя Grad – это имя функции с соответствующим кодом:

```
> Grad          # Убеждаемся, что Grad - это функция
function (x, y)
{
    .expr1 <- tan(y)
    .expr2 <- .expr1^3
    .value <- x * .expr2
       .grad <- array(0, c(length(.value), 2L),
list(NULL, c("x", "y")))
    .grad[, "x"] <- .expr2
   .grad[, "y"] <- x * (3 * (1/cos(y)^2 * .expr1^2))
    attr(.value, "gradient") <- .grad
    .value
}
```

Замечание для отличников. Желающие могут проанализировать этот несложный код, имея в виду, что «.expr1», «.expr2», «.value» и «.grad». – всего лишь имена переменных; array(…) – оператор задания массива, а attr(…) – оператор приписывания какого-либо атрибута (смысловых добавок в виде пар «имя»-«значение») к переменной.

Таким образом, команда Grad <- deriv(h, c(«x»,«y»), func = TRUE) задает функцию с именем

Grad, а любое последующее обращение к ней как функции, например, Grad(3, pi/4), вычисляет конкретные значения градиента в указанной точке.

Задание 6. Для функции

$$h(x,y)=x\mathrm{tg}^3 y$$

найти значение гессиана в точке $M\left(3;\frac{\pi}{4}\right)$.

Решение. Для вычисления в заданной точке значения гессиана – матрицы Гёссе, состоящей из частных производных функции второго порядка:

$$hess(h)=\begin{pmatrix} h''_{xx} & h''_{xy} \\ h''_{yx} & h''_{yy} \end{pmatrix},$$

повторим предыдущий код с единственной содержательной заменой команды deriv на deriv3:

```
h <- expression(x*(tan(y))^3); h
  # Объявляем выражение h(x,y)
Hessian  <- deriv3(h, c("x","y"), func = TRUE)
  # Задаем функцию градиента h
Hessian(3,pi/4)
  # Вычисляем h, grad(h) и hess(h) в точке (3; pi/4)
```

В результате отработки этого кода получим всю предыдущую информацию о значении функции и ее градиента, а также дополнительно значения всех элементов матрицы Гёссе (см. рис. 75).

Поясним, что приводимая в R–отчете матрица Гёссе в точке $M\left(3;\frac{\pi}{4}\right)$ имеет вид:

$$hess(h)=\begin{pmatrix} h''_{xx} & h''_{xy} \\ h''_{yx} & h''_{yy} \end{pmatrix}=\begin{pmatrix} 0 & 6 \\ 6 & 108 \end{pmatrix}.$$

Совершенно аналогично действию команды *deriv*, результатом выполнения команды *deriv3* является формирование функции (в нашем случае с именем

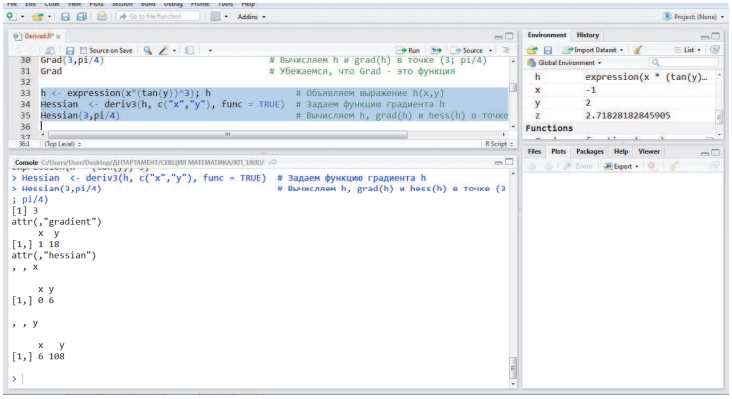


Рисунок 75

Hessian), вычисляющей гессиан в произвольно заданной точке.

Естественным недостатком команд deriv и deriv3, рассмотренных в примерах 5–6, является отсутствие информации о точном аналитическом выражении для градиента и гессиана.

Еще одним ощутимым неудобством является то, что сами значения градиента и гессиана оформлены в виде атрибутов к выводимой информации. Это означает, что увидеть на экране сами значения градиента и гессиана мы можем, а напрямую оперировать с ними – нет. Для того, чтобы можно было непосредственно обращаться к значениям гессиана или градиента, необходимо переписать их из атрибутов в какую-нибудь переменную.

Например, в последней отработанной строке мы вычислили значения функции $h(x,y)$, ее градиента и гессиана в точке $M\left(3;\frac{\pi}{4}\right)$ командой Hessian(3, pi/4). Теперь создадим переменную (скажем, «All»), в которую с помощью специальной команды запишем атрибуты Hessian(3,pi/4) (см. также 76):

```
All <- attributes(Hessian(3,pi/4))          # Записываем
  # в переменную All атрибуты
All       # Выводим на экран полученный результат в All
```

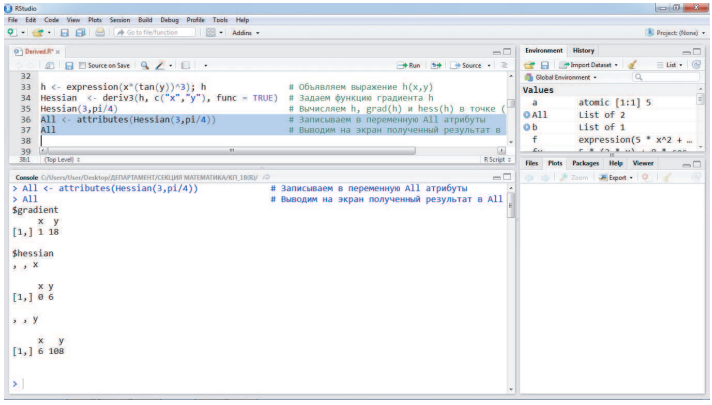


Рисунок 76

Теперь градиент и гессиан доступны по явному обращению All$gradient и All$hessian, к примеру:

```
All$gradient[,"x"]     # Первая координата градиента: h'x
All$hessian["x","y"]      # Смешанная производная h''xy
```

с результатом:

```
> All$gradient[,»x»]                    # Первая координата градиента: h'x
x
1
> All$hessian[,»x»,»y»]                    # Смешанная производная h''xy
[1] 6
```

Универсальная процедура дифференциального исчисления

Задание7* (для отличников) Разработать пользовательскую функцию, возвращающую по заданному выражению $f(x,y)$ (тип *expression*) точные аналитические выражения ее частных производных первого и второго порядка. Оформить код в виде отдельно подгружаемого файла приложения R.

Решение. Не закрывая наш файл «Deriv.R», создадим еще один файл и сохраним его под именем, например, «Symbol_Deriv.R». В этом новом окне мы будем программировать требуемую пользовательскую функцию.

1. Составим код пользовательской функции, формирующий точные формулы производных для заданного выражения:

```
Deriv2D <- function(expr) {          # Объявляем функцию
  # Deriv2D
 fx <- D(expr, "x")                  # Вычисляем f'x
 fy <- D(expr, "y")                  # Вычисляем f'y
 fxx <- D(D(expr, "x"), "x")         # Вычисляем f'xx
 fyy <- D(D(expr, "y"), "y")         # Вычисляем f'yy
 fxy <- D(D(expr, "x"), "y")         # Вычисляем f'xy

# Формируем результат в переменной Rez как список
  # (тип list), содержащий производные:
 Rez <- list(Function = expr, fx = fx, fy = fy, fxx = fxx,
fxy = fxy, fyy = fyy)
 return(Rez)        # Возвращаем результат в виде списка Rez
}
```

Здесь мы ввели название нашей функции «Deriv2D», чьим аргументом объявили переменную «expr». Эта переменная должна принадлежать типу *expression*, чтобы последующие обращения к функции D(...) были корректны.

Теперь, после отправки данного кода на компиляцию, мы можем обратиться к нашей функции следующим образом:

```
# Образец вызова функции Deriv2D:
Deriv2D(expression(x^2 + y*sin(3*x)  - sqrt(x)*y^3))
```

А еще лучше таким образом:

```
# Образец вызова функции Deriv2D:
Deriv2D(substitute(x^2 + y*sin(3*x)  - sqrt(x)*y^3))
```

В результате мы получим точные выражения для производных первого и второго порядка для заявленной функции $x^2 + y\sin(3x) - \sqrt{xy^3}$ (см. рис. 77):

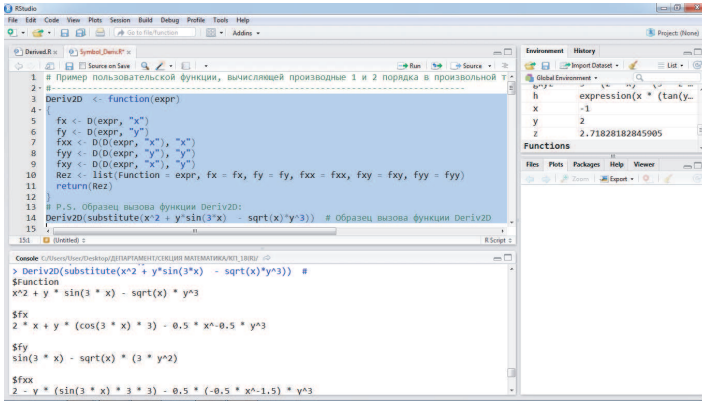


Рисунок 77

2. Теперь сделаем так, чтобы нашу функцию «Deriv2D» можно было вызывать из другого файла.

Если мы сейчас из окна файла «Symbol_Deriv.R» скопируем строчку

```
Deriv2D(substitute(x^2 + y*sin(3*x)  - sqrt(x)*y^3))
# Образец вызова функции Deriv2D
```

в окно файла «Deriv.R», то функция прекрасно сработает. Однако, это случайность, связанная с тем, что

функция Deriv2D в данный момент времени уже загружена в динамическую память R. Давайте для чистоты эксперимента очистим динамическую память R (чтобы не перезагружать RStudio). На следующем рисунке указано на что надо нажать для ее полной очистки:

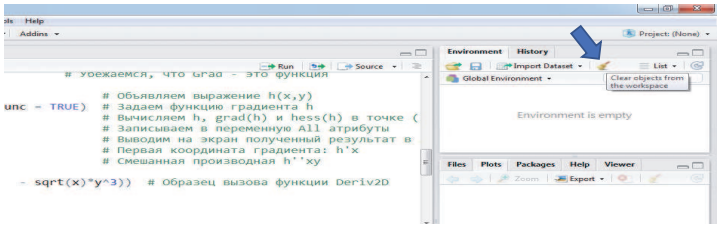


Рисунок 78

После очистки запуск функции «Deriv2D» уже приводит к ошибке:

```
> Deriv2D(substitute(x^2 + y*sin(3*x) - sqrt(x)*y^3)) # Образец вызова функции Deriv2D
Error in Deriv2D(substitute(x^2 + y * sin(3 * x) - sqrt(x) * y^3)) :
  could not find function "Deriv2D"
```

Для подключения какой-либо функции из имеющегося внешнего файла необходимо сделать две вещи: поместить файл с функцией в рабочую директорию R (если не хотим указывать точный путь к файлу) и вызвать специальную команду подключения:

```
source("Symbol_Deriv.R")      # Из рабочей директории R
  # подключаем файл Symbol_Deriv.R
```

Теперь обращение к функции становится возможным (см. рис. 79).

Замечание. Рабочая директория устанавливается по следующей ссылке меню: Tools – Global Options...– General (см. рис. 80).

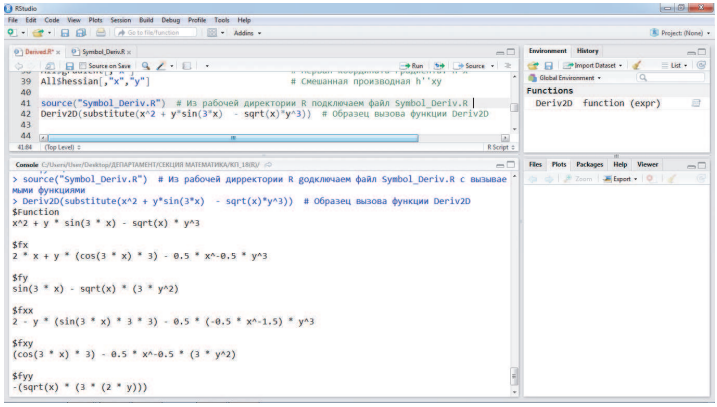


Рисунок 79

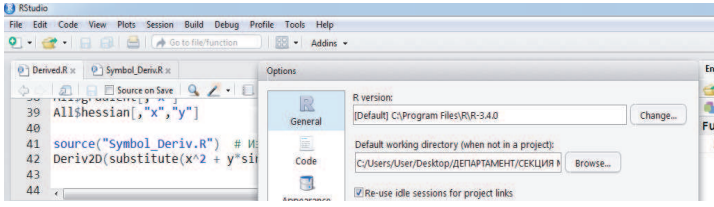


Рисунок 80

Заключение к практикуму 6

Приведем без комментария R–код еще одной полезной функции в дифференциальном исчислении функций многих переменных (см. также рис. 81):

Пример 8.

```
# Пример пользовательской функции, вычисляющей
  # градиент и гессиан в заданной точке для заданного
  # выражения двух переменных
#-------------------------------------------------------------------------
Deriv2Dn  <- function (expr,x=0, y=0)
{
 M <- c(x=x,y=y)
```

```
 value <- eval(expr)
 grad <- c(x=0,y=0)
 grad["x"] <- eval(D(expr, "x"))
 grad["y"] <- eval(D(expr, "y"))
 hess <- array(0, c(2L, 2L), list(c("x", "y"), c("x", "y")))
 hess["x","x"] <- eval(D(D(expr, "x"), "x"))
 hess["y","y"] <- eval(D(D(expr, "y"), "y"))
 hess["x","y"] <- hess["y","x"] <- eval(D(D(expr, "x"), "y"))
 Rez <- list(Function = expr, Point = M, Value_Function= value, Gradient= grad, Hessian = hess)
 return(Rez)
}
# P.S. Образец вызова функции Deriv2Dn:
Deriv2Dn(substitute(x^2 + x * y - y^3), x=1, y=2)
```

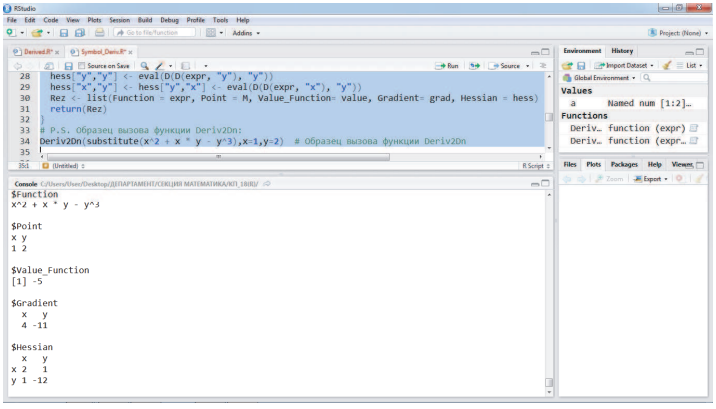


Рисунок 81

Задания для самостоятельной работы

3. Вычислить точные выражения для производных до третьего порядка включительно для функций

а) $f(x) = 3e^{-2x} + \dfrac{\operatorname{tg}\sqrt[3]{x}}{\ln x}$,

b) $f(x)=\dfrac{7x^4+\cos(1-x)}{\arcsin x}$.

4. Вычислить значения производных в точке $x=0.5$ для функций из задания 1.

5. Найти производную по направлению вектора $\vec{l}\left(2,\sqrt{5}\right)$ в точке $M(e;\pi)$ для функции $f(x,y)=\ln\left(x^2+\sin y\right)$.

6. Вычислить значение функции f, ее градиента и гессиана в точке M

a) $f(x,y)=x^{2y}$, $M(e;3)$;

b) $f(x,y,z)=-x^2y^{-3}z^4$, $M(1;-1;2)$;

$f(x,a,b)=\cos(ax)\ln(b+x)$, при $x=2,a=-3,b=e$;

c) $f(v,w)=\left(\dfrac{w}{v}\right)^3$, $A(2;3)$.

7. * Постройте аналогично примеру 7 функцию Deriv3D для функции трех переменных $f(x,y,z)$.

8. * Постройте аналогично примеру 8 функцию Deriv3Dn для функции трех переменных $f(x,y,z)$.

Практикум 7. Типы данных в R (RStudio)

Пришло время систематизировать важнейшие компоненты языка R. В первую очередь подробно разберемся с основными типами данных, хотя ранее некоторые из них нам уже встречались, среди которых были и достаточно специфические типы, например, *expression* или *language*.

Запустим RStudio и создадим новый файл с будущим кодом программы путем нажатия комбинации Ctrl+Shift+Enter.

R – динамически типизированный язык

R, действительно, является динамически типизированным языком. Это утверждение может повергнуть в ужас любого программиста и одновременно вызвать восторг у любого другого пользователя. Связано это вот с чем: во-первых, тип у каждой переменной в R может меняться во время выполнения программы даже против желания пользователя, и во-вторых, тип каждой переменной фактически вообще не объявляется, а наследуется от типа объекта, которому присваивается переменная. То есть в отличие от обычных языков программирования (C++, Pascal,...) в R переменные во время объявления не резервируются в памяти своим типом, а вводятся кодом сразу под видом конкретно заданных объектов (значений), чей тип и наследуют.

Поясним эту интересную и основополагающую особенность языка R на следующем примере.

Задание 1. Объявить переменную *n*, в которую записать число 2.

Решение. Введем следующий код в левом верхнем окне программы:

```
n <- 2        # Объявляем переменную n, равную 2
n             # Выводим на экран текущее значение n
typeof(n)     # Выясняем какому типу относится переменная n
```

В результате выполнения этого кода в R будет сформирована переменная n, типом которой будет являться максимально широкий в данном контексте тип *double* – вещественное число двойной точности (см. рис. 82):

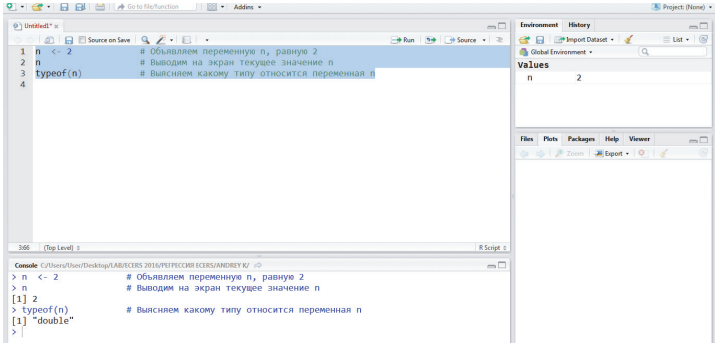


Рисунок 82

Если же мы хотим видеть в переменной n не вещественное число, а исключительно целое, то нам следует переопределить (исправить) тип переменной n (см. также рис. 83):

```
n <- as.integer(n)     # Переопределяем переменную n
  # как целое число
n                      # Выводим на экран текущее значение n
typeof(n)              # Выясняем какому типу относится
  # переменная n
```

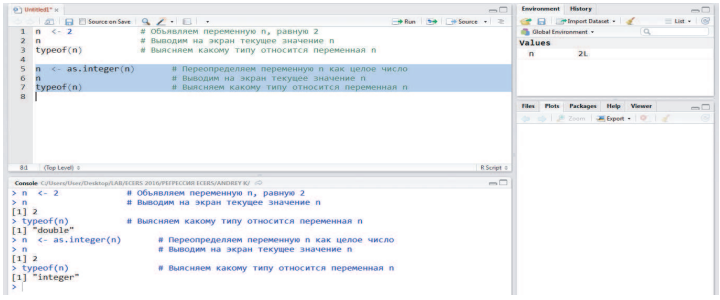


Рисунок 83

Отметим, что такое положение является достаточно удобным. Если нам не важно к какому типу соотнести число 2, то R автоматически определяет его к произвольному вещественному числу, ничем не ограничивая дальнейшее оперирование с ним.

Атомарные данные

Задание 2. Объявить в R переменные основных типов, присвоив им какие-либо подходящие значения.

Решение. Выделим шесть основных типов атомарных данных в R и для каждого из них объявим соответствующую переменную. Мы использовали здесь термин атомарные данные, т.к. помимо них существуют еще и многомерные данные: векторы, массивы, матрицы, списки,... – о них чуть позже.

Логический тип (logical)

Переменные данного типа, их называют еще булевыми, могут принимать только два различных значения: TRUE и FALSE («Правда» и «Ложь»). Если такую булеву переменную переопределить по типу в целую (integer) или вещественную (double), то значение TRUE будет конвертировано в единицу, а FALSE – в ноль.

Замечание. Напомним, что в R различаются заглавные и строчные буквы, а имена TRUE и FALSE являются зарезервированными и изменять их значения не следует, хотя и возможно.

Пример скрипта, объявляющего булеву переменную под именем Flag:

```
Flag <- TRUE          # Объявляем переменную Flag, рав-
  # ную TRUE
Flag                  # Выводим на экран текущее значение Flag
typeof(Flag)          # Выясняем какому типу относится
  # переменная Flag
```

Еще один полезный пример:

```
A <- (2 > 3)          # Объявляем переменную A, равную
  # истинности утверждения: 2>3
A                     # Выводим на экран текущее значение A
typeof(A)    # Выясняем какому типу относится переменная A
```

Консольное сообщение для последнего примера:

```
> A  <- (2 > 3) # Объявляем переменную A, равную
истинности утверждения: 2>3
> A          # Выводим на экран текущее значение A
[1] FALSE
> typeof(A)  # Выясняем какому типу относится пе-
ременная A
[1] «logical»
```

Целочисленный тип (integer)

Переменные данного типа принимают только целые значения. В первом задании мы уже приводили пример как определить целочисленный тип путем переопределения типа double:

```
n <- 2                # Объявляем переменную n, равную 2
n <- as.integer(n)    # Переопределяем переменную n как
  # целое число
```

```
n                    # Выводим на экран текущее значение n
typeof(n)            # Выясняем какому типу относится
  # переменная n
```

Замечание. Конечно, здесь две последние строки в принципе не нужны. Мы приводим их лишь для того, чтобы убедиться в правильности наших действий.

Приведем еще один типичный вариант кода, задающего именно целочисленную переменную:

```
n <- 2L              # Объявляем переменную n, равную
  # 2L – т.е. равную именно целой 2
n                    # Выводим на экран текущее значение n
typeof(n)            # Выясняем какому типу относится
  # переменная n
```

Отчет консоли R:

```
> n <- 2L     # Объявляем переменную n, равную 2L –
т.е. равную именно целой 2
> n          # Выводим на экран текущее значение n
[1] 2
> typeof(n)  # Выясняем какому типу относится пе-
ременная n
[1] «integer»
```

Таким образом, если поставить после целого числа символ «L», то R поймет данное число как целое: 35L – это обычное целое число 35, не удивляйтесь.

Важно! Однако, если попытаться в объявленную ранее целочисленную переменную записать дробное число, то ошибки, как ни странно, не возникнет, т.к. R автоматически переопределит тип нашей переменной, чтобы операция присвоения стала корректной:

```
n <- 2L          # Объявляем целочисленную переменную n,
  # равную 2
typeof(n)    # Выясняем какому типу относится перемен-
  # ная n
n <- 2.3     # Переопределяем переменную n как число 2.3
```

```
typeof(n)            # Выясняем какому типу относится
  # переменная n
```

```
> n <- 2L           # Объявляем целочисленную пере-
менную n, равную 2
> typeof(n)        # Выясняем какому типу относится
переменная n
[1] «integer»
> n  <- 2.3          # Переопределяем переменную n
как число 2.3
> typeof(n)        # Выясняем какому типу относится
переменная n
[1] “double”
```

Вещественный тип (numeric, double)

Переменные данного типа принимают произвольные вещественные значения. Данный тип присваивается автоматически для любых числовых значений как самый широкий и распространенный. Десятичным разделителем в R является точка!

```
n <- log(2)          # Объявляем переменную n, равную ln2
n                    # Выводим на экран текущее значение n
typeof(n)            # Выясняем какому типу относится
  # переменная n
```

Комплексные числа (complex)

Данный тип является уже специальным, расширяющим область вещественных чисел до комплексной плоскости. Здесь принята семантика алгебраической формы комплексного числа, т.е. число $z = x + yi$ в данной форме будет пониматься как комплексное в R, со всеми соответствующими алгебраическими расширениями.

```
z <- 3 + 4i          # Объявляем переменную z, равную 3+4i
z                    # Выводим на экран текущее значение z
typeof(z)            # Выясняем какому типу относится пере-
  # менная z

z^2                  # Степень к.ч. z
Re(z)                # Вещественная часть к.ч. z
Im(z)                # Мнимая часть к.ч. z
Arg(z)               # Аргумент ( φ ) к.ч. z
Mod(z)               # Модуль ( ρ ) к.ч. z
```

Замечание. 1. Обратите внимание, что в выражении z <- 3 + 4i знак умножения между 4 и i не ставится.

Замечание.2. Если образовать несколько комплексных чисел, то их можно складывать, вычитать, умножать и делить естественным образом, используя соответствующие знаки + – * /.

Также здесь будет уместно привести процедуру приближенного вычисления корней многочленов, чьи корни могут быть комплексными.

Следующий ниже код программы численно находит все нули алгебраического уравнения четвертой степени $x^4 - 3x^3 + 3x^2 - 3x + 2 = 0$:

```
# Решение многочленов
a <- c(2,-3,3,-3,1)        # Вектор коэффициентов многочле-
  # на y(x)=2-3x+3x^2-3x^3+1x^4
polyroot(a)                # Нахождение корней многочлена
```

Обратите внимание, что переменная «a» здесь объявлена как вектор, состоящий из пяти коэффициентов многочлена при степенях x, начиная со свободного члена, хотя мы привыкли перечислять коэффициенты наоборот со старшей степени (см. рис. 84).

Из отчета консоли нетрудно видеть, что корнями многочлена являются четыре числа: 1, 2, i и $-i$, причем, эти корни получились абсолютно точными.

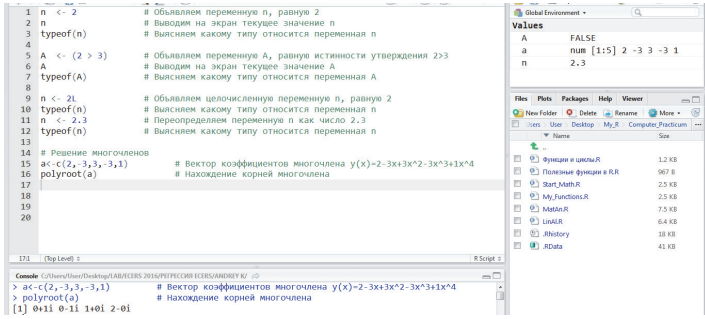


Рисунок 84

R прекрасно справляется и с формой Эйлера $\rho e^{i\varphi}$, только надо аккуратно прописывать мнимую единицу i без символа умножения. Например, для того, чтобы вычислить в обычной алгебраической форме число $z = 3e^{-i\pi/2}$, представленное формой Эйлера, необходимо использовать склейку «1i»:

```
# Форма Эйлера
3*exp(-pi*1i/2)      # Вычисление числа 3e^(-iπ/2)
```

```
> 3*exp(-pi*1i/2)
[1] 0-3i
```

Текстовые/строковые переменные (character)

Переменные данного типа содержат в себе текстовые строки, начиная от одного символа и заканчивая целыми фразами, содержащими несколько слов. Значения таких переменных всегда указываются в кавычках. Возможно также в качестве значения указать и пустые кавычки " ".

```
S <- "I am crazy about R"      # Объявляем текстовую пе-
  # ременную S
S                              # Выводим на экран текущее
```

```
  # значение S
typeof(S)                    # Выясняем какому типу отно-
  # сится переменная S
nchar(S ); n                 # Количество символов в стро-
  # ке S, включая пробелы
paste(S,"!!!", sep = "")     # Операция склейки тексто-
  # вых выражений
```

В последней строке мы привели часто используемую операцию склейки текстовых выражений. В данном примере склейка происходит без пробела, о чем говорит параметр разделения sep=" ". Если в аргумент этой функции paste попадет не текстовое выражение (character), а, например, числовое (numeric), то оно автоматически будет преобразовано в соответствующий текст.

Замечание. Обратите внимание: кавычки в MSWord: «А» не соответствуют кавычкам в R: "A". Имейте это в виду при копировании набранных текстов из word в R. В приведенных здесь примерах все корректно, т.к. скопированы рабочие коды из R.

Здесь также уместно привести операции теории множеств, т.к. объединение, пересечение, разность и принадлежность – активно используются при работе с анализом текстов:

```
# Принадлежность множеству:
is.element("A", c("C","A","D","F"))          # Является
  # ли "A" элементом множества букв
# Объединение множеств:
union(c("A","B","C","D"), c("E","F","C","D"))  # Смо-
  # трим объединенное множество
# Пересечение множеств
intersect(c("A","B","C","D"), c("E","F","C","D")) # Смо-
  # трим пересечение множеств
# Разность множеств x\y
setdiff(c("A","B","C","D"), c("E","F","C","D"))  # Смо-
  # трим разность множеств (из первого выбрасываем второе)
```

В языке R разработано большое количество функций работы со строками. Несмотря на то, что это выходит далеко за пределы нашего математического интереса, кратко перечислим некоторые из них, входящие в специализированную библиотеку «stringr»:

```
library(stringr)                    # Активируем пакет, предва
  # рительно загрузив его из репозитория
str_length("I am crazy about R!")          # Количество сим-
  # волов в строке, включая пробелы
word("I am crazy about R!", 1:3)           # Первые три слова
  # в строке
word("I am crazy about R!", -2)            # Второе слово
  # в строке  с конца
str_sub("I am crazy about R!", 3, 4)       # Третий и четвер-
  # тый символы строки
str_sub("I am crazy about R!", 1:19, 1:19)    # Извлекает
  # в массив все буквы строки s
str_sub("I am crazy about R!", 18, 18) <- "C++"; s # Заме-
  # няет 18-ый символ в строке на новые
tolower("I am crazy about R!")             # Изменяет заглав-
  # ные на строчные
toupper("I am crazy about R!")             # Изменяет строч-
  # ные на заглавные
grep("razy", c("I","am","crazy","about","R!"))  # Номер
  # слова, содержащего подстроку "razy"
pmatch("123", c("124", "12", "123"))    # То же самое
# Однократная замена "a" на "A":
str_replace(pattern = "a", replacement = "A", string = "I
am crazy about R!")
# Все замены "a" на "A":
str_replace_all(pattern = "a", replacement = "A", string =
"I am crazy about R!")
setequal(c("1","23","333"), c("333","1","23"))   # Совпа-
  # дают ли множества слов
```

Факторные переменные (factor)

Последний важнейший тип, который мы рассмотрим, – это тип факторной переменной или категориальной переменной (factor). Для простоты понимания приведем пример. Пусть при описании страховых случаев мы имеем дело с их условной классификацией «A», «A1», «B» и «Full». По сути, мы перечислили четыре текстовых значения переменной типа *character* и можем на этом остановиться. Но мы хотим ограничиться именно этими вариантами и воспринимать данные слова не просто как текст, а как уникальные исчерпывающие все случаи и непересекающиеся между собой категории.

Для этого нам потребуется наблюдаемые в «жизни» страховые случаи Cases, например, заданные перечнем:

```
Cases <- c("A", "B", "B", "A","Full","A","B","A1","A1")
  # Формируем Cases
Cases
```

просто переопределить в факторную переменную с именем Case_Factor:

```
Case_Factor <- factor(Cases)
# Объявляем категориальную переменную Case_Factor
```

Посмотрите отчет компилятора и сравните значения Cases и Case_Factor:

```
> Cases <- c(“A”, “B”, “B”, “A”,”Full”,”A”,”B”,”A1”,”A1”)  # Формируем Cases
> Cases
[1] “A” “B” “B” “A” “Full” “A” “B” “A1” “A1”
> Case_Factor <- factor(Cases)  # Объявляем категориальную переменную Case_Factor
> Case_Factor
```

```
[1] A    B    B    A    Full A    B    A1   A1
Levels: A A1 B Full
```

После этого нам становятся доступны многие полезные описательные функции в R. Например, можно посмотреть гистограмму распределения наших страховых случаев по категориям (см. также рис. 85):

```
plot(Case_Factor)
```

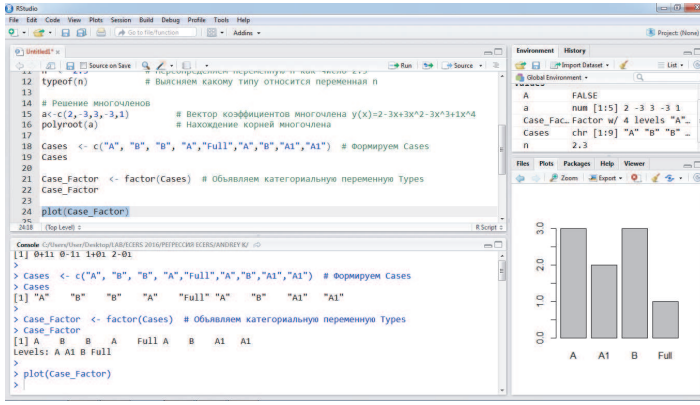


Рисунок 85

Если теперь ввести команду *typeof* определения типа переменной Case_Factor, то с удивлением обнаружится в типе не *factor*, а *integer*?!

```
typeof(Case_Factor)
```

```
> typeof(Case_Factor)
[1] «integer»
```

Хотя, разумеется, никакого *integer* здесь нет:

```
is.factor(Case_Factor)    # Проверяем, является ли Case_
  # Factor  факторной переменной
is.integer(Case_Factor)   # Проверяем, является ли Case_
  # Factor  целочисленной переменной
```

```
> is.factor(Case_Factor)
[1] TRUE
> is.integer(Case_Factor)
[1] FALSE
```

Но почему же функция typeof определяет нашу категориальную переменную как целочисленную. Ответ на этот вопрос даёт вызов функции *str* структуры объекта:

```
str(Case_Factor)    # Смотрим что это вообще за объект
  # Case_Factor
```

```
> str(Case_Factor)
 Factor w/ 4 levels “A”,”A1”,”B”,”Full”: 1 3 3 1
4 1 3 2 2
```

В отчете видно, что переменная Case_Factor является факторной с 4 уровнями, причем, как можно заметить, R трактует уровни как целые числа и внутренне представляет их себе как порядковые типа integer.

Этим активно пользуются, определяя между уровнями отношение строгого порядка («<» или «>»). К примеру, мы хотим упорядочить уровни от самых тяжелых к самым незначительным, т.е. задать такую последовательность категорий условно: «Full», «B», «A» и «A1». Это достигается введением новой категориальной, но уже упорядоченной, переменной с именем, например, Case_Factor_Order:

```
Case_Factor_Order <- ordered(Case_Factor,
levels=c("Full", "B", "A", "A1"))
Case_Factor_Order
```

```
>   Case_Factor_Order   <-   ordered(Case_Factor,
levels=c(“Full”, “B”, “A”, “A1”))
> Case_Factor_Order
[1] A    B    B    A    Full A    B    A1   A1
Levels: Full < B < A < A1
```

Из отчета видно, что в нашем случае: чем меньше, тем хуже. Теперь становится возможным сравнить, к примеру, четвертый страховой случай с пятым:

```
Case_Factor_Order[4]     # Смотрим 4-ый страховой случай
Case_Factor_Order[5]     # Смотрим 5-ый страховой случай
Case_Factor_Order[4] < Case_Factor_Order[5]  # Сравни-
  # ваем их: A < Full?
```

```
> Case_Factor_Order[4]                                  #
Смотрим 4-ый страховой случай
[1] A
Levels: Full < B < A < A1
> Case_Factor_Order[5]                                  #
Смотрим 5-ый страховой случай
[1] Full
Levels: Full < B < A < A1
> Case_Factor_Order[4] < Case_Factor_Order[5]   #
Сравниваем их : A < Full?
[1] FALSE
```

Действительно, то, что страховой случай типа A хуже страхового случая типа Full – это ложь.

Многомерные данные

Рассматриваемые в данной части практикума типы данных являются как бы надстройкой над уже введенными выше атомарными типами. Фактически, мы собираемся из уже имеющихся переменных атомарных типов строить более сложные многомерные конструкции, такие как векторы, массивы, матрицы, таблицы и списки.

Векторы (vector)

Задание 3. Образовать переменную S, которая содержала бы все десять цифр (0, 1, ... , 9), следующие друг за другом, в виде вектора.

Решение. Мы уже неоднократно использовали оператор c(...) для образования вектора из указанных аргументов:

```
S <- c(0,1,2,3,4,5,6,7,8,9)      # Задаем вектор S
S                                # Смотрим, что получилось
```

```
> S <- c(0,1,2,3,4,5,6,7,8,9)     # Задаем вектор S
> S                               # Смотрим, что получилось
 [1] 0 1 2 3 4 5 6 7 8 9
```

Вот, собственно, и всё задание.

Вообще, действие оператора c(...), т.е. понятие вектора в языке R, весьма ёмкое. Здесь вектор – это упорядоченная совокупность конечного числа объектов одного и того же типа. Интерпретатор R сам определяет какой это тип, выставляя максимально широкий из возможных, подходящих ко всем аргументам.

К примеру, если задать в векторе *X* объекты разного типа:

```
X <- c(2,"Hello!", -3.56, FALSE)      # Задаем вектор X
X                                     # Смотрим, что получилось
```

то R объявит вектор *X* как текстовый, т.к. имеется слово «Hello!», и ни один из числовых типов как объединяющий не подходит:

```
> X <- c(2,»Hello!», -3.56, FALSE)        # Задаем вектор X
> X                                       # Смотрим, что получилось
[1] «2»       «Hello!» «-3.56»  «FALSE»
```

Если же удалить объект «Hello!», то оставшийся вектор *X* примет тип *double* с заменой FALSE на ноль.

Но вернемся к нашему заданию №3. Напомним, что обращение к элементам вектора *S* задается оператором квадратных скобок [...]:

```
S <- c(0,1,2,3,4,5,6,7,8,9)   # Задаем вектор S
S                             # Смотрим, что получилось
S[2]      # Выводим на экран вторую координату вектора S
Q <- S[-3]; Q          # Присваиваем переменной Q вектор S
  # без третьей координаты
V <- S[3:5]; V         # Присваиваем переменной V  3, 4 и 5-ую
  # координаты вектора S
W <- S[S>7]; W      # Присваиваем переменной W коорди-
  # наты вектора S, превышающие 7
```

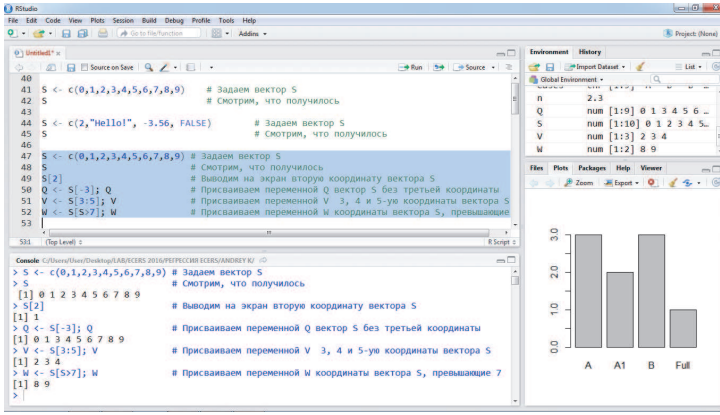


Рисунок 86

И, несмотря на то, что в R–консоли вектор представлен в виде строки значений (делается это для экономии места), на самом деле вектор – это столбец. Представляйте себе любой вектор именно как столбец значений, а не строку.

Кстати, мы объявили вектор *S* с десятью координатами. А что произойдет, если мы присвоим какой-нибудь несуществующей координате S[14] значение 88?

```
S[14] <- 88
S
```

```
> S[14] <- 88
> S
 [1]  0  1  2  3  4  5  6  7  8  9 NA NA NA 88
```

Как видим из отчета консоли, R доопределил последующие координаты вектора *S* до 14-ой и заполнил пустующие специальным зарезервированным кодом: NA (Not avialable).

Добавим яркий штрих. Пусть нам требуется ввести в качестве вектора – метрику человека: его рост, вес и возраст. Для удобства в R координаты вектора можно назвать и обращаться к ним по именам.

Делается это так:

```
D <- c(180, 70, 25)                    # Образовали вектор
  # D(180 - рост,70 - вес, 25- возраст)
names(D) <- c("height", "weight", "age")        # Назвали
  # координаты  (рост, вес, возраст)
D                                  # Смотрим, что получилось
D["age"]                           # Смотрим какой возраст
D[3]                               # то же самое
```

```
> D <- c(180,70,25)              # Образовали вектор
D(180,70,25)
> names(D) <- c(«height»,»weight»,»age») # Назвали
координаты  (рост, вес, возраст)
> D                           # Смотрим, что получилось
height weight    age
   180     70     25
> D[«age»]                     # Смотрим какой возраст
age
 25
> D[3]                           # то же самое
age
 25
```

Но можно было обойтись вообще одной строчкой:

```
D <- c(height=180, weight=70, age=25)
```

Замечание. Попробуйте теперь самостоятельно набрать следующий ниже код и убедиться, что в консольном отчете всё понятно:

```
typeof(D)        # Посмотреть тип объекта D
is.vector(D)     # Посмотреть, является ли D вектором
str(D)           # Посмотреть, что это вообще за объект D
```

Массивы (array)

Задание 4. Образовать переменную *M*, которая содержала бы как массив данные таблицы:

A	A	A	A
A	A	A	A
A	A	A	2

Решение. Как видим, наша таблица – не числовая, вернее большая ее часть – не числовая. Но массивы в R, подобно векторам, являются упорядоченными совокупностями объектов одного и того же типа, только размерность упорядочивания больше чем у вектора. Так что, двойка на последнем месте *M*[3,4] будет в представлении массива тоже текстовая, даже если мы попытаемся ее записать как число.

Для работы с массивами используются команда *array(…)*. Сначала объявляем массив *M* требуемой размерности 3x4 (3 – количество строк, 4 – количество столбцов) и первоначально заполняем его символами «A»:

```
M <- array(data = "A", dim = c(3,4)); M
# Заполняем массив размера 3x4 символами "A"
```

Затем, исправляем элемент M[3,4] на требуемую в задании двойку:

```
M[3,4] <- 2;  M        # Исправляем требуемый элемент на 2
```

Совершенно аналогично работе с векторами, строки и столбцы массива для удобства можно называть и обращаться к ним по имени. Только использовать необходимо для этого команду не *names*(...), а *dimnames*(...) (см. также рис. 87):

```
# Не называем строки - NULL, но называем столбцы -
c("One","Two","Three","Four"):
dimnames(M) <- list(NULL,
c("One","Two","Three","Four")); M
M[2,"Three"]          # Обращаемся к элементу по имени...
```

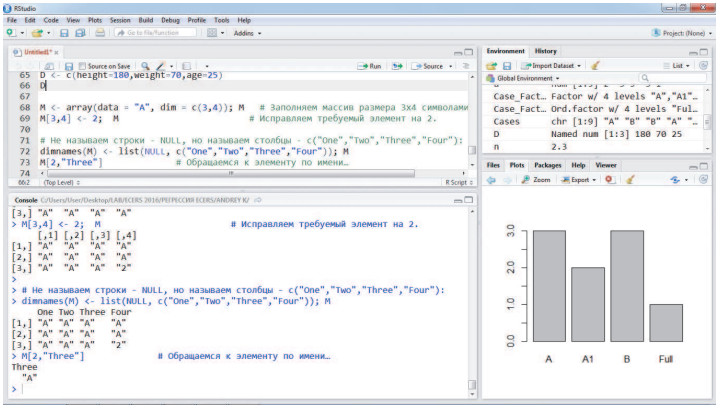


Рисунок 87

Важной особенностью в R при оперировании с массивами является работа с их размерностью. Во введенном массиве *M* мы можем изменить размерность, к примеру, на 6x2. Посмотрите, как при этом изменится представление элементов массива:

```
dim(M) <- c(6, 2); M   # Изменяем размерность массива M
  # на  6x2
```

Массивы могут быть и большей размерности, но их отображение уже менее наглядно. Если массив является двумерным и числовым, его называют матри-

цей (тип *matrix*) и работают уже в специальном процедурном режиме. Числовые векторы и матрицы мы подробно рассмотрим в следующих занятиях в контексте аппарата линейной алгебры.

Таблицы (data.frame)

Еще одним мощным расширением массивов являются объекты типа *data.frame*, т.е. по сути таблицы (часто, не утруждая себя переводом, их называют попросту «фреймами»). Объект этого типа надо представлять себе как упорядоченный набор векторов (столбцов) одной и той же длины, но быть может разных между собой типов. Классическим примером здесь служит метрическая таблица какой-либо группы людей. Первый столбец в этой таблице является факторной переменной, отвечающей за пол человека (male, female); второй и третий столбцы – вещественные переменные, отвечающие за рост и вес; четвертый столбец является текстовой переменной, хранящей Ф.И.О. и т.д.

Задание 5. Определить объект *Exam* типа *data.frame* и описать им результаты зачета (для первых трех человек):

№ п/п	Фамилия, имя, отчество	№ зачетной книжки	Баллы				Оценка (прописью)
			атт.	работа в семестре	зачет	итого	
1	Билан Алёна Юрьевна	151416	20	20	60	100	зачтено
2	Буянтогтох Монхтоголдор	153305	11	11	50	72	зачтено
3	Гаджиев Туран Элшанович	152613	3	2	0	5	не зачтено
4	Дивяшова Анастасия Владимировна	151420	20	15	45	80	зачтено
5	Ермакова Ольга Александровна	150358	4	2	45	51	зачтено

Решение. Сначала подготовим необходимые столбцы:

```
ФИО <- c("Билан","Буянтогтох","Гаджиев")
КНИЖКА <- c(151416,153305,152613)
А1 <- c(20,11,3)
```

```
A2 <- c(20,11,2)
ЗАЧЕТ <- c(60,50,0)
ИТОГ <- c(100,72,5)
ОЦЕНКА <- c("зачтено","зачтено","не зачтено")
```

Из которых вторым шагом образуем саму таблицу Exam:

```
Exam <- data.frame(Name = ФИО, N = КНИЖКА, ATT1
= A1, ATT2 = A2,
          Z = ЗАЧЕТ, All = ИТОГ , Rez = ОЦЕНКА)  # Объ-
  # являем таблицу
Exam                                  # Смотрим результат
```

Обратите внимание, что в каждом аргументе data.frame, например, в первом «Name = ФИО»: Name – это собственное имя первого столбца в таблице Exam, а ФИО – конкретный вектор введенных ранее значений для этого столбца (см. рис. 88).

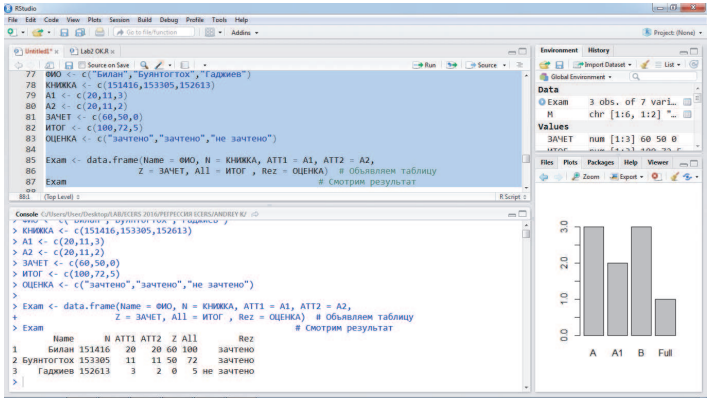


Рисунок 88

Напомним, что к данным любой таблицы можно обращаться и как к элементам массива, и как к элементам списка. Проверьте это самостоятельно пошагово:

```
Exam$Name        # Весь первый столбец таблицы Exam
  # (это наш первый вектор таблицы)
Exam[1, ]        # Вся первая строка таблицы Exam (это
  # одномерный массив)
Exam[1, 1]       # Обращаемся к элементу первой стро-
  # ки и первого столбца таблицы Exam
Exam$Name[1]     # То же самое (обращаемся к первому
  # элементу первого столбца $Name)

Exam[2, 6]       # Обращаемся к элементу [2,6] таблицы
  # Exam
Exam$All[2]      # То же самое
```

В заключение знакомства с этим замечательным типом *data.frame*, который, кстати, часто используется в эконометрике и анализе данных, укажем сверх популярную библиотеку *dplyr* и её несколько полезных функций для работы с таблицами данных. Тем более, что в подавляющем большинстве случаев подобные таблицы не вводят руками, а получают из внешних источников.

Отработайте пошагово каждую строчку приводимого ниже кода и внимательно посмотрите в консоли производимые изменения в таблице Exam.

```
library(dplyr)        # Активация библиотеки
```

filter

```
# Команда filter запишет в H новую таблицу, в которой
  # из Exam оставит только тех (те строки), кто сдал
  # зачет:
H <- dplyr::filter(Exam, Rez=="зачтено"); H   # Смотрим!

# Команда filter запишет в H новую таблицу, в которой
  # из Exam оставит только тех
# (те строки), кто набрал общий балл в пределах от 70 до
  # 85 включительно:
```

```
H <- dplyr::filter(Exam, (All <= 85) & (All >= 70)); H
# Смотрим!
```

select

```
# Команда select запишет в H новую таблицу, содержа-
  # щую указанные столбцы из Exam:
H <- dplyr::select(Exam, Name, All, Rez); H      # Смотрим!
```

rename

```
# Команда rename запишет в H новую таблицу, содержа-
  # щую Exam с рядом переименованных столбцов:
H <- dplyr::rename(Exam, Surname = Name, Result = Rez); H
# Смотрим!
```

mutate

```
# Команда mutate действует аналогично, позволяя изме-
  # нять или добавлять новые столбцы:
H <- dplyr::mutate(Exam, Attestation = ATT1 + ATT2); H
# Смотрим!
```

glimpse

```
# Одна из самых удобных функций, раскрывающих
  # структуру любого объекта...
dplyr::glimpse(H)        # Смотрим!
```

Замечание. Безусловную ссылку на библиотеку «dplyr::» в приведенных примерах можно опускать из кода, если нет активированных библиотек с одноименными командами.

Списки (List)

Последним и, в каком-то смысле, самым обобщающим типом рассмотрим тип *list*, понимаемый как список. Объекты данного типа включают в себя упорядоченные переменные различных типов и различной размерности. Представим себе, что мы имеем какой-либо вектор A, строковую переменную S (тип character) и,

скажем, таблицу *B* (тип data.frame). Мы можем объединить все эти различные переменные в один объект типа *list* следующей командой:

L <- list(A1 = A, A2 = S, A3 = B).

Аналогично созданию таблиц: A1, A2 и A3 – названия объектов внутри нашего списка L. Причем, к этим внутренним объектам можно обратиться двумя вполне естественными способами:

L[[1]], L[[2]] и L[[3]]

или

L$A1, L$A2 и L$A3.

Задание 6. Определить и ввести необходимые переменные, наиболее полно описывающие экзаменационную ведомость группы студентов (для первых трех студентов):

ФГОБУ ВО «ФИНАНСОВЫЙ УНИВЕРСИТЕТ ПРИ ПРАВИТЕЛЬСТВЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»

Факультет налогов и налогообложения

ЗАЧЕТНАЯ ВЕДОМОСТЬ № 04.06-303611

Учебный год:	2016/2017	Форма обучения:	очная	Контроль:	зачет
Семестр:	3	Группа:	НЗ-1	Дата:	

Уч. дисциплина: Теория вероятностей и математическая статистика

Экзаменатор: Зададаев Сергей Алексеевич

Департ./Кафедра:

№ п/п	Фамилия, имя, отчество	№ зачетной книжки	Баллы				Оценка (прописью)
			атт.	работа в семестре	зачет	итого	
1	Бипан Алёна Юрьевна	151416	20	20	60	100	зачтено
2	Буянтогтох Монхтоолдор	153305	11	11	50	72	зачтено
3	Гаджиев Туран Эльшанович	152613	3	2	0	5	не зачтено
4	Дивяшова Анастасия Владимировна	151420	20	15	45	80	зачтено
5	Ермакова Ольга Александровна	150358	4	2	45	51	зачтено

Решение. Такой документ проще всего представить в виде объекта типа *list*, назовем его Vedomost. Он будет состоять из номера ведомости (N.V), учебного года (Year), номера семестра (N.S), названия группы (Name.G), названия дисциплины (Subject), Фамилии

экзаменатора (Professor) и самой таблицы результатов (Exam). Объявим эти переменные и образуем соответствующий список. (Воспользуемся тем, что таблица Exam нами уже сформирована в задании №5).

```
N.V <- "04.06-303611"          # Текстовая переменная
  # номера ведомости
Year <- "2016/2017"            # Текстовая переменная
  # номера учебного года
N.S <- 3                       # Номер семестра
Name.G <- "Н3-1"               # Название группы
Subject <-    "Теория вероятностей и математическая
статистика"  # Название дисциплины
Professor <-"Зададаев Сергей Алексеевич"
# ФИО экзаменатора
# Exam... Таблица Exam была  объявлена ранее… в зада-
  # нии №5

# Объявляем список Vedomost
Vedomost <- list(N.Ved = N.V, Year = Year, N.Sem = N.S,
Groop = Name.G,
    Subject = Subject, Professor = Professor, Exam = Exam)
Vedomost                       # Смотрим результат
```

Если мы теперь хотим обратиться к различным элементам созданного списка, то нам необходимо помнить: какие поля (записи, объекты) имеются внутри нашего списка Vedomost и внутри вложенной в список таблицы Exam. Используем для этого команду glimpse:

```
glimpse(Vedomost)    # Смотрим структуру объекта Vedomost
```

```
> glimpse(Vedomost)
List of 7
 $ N.Ved    : chr “04.06-303611”
 $ Year     : chr “2016/2017”
 $ N.Sem    : num 3
 $ Groop    : chr «Н3-1»
```

```
 $ Subject  : chr «Теория вероятностей и математи-
ческая статистика»
 $ Professor: chr «Зададаев Сергей Алексеевич»
 $ Exam     :'data.frame':       3  obs.  of    7
variables:
  ..$ Name: Factor w/ 3 levels "Билан","Буянтог-
тох",..: 1 2 3
  ..$ N   : num [1:3] 151416 153305 152613
  ..$ ATT1: num [1:3] 20 11 3
  ..$ ATT2: num [1:3] 20 11 2
  ..$ Z   : num [1:3] 60 50 0
  ..$ All : num [1:3] 100 72 5
  ..$ Rez : Factor w/ 2 levels "зачтено","не зачте-
но": 1 1 2
```

Теперь можно с легкостью обращаться к любым элементам вложения (см. также рис. 89):

```
Vedomost[[1]]          # Обращение к номеру ведомости
Vedomost$N.Ved         # То же самое

Vedomost$Exam          # Обращение к таблице результатов
  # зачета
Vedomost[[7]]          # То же самое

Vedomost$Exam$All[1]     # Обращение к результату
  # в баллах первого студента в ведомости
Vedomost$Exam[1,6]       # То же самое
Vedomost[[7]][1, 6]      # То же самое
```

Задания для самостоятельной работы

1. Решить уравнения в области комплексных чисел

a) $z^2+6z+25=0$

b) $z^2-8z+25=0$

c) $z^4=81$

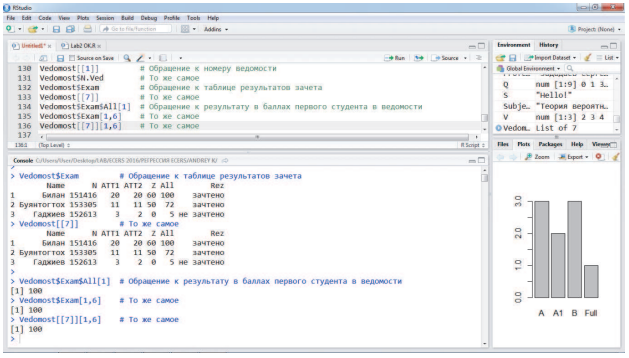


Рисунок 89

d) $z^3 - z^2 - (3+4i)z + 3 + 4i = 0$

e) $z^3 - (1+4i)z^2 + (-3+4i)z + 3 = 0$

2. Найти вещественную и мнимую части выражений

a) $\dfrac{(2+i)^4}{3-4i}$

b) $\dfrac{(3-5i)(2+5i)}{3+4i}$

c) $\dfrac{\left(4e^{-\pi i/4}\right)^3}{5e^{\pi i/3}}$

3. Описать подходящим образом в R данные, представленные в таблице:

Главные параметры проекта федерального бюджета на 2014– 2017 гг.

	2014	2015	2016	2017
	оценка	проект бюджета		
Доходы, трлн руб.	14,07	15,08	15,8	16,5
в % к ВВП	19,9	19,5	19,0	18,4
Нефтегазовые, трлн руб.	7,5	7,7	7,8	8,34
в % ко всем доходам	51,9	51,2	50,8	49,6
Расходы, трлн руб.	13,96	15,51	16,3	17,0
в % к ВВП	19,5	20,0	19,6	19,0
Дефицит (–), Профицит (+), трлн руб.	+0,11	–0,43	–0,5	–0,5
в % к ВВП	+0,2	–0,6	–0,6	–0,6

P.S. достаточно описать доходную часть.

4. Описать подходящим образом в R данные, составляющие квитанцию на оплату штрафов в ГИБДД:

<table>
<tr><td>Извещение</td><td colspan="3"><u>Получатель платежа</u>: УФК по МО (УГИБДД ГУ МВД России по Московской области)
ИНН: 7702300872
КПП: 770201001
<u>Банк получателя</u>: Отделение №1 МГТУ Банка России г.Москва 705
Расчетный счет: 40101810600000010102
БИК: 044583001
КБК: 18811630020016000140
ОКАТО: 46218000000</td></tr>
<tr><td></td><td colspan="3">Иванов Иван Иванович</td></tr>
<tr><td></td><td colspan="3">Москва, ул. Красная площадь, д.1</td></tr>
<tr><td></td><td colspan="3">(фамилия, имя, отчество, адрес плательщика)</td></tr>
<tr><td></td><td>Вид платежа</td><td>Дата</td><td>Сумма</td></tr>
<tr><td></td><td>Административный штраф, 01258789877888</td><td></td><td>1550</td></tr>
<tr><td>Кассир</td><td colspan="3">Плательщик: (подпись)</td></tr>
<tr><td colspan="4"></td></tr>
<tr><td>Квитанция</td><td colspan="3"><u>Получатель платежа</u>: УФК по МО (УГИБДД ГУ МВД России по Московской области)
ИНН: 7702300872
КПП: 770201001
<u>Банк получателя</u>: Отделение №1 МГТУ Банка России г.Москва 705
Расчетный счет: 40101810600000010102
БИК: 044583001
КБК: 18811630020016000140
ОКАТО: 46218000000</td></tr>
<tr><td></td><td colspan="3">Иванов Иван Иванович</td></tr>
<tr><td></td><td colspan="3">Москва, ул. Красная площадь, д.1</td></tr>
<tr><td></td><td colspan="3">(фамилия, имя, отчество, адрес плательщика)</td></tr>
<tr><td></td><td>Вид платежа</td><td>Дата</td><td>Сумма</td></tr>
<tr><td></td><td>Административный штраф 01258789877888</td><td></td><td>1550</td></tr>
<tr><td>Кассир</td><td colspan="3">Плательщик: (подпись)</td></tr>
</table>

5. * Придумать “житейский» список, содержащий не менее двух различных таблиц. Как можно обратиться к элементам таблиц, обращаясь к списку?

Практикум 8. Циклические процедуры в R (RStudio)

В этой части мы продолжаем систематизировать важнейшие компоненты языка R. Цель нашего ближайшего интереса – знакомство со стандартными и специфическими возможностями языка R в работе с циклическими алгоритмами и, в частности, с векторизованными вычислениями.

Как всегда, запустим RStudio и создадим новый файл программы (Ctrl+Shift+N).

Цикл for

Оператор *for*, пожалуй, один из немногих, который прочно укоренился во всех языках программирования. Семантика его использования в R следующая:

for (*i* in *a*) { ... },

где *i* – переменная цикла, последовательно принимающая все значения координат вектора *a*, а в фигурных скобках стоят выполняемые в цикле операторы. Обсудим применение данного цикла подробнее на примерах.

Задание 1. Объявить какой-нибудь вектор *a*, содержащий пропущенные значения (NA), и составить код процедуры, заменяющей все его NA на значение 0.

Решение. Объявим произвольный вектор *a* с отсутствующими значениями:

```
a <- c(1,3,NA,NA,5,2,NA,NA,7); a
# Задаем вектор с пропущенными значениями
```

и построим цикл, перебирающий координаты вектора *a*, внутри которого будем переопределять значение координат с NA на ноль:

```
for (i in 1:9) {                    # Пробегаем по всем девяти
   # номерам координат вектора a
  if (is.na(a[i])) {a[i] <- 0}    # Если значение в координате
   # a[i] пропущено заменяем его нулем
}
a
```

```
> a <- c(1,3,NA,NA,5,2,NA,NA,7); a
[1]  1  3 NA NA  5  2 NA NA  7
> for (i in 1:9) {
+     if (is.na(a[i])) {a[i] <- 0}
+ }
> a
[1] 1 3 0 0 5 2 0 0 7
```

Обратите внимание, что проверкой на отсутствие значения в *i*-ой координате является *is.na(a[i])*, которая возвращает TRUE для NA и FALSE для всех остальных значений. Можно было бы в условии оператора *if* также написать *is.na(a[i])==TRUE*. Но было бы синтаксической ошибкой попытаться составить условие *a[i]==NA* или *a[i]== «NA»* – всё это категорически не сработает, т.к. NA – это код отсутствия данных, а не зарезервированная переменная или константа.

Кстати, язык R славится тем, что многие циклические алгоритмы в нем реализованы «аппаратно», т.е. специальным образом запрограммированы под параллельные вычисления. Например, в рассмотренном задании вместо оператора *for* можно было использовать

сверх быстрый оператор *[...]* и составить вместо цикла всего одну строку кода:

```
a <- c(1,3,NA,NA,5,2,NA,NA,7); a       # Задаем вектор
  # с пропущенными значениями
a[is.na(a)] <- 0                       # Записываем нули
  # в те места вектора a, где есть NA
a                                      # Смотрим результат
```

Задание 2. Объявить вектор $\vec{b} = (1,3,1,1,5,2,3,3,7,7,3,1,3)$ и составить код процедуры, подсчитывающей количество троек среди его координат.

Решение. Аналогично объявим сначала соответствующий вектор *b*:

```
b <- c(1,3,1,1,5,2,3,3,7,7,3,1,3); b          # Задан вектор b
```

Далее введем переменную *s* (первоначально равную нулю), в которую будем прибавлять по единице всякий раз, как обнаружим тройку в координате:

```
s <- 0                        # Первоначальное число наблюдений
  # троек в координатах вектора b
```

И в заключение составим цикл, в котором переменная *i* будет пробегать все номера координат вектора *b*; при этом соответствующая координата вектора *b*[i] будет проверяться на равенство 3. В положительном исходе будем прибавлять 1 к нашему счетчику троек *s* (см. также рис. 90):

```
for (i in 1:length(b)) {          # i пробегает все номера коор-
  # динат вектора b
 if (b[i]==3) {s <- s + 1}        # Если значение равно 3, то
  # к сумме наблюдений s прибавляем 1
}
s                                 # Итоговое количество троек
```

Нетрудно заметить, что мы использовали специальный оператор *length(b)*, возвращающий количество координат в векторе *b*.

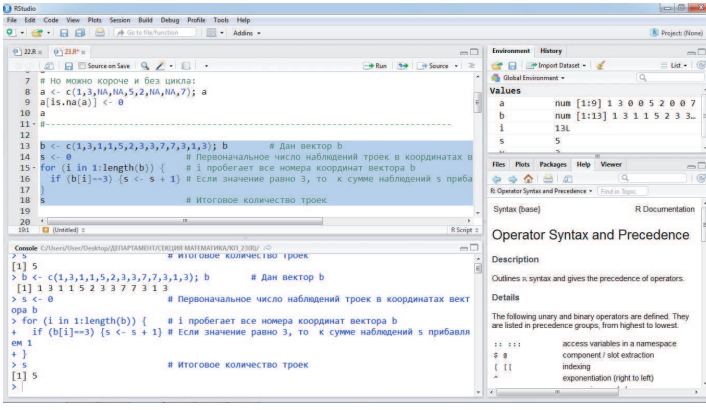


Рисунок 90

В качестве полезной альтернативы можно привести эквивалентный код, в котором переменная цикла *x* пробегает не по номерам координат вектора *b*, а по самим значениям его координат:

```
b <- c(1,3,1,1,5,2,3,3,7,7,3,1,3); b          # Дан вектор b
s <- 0                    # Первоначальное число наблюдений
  # троек в координатах вектора b
for (x in b) {            # x пробегает все значения координат
  # вектора b
 if (x==3) {s <- s + 1}       # Если значение x равно 3, то
  # к сумме наблюдений s прибавляем 1
}
s                                  # Итоговое количество троек
```

Разумеется, и это алгоритмическое действие может быть минимизировано по коду и быстродействию в R без использования оператора цикла:

```
b <- c(1,3,1,1,5,2,3,3,7,7,3,1,3); b     # Задан вектор b
length(b[b==3]); b                      # Количество элемен-
  # тов в векторе b, которые равны 3
```

Задание 3. Составить код функции *f(a)*, отвечающей на вопрос: есть ли среди координат вектора *a*, значения, превышающие 4? Проверить работоспособность функции на нескольких примерах.

Решение. В первую очередь создадим необходимую функцию *f(x)*:

```
f <- function(a) {
    Res <- "Элементов, больше 4, нет."
    for (x in a) {
      if (x > 4) {Res <- "Есть элементы, превышающие 4.";
break}
    }
   return(Res)
}
```

Здесь мы использовали следующую логику: до входа в цикл результат функции отрицательный (Res <- «Элементов, больше 4, нет.»), а в цикле Res может измениться на положительный, если найдется координата $x > 4$. Причем, достаточно, чтобы нашлась хотя бы одна такая координата. Именно поэтому, мы использовали специальный оператор *break* для выхода из цикла (прерывание цикла) при первой такой возможности. Кстати, в ряде случаев может оказаться уместным не категоричный выход из цикла, а переход к следующему значению циклической переменной – тогда надо использовать вместо *break* оператор *next*.

Далее остается проверить работоспособность функции на нескольких примерах (см. также рис. 91):

```
b1 <- c(1,3,1,1,5); b1                  # Зададим вектор b1
f(b1)                                   # Вызов функции
b2 <- c(1,3,1,1,–5,0,–2,2,3); b2        # Зададим вектор b2
f(b2)                                   # Вызов функции
```

Замечание. У нашей функции есть один небольшой недостаток: она не будет работать с векторами, в которых есть

пропущенные значения. В самом деле: что это за векторы такие, без координат?!

Тем не менее, подумайте, как можно было бы обойти этот отказ, т.е. как подправить нашу функцию, чтобы она игнорировала значения NA.

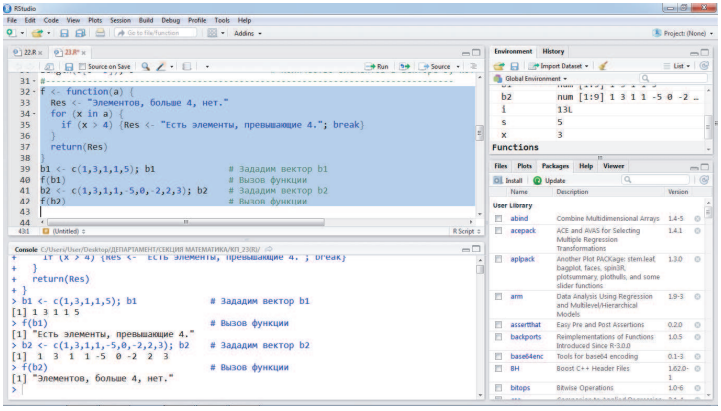


Рисунок 91

Продолжая хорошую традицию, приведем код процедуры, которая без цикла справляется с нашей задачей: быстрее, да и выглядит лаконичнее:

```
b1 <- c(1,3,1,1,5); b1                    # Зададим вектор b1
length(b1[b1>4])>0
# Если TRUE, то есть; если FALSE – то нет.
```

Цикл while

Еще одной разновидностью циклических процедур является цикл с оператором *while*. Используется он в отличие от *for* в тех случаях, когда количество циклических итераций заранее не известно: часто, когда остановка цикла происходит по некоторому событию. Семантика здесь такова:

while(...) {...},

где в первых скобках (...) стоит условие, при выполнении которого будет выполняться группа операторов из вторых скобок {...}.

Задание 4. Вычислить $\underbrace{\sqrt{\sqrt{\sqrt{\ldots\sqrt{100}}}}}_{n}$ пока полученный результат в первый раз не окажется меньше 2.

Решение. Данное задание можно было бы охарактеризовать как требующее рекуррентных вычислений.

Введем переменную x=100 и устроим рекуррентную последовательность вычисления корней из предыдущих результатов с проверкой результата на превышение 2 (см. также рис. 92):

```
x <- 100                # Задаем первоначальный x
while(x >= 2) {         # Проверяем текущее условие
 x <- sqrt(x)           # Вычисляем корень и перезаписыва-
  # ем результат снова в x
}
x                       # Ответ
```

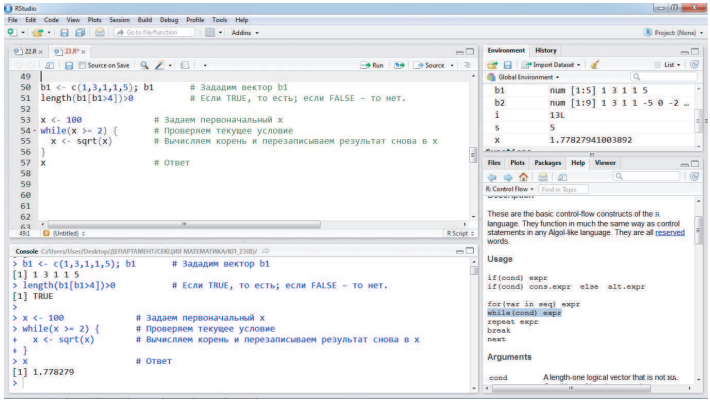


Рисунок 92

Обратите внимание, что цикл *while* выполняется пока выполняется его условие (x >= 2). Здесь также употребим оператор прерывания цикла *break*.

Цикл repeat

Цикл repeat, пожалуй, самый прямолинейный. Выглядит эта конструкция так:

repeat {...}

где в скобках указываются команды, которые данный оператор будет циклически повторять. Если вы спросите: как долго он собирается их повторять, то в ответ получите – сколь угодно долго, пока вы оплачиваете электричество, например. Из этого ясно, что выход из оператора *repeat* возможен только с помощью *break* по выполнению требуемого условия.

Приведем характерный пример:

Задание 5. Написать код «ожидания». Например, ждать пока пользователь не введет с клавиатуры символ q.

Решение. Данное задание проще всего выполнить именно с помощью оператора repeat. Введите следующие ниже строки только пока не запускайте на компиляцию:

```
repeat {
 if (readline("Введите q для выхода и нажмите
Enter...")=="q") {break}
}
```

Здесь требуется дать некоторые пояснения, чтобы понять – как это работает. Дело в том, что мы используем удобную интерфейсную оболочку RStudio. Редактируем текст в левом верхнем углу, а отрабатывает интерпретатор R наш код слева внизу (консоль).

После запуска нашего фрагмента *repeat* {...} команда *readline* выведет в консоль поясняющий текст и будет ждать действия пользователя мигающим курсором там же в строке консоли. Вам необходимо активировать это нижнее левое окно щелчком мыши и ввести символ q после чего подтвердить ввод нажатием Enter. Если

ввести другой символ и нажать Enter, то оператор repeat не закончит свою работу и снова высветит приглашение ввести символ q. Теперь попробуйте запустить наш код ... ориентируйтесь на скриншот (рис. 93):

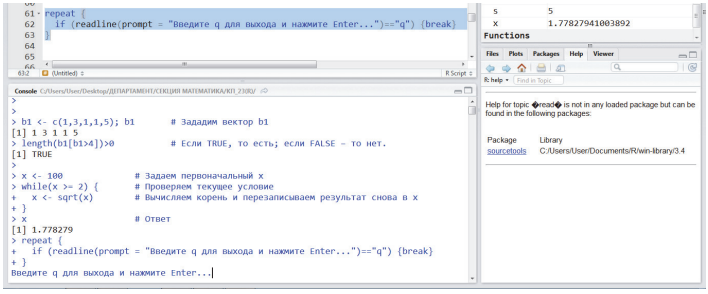


Рисунок 93

Важно! Нажатие Esc при активном окне консоли приводит к безусловному прерыванию работы всей вашей программы и переходу R в ожидание следующих команд. Этим надо решительно пользоваться при зацикливании или ошибках кода, когда R ставит в консоли символ +, ожидая продолжения команды, а пользователь продолжения не предполагает. Эта ошибка часто возникает при неверно расставленных скобках и парализует всю работу с R. Так что, если чувствуете, что R «завис» – активируйте окно консоли мышкой и нажимайте Esc!

Векторизованная процедура sapply

В следующих ниже примерах мы рассмотрим мощную процедуру семейства «apply», которая позволяет избежать употребления циклов и распараллелить векторизованные вычисления.

Процедура *sapply* позволяет применить какую-либо заранее объявленную функцию $f(t)$ к каждому элементу заданного вектора a (массива, матрицы) по правилу:

sapply(a, function(t) f(t))

или

sapply(a, function(t) {.. ~ *t* ..}),

где в фигурных скобках прописана конкретная функция через переменную *t*.

Задание 6. Программно изменить вектор $\vec{a}(1,2,3,\ldots,10)$ путем прибавления к каждой его координате единицы.

Решение. Приведем несколько решений этой задачи, на которых прокомментируем особенности работы с векторизованными объектами.

1 Способ. Самый простой (и лучший для данной конкретной задачи):

```
a <- 1:10              # Объявляем вектор
b <- a + 1             # Прибавляем единицу к каждой коор-
  # динате вектора a
b                      # Смотрим что получилось
```

2 Способ. Самый неудачный (но обожаемый студентами)

```
a <- 1:10              # Объявляем вектор a
b <- a                 # Объявляем будущий измененный
  # вектор b первоначально как a
for (i in 1:10) {
   b[i] <- a[i] + 1    # Прибавляем единицу к каждой
  # координате вектора a
}
b                      # Смотрим что получилось
```

3 Способ. Самый многообещающий (и лучший для данного типа задач)

```
a <- 1:10                              # Объявляем вектор a
b <- sapply(a, function(t) {t+1})      # Применяем функцию
  # t +1 к каждой координате a
b                                      # Смотрим что получилось
```

4 Способ. Предыдущий способ, но с вызовом ранее объявленной функции:

```
Plus1 <- function(t) {t + 1}        # Объявляем функцию Plus1
a <- 1:10                           # Объявляем вектор a
b <- sapply(a, function(t) Plus1(t))    # Применяем функ-
  # цию Plus1 к каждой координате a
b                                   # Смотрим что получилось
```

Результат везде одинаков:

```
> b     # Смотрим что получилось
 [1]  2  3  4  5  6  7  8  9 10 11
```

Но почему же последние два способа являются самыми предпочтительными для данного класса задач? Ответ на этот вопрос мы поймем из следующего примера.

Задание 7. Пусть дан произвольный вектор a, координаты которого – натуральные числа (возможно с повторениями и не по порядку). Составить программу, которая приписывала бы к каждой его координате слово «Even» для четных значений и «Odd» для нечетных.

Решение. Для начала составим функцию, которая приписывает натуральному числу его четность:

```
OddEven <- function(t) {                # Объявляем функцию
  # OddEven
 if (t%%2 == 0) Res <- paste(t,"-","Even")          # Для
  # четного значения
       else Res <- paste(t,"-","Odd")               # Для
  # нечетного значения
 return(Res)                    # Возвращаем реультат
}
OddEven(3)                      # Смотрим результат для 3
OddEven(8)                      # Смотрим результат для 8
```

```
> OddEven(3)
[1] «3 - Odd»
```

```
> OddEven(8)
[1] «8 - Even»
```

Здесь мы использовали операцию нахождения остатка от деления %%, т.е число *t%%2* может быть равно 0 для случая четного *t* и равно 1 – для нечетного *t*.

Кстати, обратим внимание, что для векторизованных обращений наша функция OddEven не годится:

```
a <- c(1,4,5,2,2,6,7)          # Объявляем вектор a
OddEven(a)                     # Убеждаемся, что для векто-
  # ров функция не работает
```

```
> OddEven(a)                          # Убеждаемся,
что для векторов не работает
[1] "1 - Odd" "4 - Odd" "5 - Odd" "2 - Odd" "2 -
Odd" "6 - Odd" "7 - Odd"
Warning message:
In if (t%%2 == 0) Res <- paste(t, "-", "Even") else
Res <- paste(t,  :
  the condition has length > 1 and only the first
element will be used
```

Замечание. Мы уже как-то отмечали, что оператор if (...) {} else {} не поддерживает векторизованных обращений.

Но это нам не помешает модифицировать координаты вектора *a* без цикла с помощью процедуры *sapply* – вот в чем ее неоспоримое преимущество (см. также рис. 94):

```
b <- sapply(a, function(t) OddEven(t))
# Применяем функцию к каждой координате
b                                   # Смотрим что получилось
```

Вообще, ранее мы указывали, что если хотим использовать векторизованную технику обращения к функции, то вместо оператора if (...) {...} else {...} не-

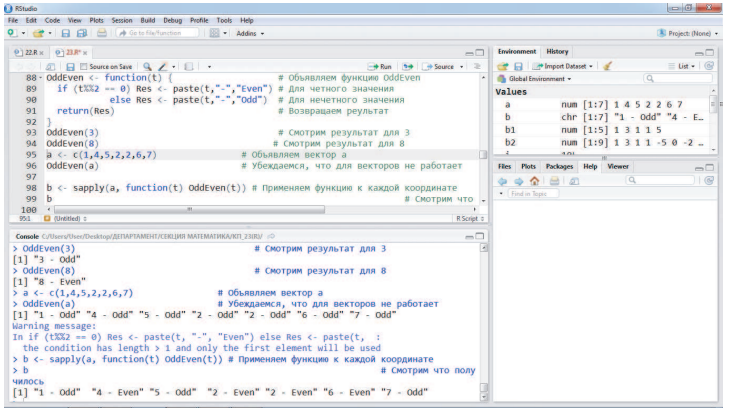


Рисунок 94

обходимо использовать конструкцию ifelse(...), а еще лучше оператор [...].

Действительно, грамотно составленная функция OddEven не нуждалась бы в дальнейшем применении sapply:

```
OddEven <- function(t) {                    # Объявляем
  # функцию OddEven
   Res <- ifelse(t%%2==0,"Even","Odd")      # Использу-
  # ем ifelse
   return(paste(t,"-",Res))
}
a <- c(1,4,5,2,2,6,7)                       # Объявляем вектор а
OddEven(a)                                  # Все прекрасно работает
  # для векторов
```

```
> OddEven <- function(t)  {
# Объявляем функцию OddEven
+   Res <- ifelse(t%%2==0,"Even","Odd")
# Используем ifelse
+   return(paste(t,"-",Res))
+ }
```

```
> a <- c(1,4,5,2,2,6,7)
# Объявляем вектор a
> OddEven(a)
# Все прекрасно работает для векторов
[1] «1 - Odd»  «4 - Even» «5 - Odd»  «2 - Even»
«2 - Even» «6 - Even» «7 - Odd»
```

Тем не менее, наличие sapply в нашем арсенале позволяет справляться с любыми функциями, поддерживающими или не поддерживающими векторные обращения, тем более, что конструкция ifelse иногда тоже не срабатывает для сложных обращений с использованием ряда математических функций (sqrt, например) из-за некорректного распараллеливания.

Задание 8. Функция $f(x)$ задана системой:

$$f(x)=\begin{cases} x^2, & x\ge 0 \\ 4\sqrt{-x}, & x<0 \end{cases},$$

а функция $g(x)$:

$$g(x)=\begin{cases} \operatorname{arctg} x, & x<1 \\ \dfrac{\pi}{4}\sin(\pi x/2), & 1\le x\le 2 \\ 2-x, & 2<x \end{cases}$$

На отрезке $[-3, 3]$ построить графики функций $f(x)$, $g(x)$ и $f(g(x))$.

Решение. С первыми двумя графиками особых проблем не возникает (см. также рис. 95).
$f(x)$:

```
f <- function(x) {              # Вводим функцию f(x)
  A <- (x<0)                    # Создаем массив из "TRUE"
  # и "FALSE " для x < 0
  B <- (x >=0)                  # Создаем массив из "TRUE"
  # и "FALSE " для x >= 0
  x[A] <- 4*sqrt(-x[A])         # Вычисляем левую ветвь
  # функции
```

```
  x[B] <- x[B]^2          # Вычисляем правую ветвь функции
  Res <- x                # Записываем измененный x в пере-
  # менную Res
  return(Res)             # Возвращаем Res как результат f(x)
}
x <- seq(-3,3,length.out = 1001)          # Разбиваем отрезок
  # [-3; 3] на 1000 частей
y <- f(x)                 # Вычисляем значения функции f(x)
  # во всех точках
plot(x,y,type = "l",lwd = 2,col="blue")   # Строим график
  # y=f(x)
abline(h = 0, v = 0, col = "gray40")      # Рисуем оси
  # координат
```

Замечание. Можно было бы обойтись и без определения B <- (x >= 0), т.к. B=!A, то есть B есть «не A». Просто везде вместо B можно было бы написать !A.

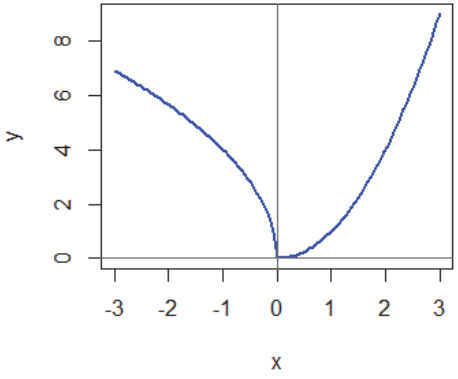


Рисунок 95

$g(x)$ (см. также рис. 96):

```
g <- function(x)              # Вводим функцию g(x)
{
 A <- (x<1)                   # Создаем массив из "TRUE "
  # и "FALSE " для x < 1
```

```
 B <- (x>=1)&(x<=2)          # Создаем массив из "TRUE"
  # и "FALSE" для (x >= 1) и (x <= 2)
 D <- (x>2)                  # Создаем массив из "TRUE"
  # и "FALSE" для x > 2
 x[A] <- atan(x[A])          # Вычисляем левую ветвь
  # функции
 x[B] <- pi*sin(pi*x[B]/2)/4 # Вычисляем середину
  # функции
 x[D] <- x[D] - 2            # Вычисляем правую
  # ветвь функции
 Res <- x                    # Записываем изменен-
  # ный x в переменную Res
 return(Res)          # Возвращаем Res как результат g(x)
}
x <- seq(-3,3,length.out = 1001)  # Разбиваем отрезок [-3;
  # 3] на 1000 частей
y <- g(x)                   # Вычисляем значения
  # функции g(x) во всех точках
plot(x,y,type = "l",lwd = 2,col="brown")  # Строим гра-
  # фик y=g(x)
abline(h = 0, v = 0, col = "gray40")      # Рисуем оси
  # координат
```

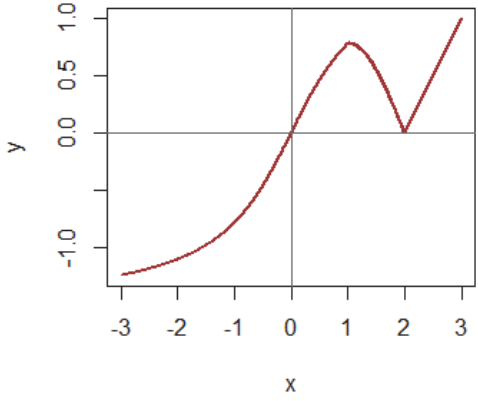


Рисунок 96

Осталось самое интригующее: как технично образовать сложную функцию $f(g(x))$? Мы приготовили себе в этом месте большой подарок, но не будем спешить его «распаковать».

1 Способ: Воспользуемся в первом варианте универсальной процедурой *sapply*. Мы уже объявили функции $f(x)$ и $g(x)$, и какие бы они не были, нам фактически осталось изменить x на $y=g(x)$, а полученный вектор y изменить на $z=f(y)$:

```
x <- seq(-3,3,length.out = 1001)          # Разбиваем отрезок
  # [-3; 3] на 1000 частей
y <- sapply(x, function(t) g(t))          # Вычисляем значе-
  # ния g(x) во всех точках x
z <- sapply(y, function(t) f(t))          # Вычисляем значе-
  # ния  f(y) во всех точках y=g(x)
plot(x,z, type = "l",lwd = 2,col="red")   # Строим график
  # z=f(g(x))
abline(h = 0, v = 0, col = "gray40")      # Рисуем оси коор-
  # динат
```

Но можно было бы обойтись и без *sapply*, т.к. мы запрограммировали качественные функции, поддерживающие векторизованные обращения:

2 Способ (подарок):

```
x <- seq(-3,3,length.out = 101)           # Разбиваем отрезок
  # [-3; 3] на 100 частей
z <- f(g(x))              # Образуем сложную функцию сразу...
plot(x,z, type = "l",lwd = 2,col="red")   # Строим график
  # z=f(g(x))
abline(h = 0, v = 0, col = "gray40")      # Рисуем оси коор-
  # динат
```

Согласитесь, это очень эффектно – сохранить семантику математического анализа во фразе z <- f(g(x)).

Итог один и тот же и совсем не очевиден изначально (см. рис. 97):

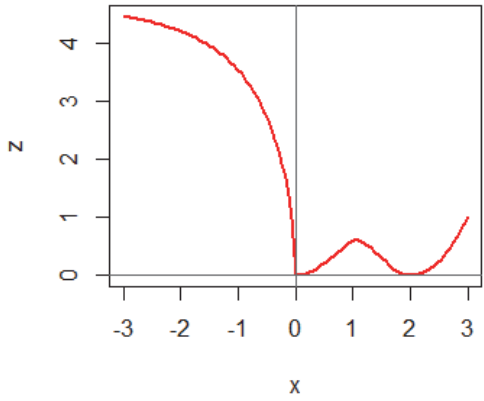


Рисунок 97

Задание 9. Заменить все нечетные значения массива

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	1	2	3	4	5
[2,]	6	7	8	9	10
[3,]	11	12	13	14	15
[4,]	16	17	18	19	20

их квадратами.

Решение. В первую очередь образуем сам массив данных:

```
M <- array(1:20, dim = c(4,5))        # Объявляем массив M
M                                     # Убеждаемся, что ошиблись
```

Однако, если посмотреть результат, то выяснится, что по умолчанию массив заполнен неправильно:

```
> M <- array(1:20, dim =c(4,5))
> M
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
[2,]    2    6   10   14   18
[3,]    3    7   11   15   19
[4,]    4    8   12   16   20
```

Мы видим, что заполнение массива происходит по строчкам, а не по столбцам. В матрицах (тип matrix) есть для этого специальный управляющий параметр *byrow*, но для работы с массивами такого параметра нет. Для того, чтобы числа 1,..,20 заполняли массив не по строчкам, а по столбцам нам придется заполнить транспонированный массив по строчкам, а потом его обратно транспонировать. Операция транспонирования – это взаимная замена строк на столбцы (рис. 98).

```
Trans <- array(1:20, dim =c(5,4))      # Объявляем вспомо-
  # гательный массив Trans
Trans                                  # Смотрим: что получилось
M <- t(Trans)                          # Транспонируем (переворачива-
  # ем) результат
M                                      # Смотрим: теперь все правильно
```

Если не совсем понятна в этом месте операция транспонирования, то мы еще обсудим ее в алгебре матриц. В конце концов, такой массив M можно было ввести и вручную.

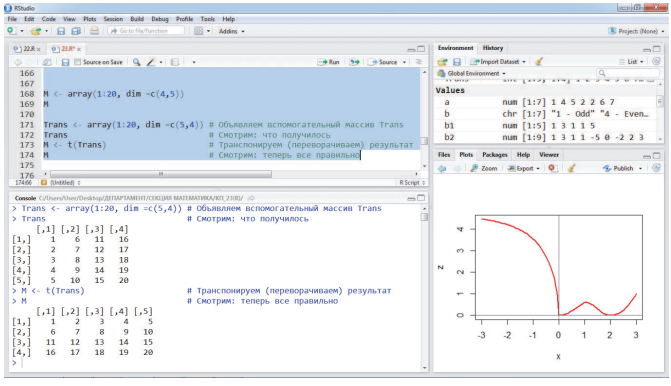


Рисунок 98

Теперь нам необходимо исправить все нечетные числа массива на их квадраты. Образуем функцию, которая заменяет нечетное число его квадратом:

```
Odd_Sq <- function(t) {                    # Объявляем функцию
  # Odd_Sq
 if (t%%2 == 1) {return(t^2)}        # Если t нечетное - воз-
  # вращаем его квадрат
      else {return(t)}                    # Если t четное - остав-
  # ляем без его изменения
}
Odd_Sq(4)                                  # Проверяем на 4
Odd_Sq(7)                                  # Проверяем на 7
Odd_Sq(1:4)      # Убеждаемся, что с векторами не работает
```

```
> Odd_Sq(4)                                # Проверяем на 4
[1] 4
> Odd_Sq(7)                                # Проверяем на 7
[1] 49
> Odd_Sq(1:4)        # Убеждаемся, что с векторами
не работает
[1]  1  4  9 16
Warning message:
In if (t%%2 == 1) { :
  the condition has length > 1 and only the first
element will be used
```

С векторными обращениями наша функция, конечно, не работает, но нас это не беспокоит, т.к. мы уже умеем применять любую функцию к элементам вектора – применим с помощью sapply функцию Odd_Sq ко всему массиву M:

```
Modif <- sapply(M, function(t) Odd_Sq(t))     # Применяем
  # Odd_Sq ко всем элементам M
Modif                              # Смотрим, что не совсем то...
```

```
> Modif <- sapply(M, function(t) Odd_Sq(t))    #
Применяем Odd_Sq ко всему M
> Modif                                        #
Смотрим что не совсем то...
```

```
 [1]   1   6 121  16   2  49  12 289   9   8 169
18   4  81  14 361  25  10 225
[20]  20
```

Практически все верно, но потеряна размерность исходного объекта M. Дело в том, что процедура *sapply* представляет любой массив в виде вектора и оперирует с его элементами. Эту деталь несложно исправить, приписав полученному ответу Modif размерность исходного объекта M:

```
dim(Modif) <- dim(M)          # Возвращаем Modif раз-
  # мерность M
Modif                         # Любуемся хорошей работой!
```

```
> dim(Modif) <- dim(M)
# Возвращаем Modif размерность M
> Modif                 # Любуемся хорошей работой!
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    9    4   25
[2,]    6   49    8   81   10
[3,]  121   12  169   14  225
[4,]   16  289   18  361   20
```

Задания для самостоятельной работы

1. Образовать матрицу, каждый элемент которой равен сумме номеров своей строки и столбца. Матрицу представить в виде массива размера 5x5 и вывести на экран. Указание: воспользуйтесь циклом for.

2. Программно переставить координаты вектора $\vec{b}$

$$\vec{b} = (1,3,6,8,11,NA,10,9,7,5,2,2,2,0,0)$$

в обратном порядке следования.

3. Написать код программы, которая вычисляла бы сумму элементов произвольного вектора $\vec{d}$, занима-

ющих нечетные места. Результат проверить на примере вектора

$$\vec{d} = (4,-3,6,8,11,0,5,9,17,5,3,2,-1,0,4,12).$$

4. Составьте процедуру, которая модифицировала бы элементы произвольного целочисленного массива следующим образом: все единицы заменяла бы словом «One», двойки – словом «Two», а остальные числа оставляла бы без изменения. Проверить работоспособность процедуры на нескольких характерных примерах.

5. Функция $f(x)$ задана системой:

$$f(x) = \begin{cases} \cos x, & x \le 0 \\ 1 + \sqrt[2]{x}, & 0 < x \le 4 \\ 3 - \ln(x-3), & 4 < x \end{cases}$$

а функция $g(x)$:

$$g(x) = \begin{cases} -x, & x \ge 1 \\ x^2 - 2, & x < 1 \end{cases}$$

На отрезке $[-6, 6]$ построить графики функций $f(x)$, $g(x)$ и $f(g(x))$.

6. Внимательно изучив действие программы:

```
plot(0:10,0:10, type="n")
for(k in 1:9){
   text(k,k, paste(k), col="blue", cex = 2)
   Sys.sleep(1.0)          # системное прерывание на 1.0 sec
}
```

добейтесь того, чтобы на экране появлялись не числа, а первые буквы латинского алфавита, причем, красного цвета.

Указание: образуйте вектор a("A", "B", "C", "D", "E", "F" , "G", "H", "I").

Практикум 9. Численное решение дифференциальных уравнений в R (RStudio)

Обратимся к еще одному применению вычислительных компьютерных средств в решении математических задач. В данном разделе мы познакомимся с общей идеей численных методов решения дифференциальных уравнений на примере схемы Эйлера и ее реализации на языке R.

Отметим, что на практике дифференциальные уравнения в подавляющем большинстве случаев решают именно численными методами, но их корректное применение всегда требует специального теоретического исследования сходимости разностных схем и выходит далеко за рамки нашего обсуждения.

Мы же остановимся лишь на вычислительном аспекте применения схемы Эйлера и прокомментируем основные теоретические проблемы по возможности на простых примерах.

Задача Коши

Задание 1. Изобразить на плоскости семейство интегральных кривых для уравнения $y' = \frac{1}{1+x^2}$ и вы-

делить среди них частное решение, удовлетворяющее начальному условию $y(0)=0$.

Решение. Рассмотрим заданное обыкновенное дифференциальное уравнение первого порядка:

$$y' = \frac{1}{1+x^2}.$$

Общее решение (совокупность всех частных решений) этого уравнения несложно получить путем непосредственного интегрирования его правой и левой частей:

$$y = \int \frac{1}{1+x^2} dx = \operatorname{arctg} x + C.$$

Если теперь на плоскости (x,y) изобразить графики частных решений (интегральные кривые) при различных значениях константы C, то получим следующую ниже картину, причем жирным выделено решение, проходящее через начало координат, т.е. удовлетворяющее нашему начальному условию $y(0)=0$ и фактически оказывающееся при $C=0$: $y(x)=\operatorname{arctg} x$ (см. рис. 99).

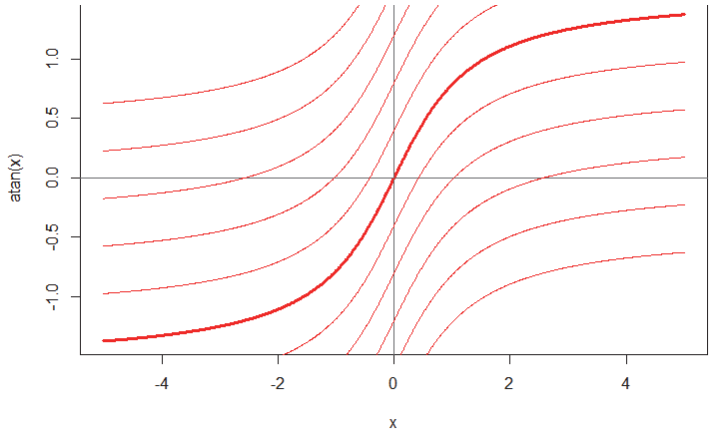


Рисунок 99

Код процедуры построения данного рисунка на R здесь несложный:

```
x <- seq(-5, 5, by = 0.01)     # Задаем последовательность x
plot(x,atan(x), type = "l", lwd = 3, col = "red")     # Рисуем
  # arctg(x) жирной линией (lwd = 3)
abline(h = 0, v = 0, col = "gray40")                  # Рисуем
  # оси координат
for (C in seq(-2, 2, by=0.4)) {                       # Выбираем
  # последовательность для C
 lines(x,atan(x)+C, type = "l", lwd = 1, col = "red")
# Рисуем другие интегральные кривые
}
```

Заметим, что по сути мы решили задачу Коши

$$\begin{cases} y' = \dfrac{1}{1+x^2} \\ y(0) = 0 \end{cases}$$

определив её точное единственное решение $y(x) = \operatorname{arctg} x$.

В следующем задании получим это решение с помощью приближенного метода.

Схема Эйлера

Идея применения схемы Эйлера численного решения задачи Коши для дифференциального уравнения первого порядка:

$$\begin{cases} y' = f(x;y) \\ y(x_0) = y_0 \end{cases}$$

связана с заменой входящей в уравнение производной y' ее разностным аналогом. Поясним подробнее о чем, собственно, идет речь и в каком смысле мы собираемся искать приближенное решение.

В первую очередь производится разбиение отрезка, на котором ищется решение, на n равных между собой частей, т.е. образуется конечная совокупность x_i, её еще называют сетью, а сами точки – узлами сети:

$$x_0, x_1, x_2, \ldots, x_n.$$

Отметим, что расстояние между соседними точками при этом одинаково и равно:

$$dx = x_{i+1} - x_i = \frac{x_n - x_0}{n}.$$

Вторым шагом записывают разложение точного решения $y(x_{i+1})$ в ряд Тейлора в точке x_i:

$$y(x_{i+1}) = y(x_i) + y'(x_i)dx + \frac{y''(x_i)}{2!}dx^2 + \ldots$$

и отбрасывают слагаемые старше первой степени dx. Здесь-то и происходит возникновение приближенного равенства:

$$y(x_{i+1}) \approx y(x_i) + y'(x_i)dx.$$

Ясно, что чем меньше выбор шага dx, тем точнее приближение. Далее из полученного приближения нетрудно выразить производную в i-ом узле сети $y'(x_i)$:

$$y'(x_i) \approx \frac{y(x_{i+1}) - y(x_i)}{dx},$$

что, конечно, не может не напоминать определение производной при малых dx – это и есть разностный аналог производной. Таким образом, если подставить в уравнение $y' = f(x;y)$ полученное выражение для производной $y'(x_i)$ в узле сети x_i:

$$y'(x_i) \approx \frac{y(x_{i+1}) - y(x_i)}{dx} \approx f(x_i; y(x_i)),$$

то получится некоторое рекуррентное соотношение между значениями неизвестной функции $y(x)$ в соседних узлах сети:

$$y(x_{i+1}) \approx y(x_i) + dx \cdot f(x_i; y(x_i)).$$

Мы не будем переобозначать неизвестную функцию $y(x_i)$ на ее приближенный аналог, но со следующей строчки мы вернем точный знак равенства, и это будет верно уже исключительно для приближенного решения, а не точного:

$$y(x_{i+1}) = y(x_i) + dx \cdot f(x_i; y(x_i)).$$

Данный тип уравнений называют разностным уравнением. Решается такое уравнение достаточно просто, т.к. для нахождения $y(x_{i+1})$ необходимо всего лишь знать предыдущее значение $y(x_i)$. Поэтому сначала задается начальное условие $y(x_0)=y_0$ и с его помощью находят последующее значение $y(x_1)$:

$$y(x_1)=y(x_0)+dx\cdot f\left(x_0;y(x_0)\right).$$

Далее подставляют найденное значение $y(x_1)$, как уже известное, в ту же правую часть рекуррентного соотношения и определяют следующее значение $y(x_2)$ и т.д. пока не доберутся до конца отрезка x_n.

Запишем в итоге получившуюся схему Эйлера в терминах $y_i=y(x_i)$ – приближенного сеточного аналога точного решения:

Итог по схеме Эйлера

Если требуется численно решить задачу Коши

$$\begin{cases} y'=f(x;y) \\ y(x_0)=y_0 \end{cases} \qquad (1)$$

методом Эйлера, то необходимо найти все значения приближенного решения $\{y_i\}$, заданные рекуррентной зависимостью:

$$\begin{cases} y_{i+1}=y_i+dx\cdot f(x_i;y_i), \quad i=1,2,..,n \\ y(x_0)=y_0 \end{cases} \qquad (2)$$

Полученная числовая совокупность значений функции $\{y_i\}$ в узлах сети и будет дискретным (разностным) аналогом решения задачи Коши, а если ее соседние точки на графике соединить отрезками прямых, то такое приближенное решение будет уже непрерывным аналогом решения, называемым ломаной Эйлера.

Прокомментируем конкретное применение схемы Эйлера, ее фактическую суть и «подводные камни» на примерах.

Задание 2. Решить приближенно с помощью схемы Эйлера задачу Коши

$$\begin{cases} y' = \dfrac{1}{1+x^2} \\ y(-5) = \operatorname{arctg}(-5) \end{cases}$$

Решение. Для удобства запишем систему (2) в нашем конкретном случае:

$$\begin{cases} y_{i+1} = y_i + dx \cdot \dfrac{1}{1+x_i^2}, \quad i = 1,2,..,n \\ x_0 = -5; \;\; y_0 = \operatorname{arctg}(-5) \end{cases} \qquad (3)$$

Теперь нам остается аккуратно составить алгоритм вычислений $\{y_i\}$.

1. Будем искать решение справа от начальной точки $x_0 = -5$, скажем, на отрезке $x \in [-5;5]$. Введем переменные концов отрезка x_0 и x_n, длину шага dx, само разбиение отрезка с помощью оператора *seq* и n – общее количество точек разбиения:

```
xo <- -5                          # Начальное значение x
xn <- 5                           # Конечное значение x
dx <- 1.2                         # Величина шага dx
x <- seq(xo, xn, by = dx)         # Разбиение отрезка на xi
n <- length(x)                    # Количество получившихся xi
```

2. Образуем наш численный аналог решения – последовательность значений функции в узлах сети, первоначально заполнив этот массив (вектор) нулями:

```
y <- rep(0, n)                    # Заполняем массив y[i] нулями для
  # всех xi
y[1] <- yo <- atan(-5)            # Задаем начальное значе-
  # ние как первое значение y[1]
```

3. Вычисляем в цикле последующие y[i+1] согласно нашей формуле (3). Обратите внимание, что переменная цикла изменяется не до n, а до $(n-1)$:

```
for (i in 1:(n-1)) {
  y[i+1] <- y[i] + dx/(1 + x[i]^2)
}
```

4. Последним шагом отображаем полученный результат на графике:

```
plot(x,y, type = "l",lwd = 5, col = "blue")    # Рисуем y(x) -
  # численное решение
abline(h = 0, v = 0, col = "gray40")              # Рисуем оси
  # координат
```

Если при этом мы хотим сравнить полученное приближенное решение с точным $y(x) = \operatorname{arctg} x$, то добавим еще пару строк с построением точного решения поверх предыдущего рисунка:

```
t <- seq(xo, xn, by = 0.01)                       # Разбиваем
  # вспомогательную ось ox (~t)
lines(t, atan(t), type = "l", lwd = 4, col = "red")   # Рисуем
  # точное решение
```

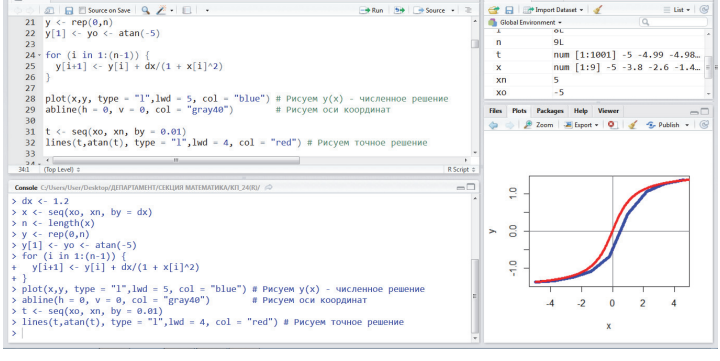


Рисунок 100

Но прежде, чем разбираться с результатом, давайте нанесем еще несколько интегральных кривых на полученный рисунок:

```
for (j in c(0.15,1/2,1/3,-1/2,-1/4,-1,1)) {
  lines(t, atan(t) - j, type = "l", lwd = 1, col = "red")
# Рисуем интегральные кривые
}
```

Теперь смотрим рис. 101:

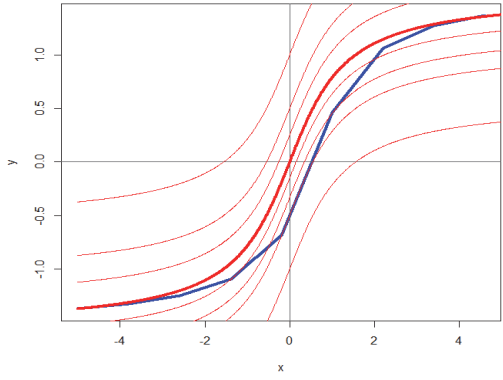


Рисунок 101

Мы специально сделали шаг $dx = 1.2$ – просто огромным, чтобы показать: что происходит в схеме Эйлера. Наше приближенное решение (синий цвет) в каждой итерации, т.е. в каждой точке x_i на каждом изломе принимает вид касательной к графику той интегральной кривой, с которой пересекается в текущем узле сети. Таким образом, первоначально стартует приближенное решение с касательной к настоящему решению, но неизбежно съезжает с него на соседнюю интегральную кривую и в следующей точке уже ориентируется на касательную к другой (пусть даже и очень близкой к исходной) интегральной кривой.

Это один из самых главных недостатков схемы Эйлера. Нет никакой гарантии, что, съехав на соседнюю

интегральную кривую, решение не унесет нас совсем в другую сторону. Конечно, можно сильно уменьшить шаг dx, но все равно есть шанс рано или поздно отклониться сколь угодно далеко от истинного решения. Здесь важную роль играет устойчивость самого решения и взаимная конфигурация поля интегральных кривых (см. задание 4).

Давайте все же уменьшим шаг до приемлемого, уберем соседние интегральные кривые и изменим цвет и толщину графика точного решения так, чтобы было видно попадает наше точное решение в синий коридор приближенного решения или нет:

```
# Пример с y'=1/(1+x^2); y(-5)=arctg(-5) Точное решение
  # y(x) = arctg(x)

xo <- -5                              # Начальное значение x
xn <- 5                               # Конечное значение x
dx <- 0.01                            # Величина шага dx
x <- seq(xo, xn, by = dx)             # Разбиение отрезка на xi
n <- length(x)                        # Количество получив-
  # шихся точек разбиения xi
y <- rep(0,n)    # Заполняем массив y[i] нулями для всех xi
y[1] <- yo <- atan(-5)                # Задаем начальное зна-
  # чение как первое значение y[1]
for (i in 1:(n-1)) {
 y[i+1] <- y[i] + dx/(1 + x[i]^2)      # Программируем пра-
  # вую часть уравнения
}
plot(x,y, type = "l", lwd = 5, col = "blue")          # Рисуем
  # y(x) - численное решение
abline(h = 0, v = 0, col = "gray40")                  # Рисуем
  # оси координат
t <- seq(xo, xn, by = 0.01)
lines(t,atan(t), type = "l",lwd = 1, col = "yellow")  # Ри-
  # суем точное решение
```

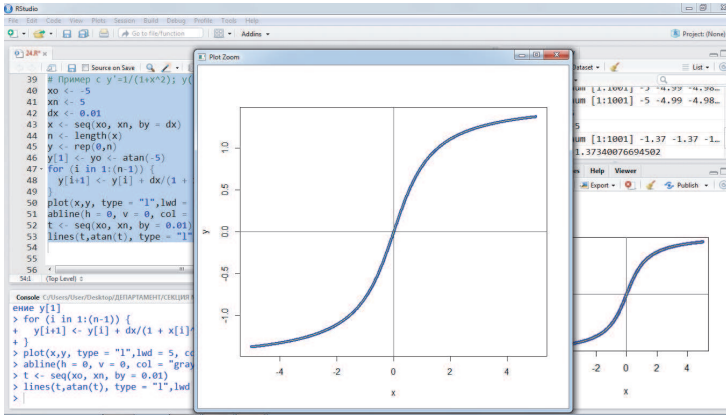


Рисунок 102

Если увеличить масштаб отображения окна (Ctrl + колесо мыши), то можно увидеть «точное» попадание желтой кривой в синий коридор.

Задание 3. Решить приближенно с помощью схемы Эйлера задачу Коши

$$\begin{cases} y' + y = 1 + x \\ y(-2) = e^2 - 2 \end{cases}$$

и сравнить графически полученный результат с точным решением. Найти минимум точного и приближенного решения и сравнить их значения.

Решение. Заметим, что само дифференциальное уравнение здесь является линейным неоднородным первого порядка с постоянными коэффициентами и может быть легко решено одним из трех методов: методом Бернулли, методом вариации постоянной или методом неопределенных коэффициентов (проделайте это самостоятельно). Общее решение принимает вид:

$$y(x) = x + Ce^{-x},$$

а подстановка начального условия $x_0 = -2;\ y_0 = e^2 - 2$ приводит к уравнению на C:

$$-2 + Ce^2 = e^2 - 2,$$

из которого $C = 1$ и точное решение задачи Коши получается следующим:

$$y(x) = x + e^{-x}.$$

Теперь решим задачу численно. Будем искать приближенное решение на отрезке, например, $x \in [-2;7]$. Для удобства запишем систему (2) в нашем конкретном случае:

$$\begin{cases} y_{i+1} = y_i + dx \cdot (1 + x_i - y_i), \quad i = 1,2,..,n \\ x_o = -2; \quad y_o = e^2 - 2 \end{cases}$$

и внесем необходимые изменения в итоговый код программы для задания 2 (выделено синим жирным):

```
# Пример с y'+ y  = 1 + x ; y(-2)= -2 + exp(2)  Точное реше-
  # ние y(x) =  x + exp(-x)

xo <- -2                              # Начальное значение x
xn <- 7                               # Конечное значение x
dx <- 0.01                            # Величина шага dx
x <- seq(xo, xn, by = dx)             # Разбиение отрезка на xi
n <- length(x)                        # Количество получив-
  # шихся точек разбиения xi
y <- rep(0,n)                         # Заполняем массив y[i]
  # нулями для всех xi
y[1] <- yo <- -2 + exp(2)             # Задаем начальное зна-
  # чение как первое значение y[1]
for (i in 1:(n-1)) {
 y[i+1] <- y[i] + dx*(1 + x[i] - y[i])
# Программируем правую часть уравнения
}
plot(x,y, type = "l",lwd = 5, col = "blue")
# Рисуем y(x) - численное решение
abline(h = 0, v = 0, col = "gray40")  # Рисуем оси координат
t <- seq(xo, xn, by = 0.01)
```

```
lines(t, t+exp(-t), type = "l",lwd = 1, col = "yellow")
# Рисуем точное решение
```

В итоге получим прекрасное соответствие точного и приближенного решений для не очень малого шага (см. рис. 103) :

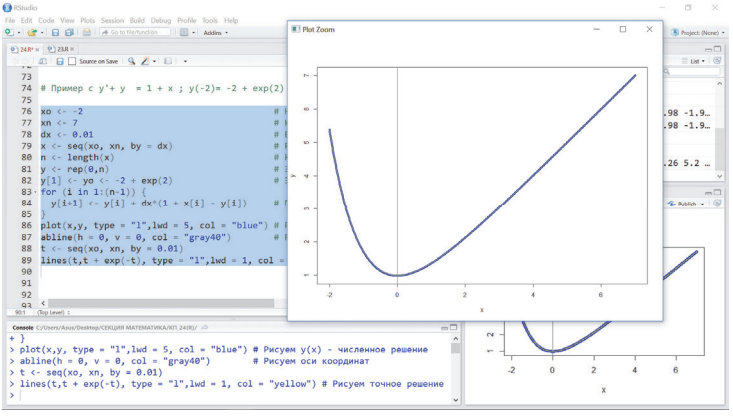


Рисунок 103

Переходя ко второй части задания (определение минимума полученного решения), заметим, что точка истинного минимума для точного решения задачи Коши

$$y(x) = x + e^{-x}$$

определяется из уравнения

$$y'(x) = 1 - e^{-x} = 0$$

и оказывается равна:

$$x_{min} = 0,$$

что дает значение самого минимума:

$$y_{min} = y(0) = 0 + e^{0} = 1.$$

Замечание. Здесь $y''(x_{\min} = 0) = e^{-x}\Big|_{x=0} = 1 > 0$, так что речь идет, действительно, о точке минимума.

Однако найти подобным аналитическим образом минимум приближенного решения не представляется

возможным, т.к. наше приближенное решение задано исключительно как набор значений функции $\{y_i\}$ в узлах сети $\{x_i\}$. Такую дискретную функцию невозможно дифференцировать (в каждой точке она разрывна) и лучшее что остается – просто найти минимальное значение из всех ее сеточных значений.

```
# Нахождение минимума y:
min(y)                  # Минимальное значение среди всех y[i]
```

Но если нам необходимо определить и соответствующее значение самой точки x_{min}, то необходимо отследить для какого номера i достигается это минимальное значение:

```
# Нахождение точки минимума и самого минимума:
i <- which.min(y)       # Номер в массиве разбиений отрез-
  # ка [xo, xn] для минимума
x[i]                    # Точка минимума (xmin)
y[i]                    # Значение минимума (ymin). Мож-
  # но ввести просто min(y)
```

Кстати, получающееся значения x_{min} и y_{min} более, чем убедительно, совпадают с точными:

```
> x[i]                  # Точка минимума (xmin)
[1] -0.01
> y[i]                  # Значение минимума (ymin).
Можно ввести просто min(y)
[1] 0.9899832
```

Здесь точность определения точки минимума, конечно, не превышает точности самого шага $dx = 0.01$.

Аналогично можно получить максимальное значение путем замены min на max:

```
# Нахождение точки максимума и самого максимума:
i <- which.max(y)           # Номер в массиве разбиений
  # отрезка [xo, xn] для максимума
x[i]                        # Точка максимума (xmax)
y[i]                        # Значение максимума (ymax).
  # Можно ввести просто max(y)
```

Замечание. Функции which.max(y) и which.min(y) в случае нескольких одинаковых экстремумов возвращают все соответствующие номера в виде массива по возрастанию.

Задание 4. Решить приближенно с помощью схемы Эйлера задачу Коши для уравнения Риккати:

$$\begin{cases} y' + y^2 = \dfrac{1}{2} + \cos x + \left(\dfrac{x}{2} + \sin x\right)^2 \\ y(-4) = \sin(-4) - 2 \end{cases}$$

и сравнить графически полученный результат с точным решением задачи:

$$y(x) = \sin x + \frac{x}{2}.$$

Решение. Желающие получить общее решение данного уравнения могут воспользоваться тем, что нам известно одно из его частных решений $y(x) = \sin x + \dfrac{x}{2}$, и свести уравнение с помощью замены $y(x) = z(x) + \sin x + \dfrac{x}{2}$ к уравнению Бернулли.

Мы же, как в предыдущих случаях выберем отрезок и запишем рекуррентную систему для нахождения сеточного решения. Будем рассматривать решение на отрезке $x \in [-4;7]$. Подставляя в (2) правую часть нашего приведенного уравнения:

$$y' = \frac{1}{2} + \cos x + \left(\frac{x}{2} + \sin x\right)^2 - y^2,$$

получим систему:

$$\begin{cases} y_{i+1} = y_i + dx \cdot \left(\dfrac{1}{2} + \cos x_i + \left(\dfrac{x_i}{2} + \sin x_i\right)^2 - y_i^2\right), \quad i = 1,2,..,n \\ x_o = -4; \quad y_o = \sin(-4) - 2. \end{cases}$$

Соответственно это приведет к следующим изменениям кода (см. также рис. 104):

```
# Пример с y'+ y^2 = 1/2 + cos(x) + (x/2+sinx)^2;
  # y(-4)=sin(-4) - 2
# Точное решение y(x) = sin(x) + x/2

xo <- -4                          # Начальное значение x
xn <- 7                           # Конечное значение x
dx <- 0.01                        # Величина шага dx
x <- seq(xo, xn, by = dx)         # Разбиение отрезка на xi
n <- length(x)                    # Количество получив-
  # шихся точек разбиения xi
y <- rep(0, n)                      # Заполняем массив y[i]
  # нулями для всех xi
y[1] <- yo <- -2 +sin(-4)             # Задаем начальное
  # значение как первое значение y[1]
for (i in 1:(n-1)) {
  y[i+1] <- y[i] + dx*(1/2 + cos(x[i]) + (x[i]/2 + sin(x[i]))^2
- y[i]^2)
}
plot(x,y, type = "l",lwd = 5, col = "blue")          # Рисуем
  # y(x) - численное решение
abline(h = 0, v = 0, col = "gray40")                 # Рисуем
  # оси координат
t <- seq(xo, xn, by = 0.01)
lines(t, sin(t) + t/2, type = "l",lwd = 2, col = "red")  # Рису-
  # ем точное решение
```

Однако результат нас не может порадовать из-за существенного отклонения приближенного решения от истинного:

Это как раз тот случай, когда шаг dx оказывается слишком грубым и приближенное решение соскальзывает на уходящие в сторону интегральные кривые. Сделаем несколько попыток последовательного уменьшения шага ... (см. рис. 105):

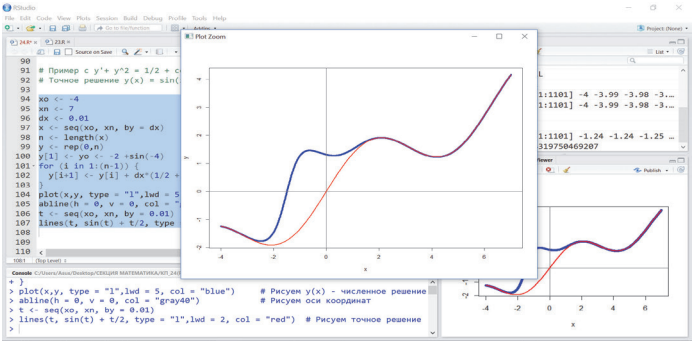


Рисунок 104

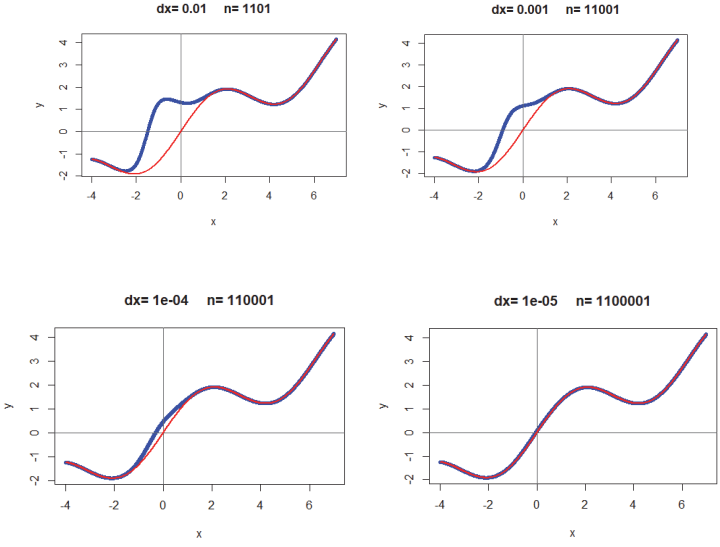


Рисунок 105

В последнем, уже удовлетворительном случае, нам пришлось разбить отрезок более, чем на миллион точек. И это не предел для R в резервировании массивов. Однако десять миллионов элементов массива – это уже возможный предел. В этих случаях необходимо

оптимизировать вычисления с динамической записью результатов в файл или выбирать более короткий отрезок поиска решения.

В заключении необходимо отметить, что на практике точное решение неизвестно и, если говорить о схеме Эйлера, то выбор шага dx остается практически на интуиции исследователя. Обычно последовательно уменьшают шаги, стремясь зафиксировать стабилизацию приближенного решения и, вместе с тем, подставляют близкие начальные значения и изучают соседние интегральные кривые на предмет устойчивости или особенностей.

В настоящее время схемой Эйлера практически не пользуются, однако, быстро и просто «прикинуть» регулярное решение ей всегда можно.

Задание 5.* Исследовать приближенное решение, полученное в предыдущем задании, на локальный максимум в окрестности точки $x = 2$ и локальный минимум в окрестности точки $x = 4$.

Решение. Обратимся еще раз к графику, полученному в задании 4, итогового приближенного решения задачи Коши (см. рис. 106):

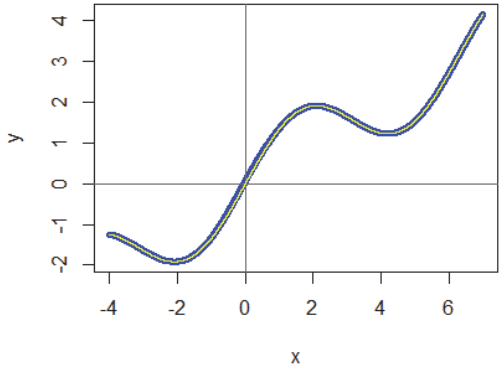


Рисунок 106

Если мы введем код, подобный тому, который мы использовали в задании 3:

```
# Нахождение точки минимума и самого минимума:
i <- which.min(y)              # Номер в массиве разбиений
  # отрезка [xo, xn] для минимума
x[i]                           # Точка минимума (xmin)
y[i]                           # Значение минимума (ymin).
  # Можно ввести просто min(y)

# Нахождение точки максимума и самого максимума:
i <- which.max(y)              # Номер в массиве разбиений
  # отрезка [xo, xn] для максимума
x[i]                           # Точка максимума (xmax)
y[i]                           # Значение максимума
  # (ymax). Можно ввести просто max(y)
```

то в результате получим наибольшее и наименьшее значения приближенного решения на всем диапазоне $i = 1,\ldots,n$, т.е. фактически на всем отрезке $[x_0; x_n]$:

```
> x[i]                         # Точка минимума (xmin)
[1] -2.09615
> y[i]                         # Значение минимума (ymin).
Можно ввести просто min(y)
[1] -1.912823
> x[i]                         # Точка максимума (xmax)
[1] 7
> y[i]                        # Значение максимума (ymax).
Можно ввести просто max(y)
[1] 4.156987
```

а нам требуется определить локальные экстремумы в окрестностях точек

$$x = 2 \quad \text{и} \quad x = 4 .$$

Таким образом, нам необходимо так уменьшить диапазон изменения i, чтобы локальный максимум стал глобальным и локальный минимум стал глобаль-

ным. Естественно, для каждой окрестности и каждого экстремума диапазон будет свой.

Из графика хорошо видно, что на отрезке $x \in [-4;4]$ локальный максимум является и глобальным. Следовательно, можно ограничить изменение i в соответствующем диапазоне $i = 1,\ldots,k$ и применить предыдущую схему:

```
k <- which(x>=4)[1]          # Номер в массиве, до которого
  # ищем максимум
i <- which.max(y[1:k])       # Номер глобального max в мас-
  # сиве разбиений отрезка [xo, 4]
x[i]                         # Точка максимума (xmax)
y[i]                         # Значение максимума (ymax).
  # Можно ввести max(y[1:k])
```

```
> x[i]                       # Точка максимума (xmax)
[1] 2.09279
> y[i]                       # Значение максимума (ymax).
Можно ввести max(y[1:k])
[1] 1.913587
```

С другой стороны, из графика нетрудно видеть, что на отрезке $x \in [3;7]$ искомый локальный минимум является глобальным. Аналогично ограничиваем $i = k,\ldots,n$ и получаем:

```
k <- which(x>=3)[1]              # Номер в массиве, начиная
  # с которого ищем минимум
i <- which.min(y[k:n]) + k - 1   # Номер в массиве разбие-
  # ний отрезка [3, xn]
x[i]                             # Точка минимума (xmin)
y[i]                             # Значение минимума (ymin).
  # Можно ввести min(y[k:n])
```

```
> x[i]                       # Точка минимума (xmin)
[1] 4.18879
> y[i]                       # Значение минимума (ymin).
Можно ввести min(y[k:n])
[1] 1.228369
```

Но почему такие странные условия в первых строчках? Например:

```
k <- which(x>=4)[1]
```

Дело в том, что абсолютно точного значения $x = 4$ в нашем разбиении

```
x <- seq(xo, xn, by = dx)
```

может не оказаться (это зависит от кратности шага). Поэтому мы ищем все номера, для которых x переваливает за 4. Функция which(x>=4) возвращает нам массив таких номеров, из которого мы выбираем первое значение which(x>=4)[1], соответствующее, либо самой точке , либо максимально близкой к ней.

Задания для самостоятельной работы

1. Решить приближенно с помощью схемы Эйлера задачу Коши:

$$\begin{cases} y' = -\sin x \\ y(0) = 1 \end{cases}$$

на отрезке $x \in [0;13]$ и сравнить графически полученный результат с точным решением задачи:

$$y(x) = \cos x.$$

2. Решить приближенно с помощью схемы Эйлера задачу Коши для уравнения Бернулли:

$$\begin{cases} y' - y = xy^3 \\ y(0) = 1 \end{cases}$$

на отрезке $x \in [0;0.63]$ и сравнить графически полученный результат с точным решением задачи:

$$y(x) = \frac{1}{\sqrt{0.5\exp(-2x) - x + 0.5}}.$$

Указание: выбрать необходимый шаг dx.

3. Какую задачу Коши и на каком отрезке численно решает следующий код:

```
xo <- -2
xn <- 2
dx <- 0.1
yo <- sqrt(6)
x <- seq(from = xo, to = xn, by = dx)
n <- length(x); n
y <- rep(0,n)
y[1] <- yo
for (i in 1:(n-1)) {
  y[i+1] <- y[i] + dx*(2*x[i]^3 - 3*x[i]+ 3/4)/y[i]
}
plot(x,y, type = "l",lwd = 5, col = "blue")
abline(h = 0, v = 0, col = "gray40")
lines(x, sqrt(x^4-3*x^2+1.5*x+5), type = "l", lwd = 1,
col = "yellow")
```

Каково точное решение данной задачи Коши? Корректен ли выбор шага в схеме Эйлера? Если некорректен – исправьте на достаточный.

4. * Определить приближенно точки локальных экстремумов и их значения для решения задачи Коши из предыдущего задания. Указание: проверяйте полученные ответы на соответствие графику.

Практикум 10. Задание векторов и матриц в R (RStudio)

Объявление векторов

Напомним, что в языке R во всех именах переменных / объектов / пакетов / функций различаются строчные и прописные буквы, то есть переменные с именами «a» и «A» – разные.

Задание 1. Образовать в R векторы:

$$\vec{a} = \begin{pmatrix} 1 \\ 2 \\ -3 \\ -4 \end{pmatrix}, \vec{b} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \vec{c} = \begin{pmatrix} 0 \\ 2 \\ 4 \\ 6 \end{pmatrix} \text{ и } \vec{d} = \begin{pmatrix} 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}.$$

Решение. Составим следующий текст в левом верхнем поле нового документа (можно просто скопировать эти 5 строчек в область рабочего листа RStudio, но лучше все-таки ввести с клавиатуры):

```
a  = c(1, 2, -3, -4)      # Сформировать вектор a из набора
  # чисел 1,2,-3 и -4
b <- rep(1, 4)            # Сформировать вектор b путем по-
  # втора "1" четыре раза
seq(0, 6, 2) -> c         # Сформировать вектор c как после-
  # довательность от 0 до 6 с шагом 2
```

```
d = 2:5                    # Сформировать вектор d с координа-
  # тами 2, 3, 4, 5
a; b; c; d                 # Вывести в поле консоли значения
  # объектов a, b, c и d
```

Если теперь в RStudio выделить этот фрагмент и нажать мышкой кнопку Run, то снизу на листе консоли отобразятся действия всех выделенных команд, в том числе, и значения образованных четырех векторов (см. рис. 107):

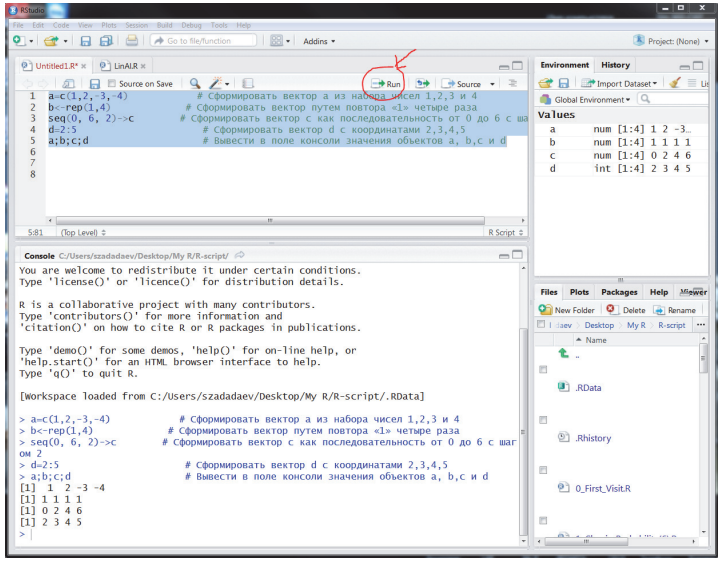


Рисунок 107

Впрочем, текущие значения данных объектов автоматически отображаются на правом верхнем листе «Enviroment». Выделенный текст можно было запустить на компиляцию и с клавиатуры, нажав одновременно Ctrl + Enter. Если нажать это сочетание без предварительного выделения текста кода, то на компиляцию будет отправлена вся строка, где стоит курсор мыши, так называемая *построчечная* компиляция.

Из приведенных примеров нетрудно вспомнить, что оператор присваивания в R допускает три формы: (=), (->) и (<-); оператор c(...) формирует вектор из перечисляемых внутри аргументов (чисел), оператор rep(...) реализует повторы, seq(...) – задает последовательность, а знак решетки (#) является оператором комментария.

Заметим, что лучше не использовать для имени переменной символ «c», т.к. под символом «c» зарезервирована команда объединения в вектор c(...).

Замечание. Не забудьте, что символ «c» одинаков в русской и латинской раскладках и при неаккуратном использовании раскладок ошибку очень сложно заметить, особенно в тех случаях, когда на компьютере используется автоматический переключатель клавиатуры.

Обращение к координатам векторов с целью их считывания или изменения здесь вполне естественно:

```
a[2] <- 10       # Записать во вторую координату ранее
  # объявленного вектора a число 10
```

При наборе длинных выражений или имен функций полезно нажать клавишу Tab, по которой RStudio предложит для выбора возможные продолжения фразы или доступные аргументы набранной функции.

Вообще, нажимайте Tab почаще, чтобы избежать рутинных наборов с клавиатуры.

Укажем еще один вариант задания векторов – наследование из массивов:

```
x <- array(2, dim = c(1,10)); x    # Объявляем одномерный
  # массив x из двоек
s <- as.vector(x); s               # Объявляем вектор s, со
  # ставленный из элементов массива x
```

После выполнения этих двух команд компилятор выдаст в левом нижнем окне консоли сообщения о формировании массива x и вектора s:

```
> x <- array(2,dim = c(1,10)); x    # Объявляем
одномерный массив x из двоек
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
[,10]
[1,]    2    2    2    2    2    2    2    2    2     2

> s <- as.vector(x); s              # Объявляем
вектор s, составленный из элементов массива x
[1] 2 2 2 2 2 2 2 2 2 2
```

Кстати, массив здесь может быть не только одномерным.

Объявление матриц

В пакете R существует достаточно большое количество способов образования матриц, укажем наиболее популярные.

Задание 2. Образовать в R три матрицы:
S – нулевая матрица, размера (10×10);

$$Q=\begin{pmatrix}1 & 1 & 1\\ 1 & 2 & -2\\ 1 & 3 & 0\end{pmatrix} \quad \text{и} \quad P=\begin{pmatrix}-1 & 0 & 1\\ 2 & 2 & 2\\ 4 & 5 & 6\end{pmatrix}.$$

Решение. Составим следующий текст программы:

```
S <- matrix(0, nrow=10, ncol=10)          # Образовать ма-
  # трицу из нулей размера 10x10
Q <- cbind(rep(1,3), 1:3, c(1,-2,0))      # Составить матри-
  # цу из трех столбцов-векторов
P <- rbind(seq(-1,1,1), rep(2,3), 4:6)    # Составить матри-
  # цу из трех строк-векторов
S; Q; P                                   # Вывести в поле
  # консоли значения объектов S, Q и P
```

Обратите внимание на разницу формирования матриц в последних двух случаях rbind и cbind (***row*** – строка, ***col*** – столбец) (см. рис. 108):

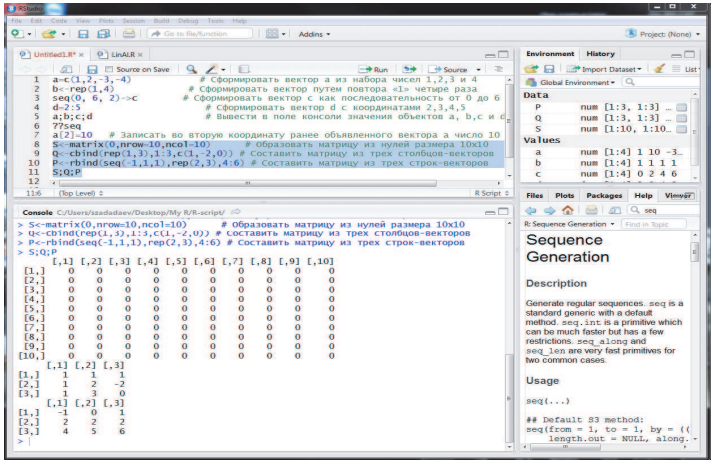


Рисунок 108

Отметим, что числовые массивы фактически уже являются матрицами:

```
x <- array(2,dim = c(3,5)); x      # Объявляем одномерный
  # массив x из двоек
s <- as.matrix(x); s               # Объявляем одномерный
  # массив x вектором s
```

```
> x <- array(2,dim = c(3,5)); x     # Объявляем од-
номерный массив x из двоек
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    2    2    2    2
[2,]    2    2    2    2    2
[3,]    2    2    2    2    2
> s <- as.matrix(x); s                    # Объявляем
одномерный массив x вектором s
     [,1] [,2] [,3] [,4] [,5]
[1,]    2    2    2    2    2
[2,]    2    2    2    2    2
[3,]    2    2    2    2    2
```

Тем не менее, наиболее удобным и универсальным оказывается способ, основанный на конвертации матриц из текстовых файлов или из MS Excel.

Задание 3. Образовать в Excel матрицу:

$$W = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0{,}2 & -0{,}3 & 2{,}7 & 0 \\ 11 & 21 & -3{,}7 & 0{,}03 \end{pmatrix},$$

Загрузить ее в R и образовать из первых трех ее столбцов матрицу A, а из последнего столбца – матрицу (столбец) B.

Решение. Скопируем на рабочий лист R-код «считывания» из буфера обмена данных Excel-формата:

```
# Чтение в переменную Data таблицы данных из буфера
  # обмена excel-формата:
Data <- read.table("clipboard", h=FALSE, dec=",", sep = "\t")
W <- as.matrix.data.frame(Data)          # Объявить в R
  # таблицу чисел Data матрицей W
W                                  # Вывести в поле консоли значение W
```

Но прежде чем запускать этот код на компиляцию, сформируем в файле Excel требуемую числовую матрицу W, выделим ее и скопируем в буфер обмена комбинацией Ctrl+c (см. рис. 109):

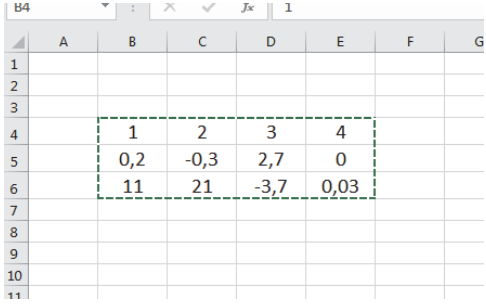


Рисунок 109

Теперь в RStudio выделяем наш код и запускаем Run (см. рис. 110).

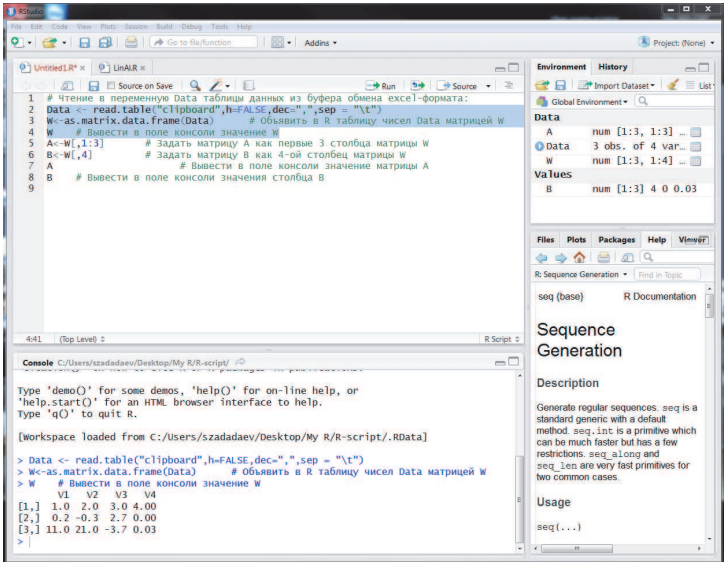


Рисунок 110

В результате переменная *W* содержит нашу Excel-матрицу.

Замечание. Обратите внимание, что десятичным разделителем в Excel является запятая, а в R – точка. Об этом мы особо указали в аргументах функции чтения:

```
Data <- read.table("clipboard", h=FALSE, dec=",", sep = "\t")
```

h = FALSE – отключить заголовки столбцов;
110)",," – десятичным разделителем в данных является запятая (в excel);
sep = "\t" – разделителем чисел в данных является символ табуляции (в excel).

Далее нам остается из уже имеющейся матрицы *W* образовать требуемые матрицы *A* и *B* (*B* – фактически будет являться вектором):

```
A <- W[ ,1:3]       # Задать матрицу A как первые 3 столб-
  # ца матрицы W
B <- W[ ,4]         # Задать матрицу B как 4-ой столбец
  # матрицы W
A                   # Вывести в поле консоли значение матрицы A
B                   # Вывести в поле консоли значения столбца B
```

Обратите внимание, что вектор *B* для экономии места консоли представлен в виде строки, хотя, в действительности, он заявлен как вектор (см. рис. 111):

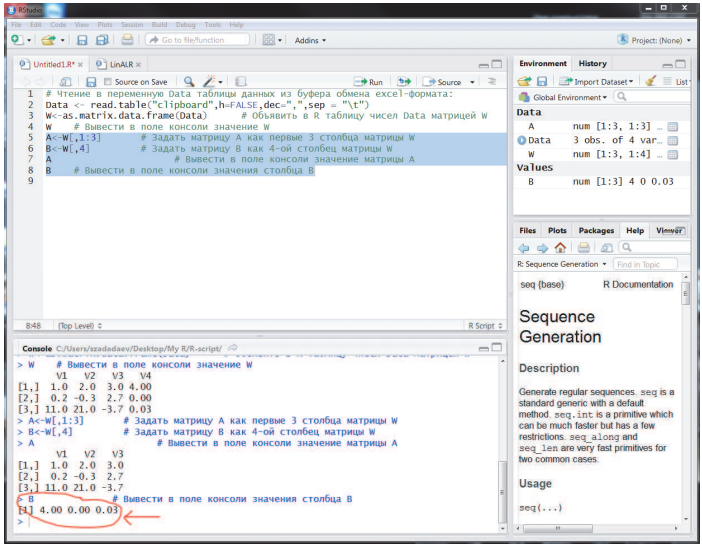


Рисунок 111

В этом нетрудно убедиться, если попытаться отправить матричные результаты расчетов обратно в Excel. Это легко сделать так же через буфер обмена с помощью строки:

```
# Скопировать в буфер обмена объект B в Excel-формате:
write.table(B,"clipboard",quote=FALSE, col.names =
FALSE, row.names = FALSE, sep = "\t", dec=",")
```

Теперь остается не забыть скопировать записанное содержимое буфера обмена обратно в Excel комбинацией Ctrl+v (см. рис. 112):

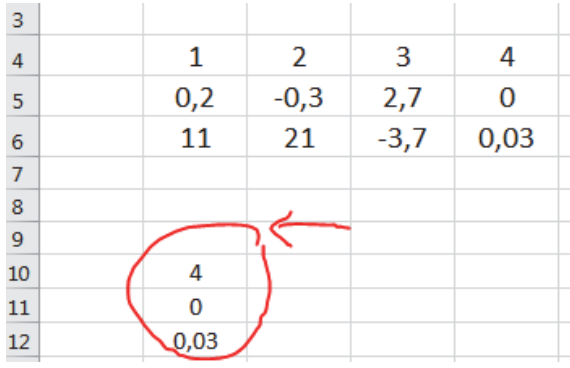


Рисунок 112

Замечание. Вместо «clipboard» во всех функциях чтения/записи может быть указан путь к диску с именем текстового файла. Например, чтобы записать данные переменной Data в текстовый файл aaa.txt необходимо ввести команду:

```
# Запись в файл data.txt содержимого таблицы (матри-
# цы) Data
write.table(Data,"C:/Users/aaa.txt",col.names = TRUE,
row.names = FALSE, quote=FALSE,sep = "\t",dec=",")
```

Здесь важно, что при указании пути к файлу должны использоваться символы (/) или (\\) вместо обычных (\). Такой файл можно открыть и в MS Excel, указав в качестве разделителя знак табуляции. Если вместо пути указано только имя файла, результат будет записан в рабочую директорию RStudio.

Замечание. Получить путь к файлу можно следующим образом: укажите на требуемый файл мышью и, удерживая нажатыми Ctrl+Shift, кликните правой клавишей мыши. Из развернувшегося меню необходимо выбрать пункт «Копи-

ровать как путь» – теперь в буфере обмена находится путь к файлу, его можно скопировать в нужное место кода RStudio по Ctrl+v и не забыть заменить все символы (\) на (/) или на (\\).

Замечание. Не забываем, что для более детального изучения всех возможностей каких-либо операторов или пакетов языка R необходимо нажать клавишу F1 при поставленном курсоре перед нужной командой или составить команду по следующему правилу:

```
# Вызов справки, например, по функции seq.
??seq
# или
help(seq)
```

Задания для самостоятельной работы

1. Образовать и вывести на экран вектор a, содержащий 50 одинаковых координат, равных 3.

2. Образовать и вывести на экран вектор b, содержащий 100 координат 1,2,..,100.

3. Образовать квадратную матрицу A, состоящую из последовательности чисел 1,2,...,16, которая заполняет матрицу

a) по столбцам;

b) по строкам.

Подсказка: при вызове команды «matrix» использовать специальный параметр «byrow», о котором можно узнать из справки по функции «matrix».

4. Решить предыдущие упражнения 1–3 с помощью MS Excel и буфера обмена.

5. Образовать и вывести на экран консоли матрицу *G* размера (10x10), элементы которой равны сумме соответствующих строки и столбца: $G_{ij} = i + j$;

6. Образовать в R единичную матрицу размера (40x40) и перенести ее через буфер обмена в excel- таблицу.

Практикум 11. Сохранение результатов в R и импорт/экспорт данных из Excel (RStudio)

В этом разделе мы обсудим возможные коммуникации R с данными MS Excel и общие способы хранения/записи результатов работы в R.

Команда read.table

Напомним, что универсальным средством обмена данными между Excel и R может выступать буфер обмена. Так, команда read.table:

```
Data <- read.table("clipboard", h=TRUE, dec=",", sep = "\t")
# Чтение из буфера обмена данных excel-формата в таб-
  # лицу data.frame с заголовками столбцов
```

производит чтение данных Excel-формата из буфера обмена в переменную Data типа data.frame, при этом параметр h отвечает за то, требуется ли считать первую строку таблицы именами столбцов или нет.

Обратная команда write.table:

```
write.table(Data, "clipboard", quote=FALSE, col.names =
TRUE, row.names = FALSE, sep = "\t", dec=",")
# Запись в буфер обмена данных в Excel-формате
```

производит запись данных таблицы Data из R в буфер обмена в Excel- формате.

Напомним также, что вместо буфера обмена «clipboard» может стоять полный адрес к текстовому файлу, записанный прямым слешом (/), например, «C:/Users/User/Desktop/data.txt».

На практике в большинстве простых задач этого приёма вполне хватает, но если цель состоит в автоматизации объемных и постоянно возникающих рутинных вычислений, то нам потребуются более мощные средства, чем буфер обмена.

Процедура read.csv или read.csv2

Действие указанной в заголовке процедуры аналогично read.table, но применяется она для чтения данных из Excel-файлов с расширением .csv (файлы, данные которых разделены запятой). В России с такими файлами особо не работают из-за того, что у нас запятая – это десятичный разделитель, а не разделитель данных. Конечно, можно искусственно такой «российский» файл сохранить под расширением .csv, но это все-равно не сделает его подходящим.

Для того, чтобы файл стал действительно с разделяющими запятыми, а десятичным разделителем была бы у него точка – требуется существенная перенастройка MS Excel, что не всегда удобно для обычной повседневной работы.

Вывод прост: используйте эти команды (без разницы read.csv или read.csv2) только для чтения аутентичных csv-файлов из европейских или американских источников:

```
Data <- read.csv("Путь и имя файла.csv", header =TRUE)
# Чтение данных из excel-файла формата csv
write.csv(Data, "Путь и имя файла.csv ", col.names =
TRUE, row.names = FALSE)   # Запись таблицы Data
  # из R в Excel-файл формата csv
```

Замечание. Здесь и далее в указанных примерах, разумеется, можно менять значения параметров, а справка по их исчерпывающему перечню, как обычно, доступна по F1.

Замечание. Разница между операторами read.csv и read.csv2 следующая: read.csv – оперирует с разделителями данных в виде запятых, а read.csv2 – с разделителями данных в виде точки с запятой, однако, десятичным разделителем у них всегда используется точка!

Библиотека «xlsx»

Все, о чем мы будем говорить в этом и следующих подразделах относится к Excel 2016, точнее к Excel-файлам с расширением xlsx.

Но прежде, чем приступить к обсуждению команд чтения / записи таких файлов, необходимо установить мощную библиотеку «xlsx», которая в свою очередь не будет корректно работать без предварительной установки на компьютер объектно-ориентированного языка программирования Java. Мы, конечно, не собираемся сейчас учиться программировать на Java, но библиотека «xlsx» использует объекты Java и без установки последней не обойтись.

Как проверить: стоит ли на компьютере Java? Просто начнем устанавливать пакет «xlsx» (можно стандартно из меню Tools – Install Packages...) и посмотрим на сообщения об ошибках. Если ошибок нет – Java присутствует.

В противном случае устанавливаем пакет Java, пройдя последовательно по двум ссылкам ниже, но не забудьте при этом выбрать соответствующую операционную систему и принять все умолчания, нажав после вопросов «ok»:

1. https://java.com/ru/download/
2. http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Теперь снова повторяем установку пакета «xlsx» (на этот раз уже удачную).

Создадим новый документ в R и подключим две необходимые нам в дальнейшем библиотеки (см. рис. 113):

```
# Загружаем библиотеки:
library("xlsx")          # Для считывания данных из
  # excel-файлов типа xlsx
library("dplyr")         # Для glimpse и select
```

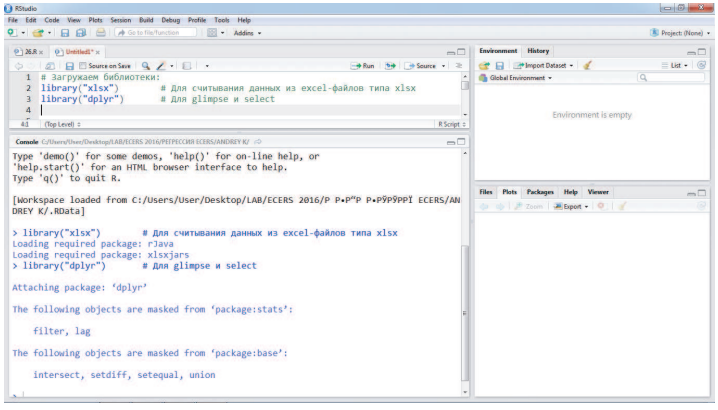


Рисунок 113

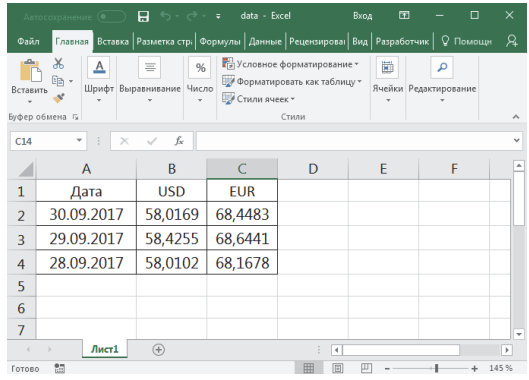
Процедура read.xlsx

Задание 1. Создать Excel-файл, содержащий на первом листе следующую таблицу (см. рис. 114).

Сохранить его на рабочем столе под именем data.xlsx и загрузить его данные в R без использования буфера обмена.

Решение. Введем указанную таблицу в ячейки файла и сохраним его как книгу Excel (см. рис. 115):

Далее нам необходимо прописать путь к этому файлу. Самый техничный способ – это указать мышью на файл data.xlsx и, удерживая нажатыми клавиши **Ctrl** и **Shift**, нажать правую кнопку мыши, после чего выбрать пункт «Копировать как путь» (см. рис. 116).



	A	B	C	D	E	F
1	Дата	USD	EUR			
2	30.09.2017	58,0169	68,4483			
3	29.09.2017	58,4255	68,6441			
4	28.09.2017	58,0102	68,1678			
5						
6						
7						

Рисунок 114

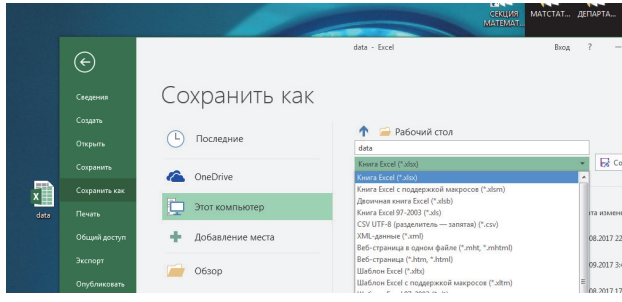


Рисунок 115

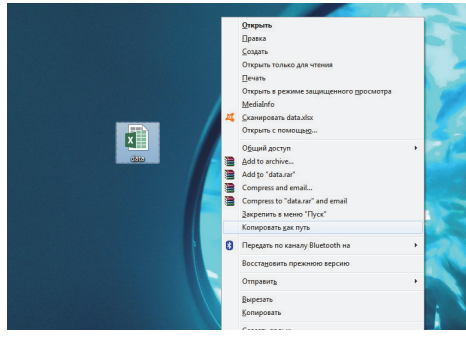


Рисунок 116

В итоге в буфере обмена находится путь к нашему файлу. Однако вставить его в соответствующую команду R нужно с некоторыми изменениями. Образуем в R текстовую переменную PATH, в которую скопируем из буфера обмена наш путь (Ctrl+v):

```
PATH <- "C:\Users\User\Desktop\data.xlsx"      # Непра-
  # вильный путь к файлу из буфера!!!
```

и обязательно вставим к каждому обратному слешу (\) еще такой же:

```
PATH <- "C:\\Users\\User\\Desktop\\data.xlsx"  # Кор-
  # ректный путь к файлу
```

Это и есть одна из корректных форм записи пути в R.

Теперь нам остается задать саму команду чтения таблицы данных с первого листа Excel-файла, путь к которому мы указали в переменной PATH:

```
Data <- read.xlsx(PATH, 1, encoding = "UTF-8", header =
TRUE)  # Считывается 1-ый лист из указанной книги
  # Excel
```

Интересно, что если бы имя нашего листа было написано латиницей: не «Лист1», а скажем, «List1», то мы могли бы обратиться к нему не по номеру (индексу), а по имени:

```
Data <- read.xlsx(PATH)"List1", encoding = "UTF-8",
header = TRUE)  # Считывается лист "List1" из указан-
  # ной книги Excel
```

С именами листов на кириллице будет выдаваться ошибка, но всегда можно обратиться по номеру листа, хотя и нежелательно, т.к. листы в файле могут быть однажды переставлены.

Посмотрим результат (см. также рис. 117):

```
glimpse(Data)      # Смотрим структуру Data
Data               # Смотрим саму таблицу данных Data
```

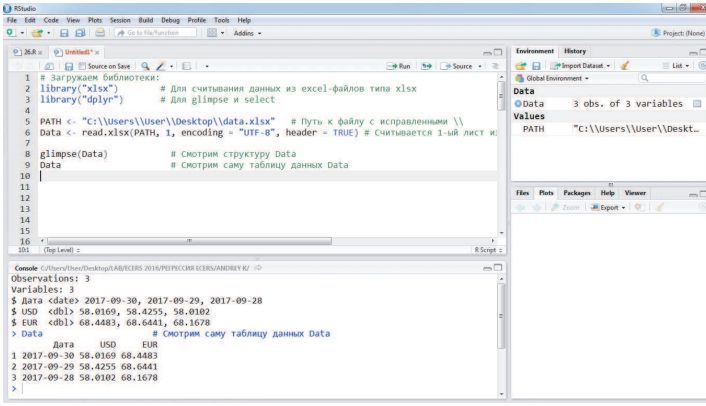


Рисунок 117

Обратите внимание, что glimpse(Data) сообщает о наличии трех столбцов в таблице Data (тип data.frame): Data$Дата, Data$USD и Data$EUR.

Также можно заметить, что десятичные разделители из запятых превратились в точки, а даты оказались автоматически распознаны в R, и их формат изменен на европейский.

Процедура write.xlsx, а лучше write.xlsx2

Задание 2. Переставить в R второй и третий столбцы таблицы Data из предыдущего задания и записать полученную таблицу на новый лист файла data.xlsx.

Решение. Образуем новую таблицу Data2, равную Data с переставленными столбцами:

```
Data2 <- select(Data, Дата, EUR, USD)  # Выбираем
  # в новую таблицу Data2 нужные столбцы из таблицы
  # Data в указанном порядке:  Дата, EUR, USD
Data2                                  # Смотрим что получилось
```

с отчетом в R-консоли:

```
> Data2 <- select(Data, Дата, EUR, USD)
```

```
> Data2
        Дата      EUR      USD
1 2017-09-30 68.4483 58.0169
2 2017-09-29 68.6441 58.4255
3 2017-09-28 68.1678 58.0102
```

Теперь запишем данные Data2 в существующий файл data.xlsx на новый лист с именем «New»:

```
write.xlsx2(Data2, PATH, sheetName="New", col.
names=T, row.names=F, append=TRUE, showNA=FALSE)
# Запись Data2 в существующий файл
```

Важно, чтобы при запуске этой команды сам файл, указанный в пути PATH, был закрыт. Если файл будет открыт в Excel, R не сможет записать в него данные и выдаст ошибку. Так же к ошибке приведет и попытка записать данные на уже существующий лист данного файла. Замечание. Эту команду фактически нельзя запускать два раза подряд, т.к. после первого раза будет создан лист «New» и для второго запуска лист «New» уже не будет новым.

В нашем случае все сработало корректно и после открытия файла data.xlsx убеждаемся в появлении нового листа «New» с заявленными данными (см. рис. 118):

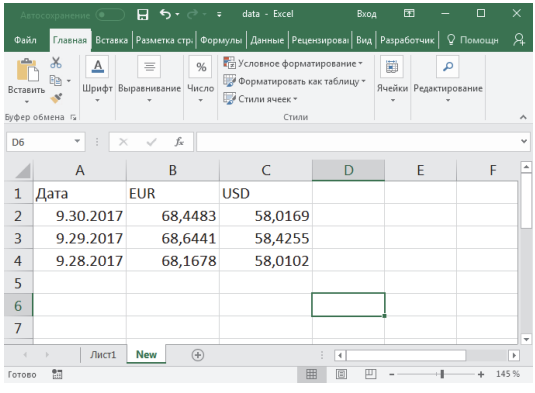


Рисунок 118

Правда, дату R вернул все-таки в европейской транскрипции.

Если мы хотим образовать новый файл, то параметр *append* должен быть равен FALSE или сокращенно F. Следующие две строчки создадут новый файл data2.xlsx с листом «New» по адресу «C:\\Users\\User\\Desktop\\data2.xlsx»:

```
PATH2 <- "C:\\Users\\User\\Desktop\\data2.xlsx"
write.xlsx2(Data2, PATH2, sheetName="New", col.
names=T, row.names=F, append=F, showNA=FALSE)
# Запись Data2 в новый файл
```

При записи командой write.xlsx2 нужно помнить, что записываемый в xlsx-файл объект R должен быть таблицей, т.е. типа data.frame. Часто внутренние данные в R могут быть списками, но формально не приведенными к фреймам. В этом случае необходимо перед записью дополнительно использовать команду:

```
Data2 <- as.data.frame(Data2)   # Объявить Data2 типом
  # data.frame
```

которая устанавливает требуемый тип объекта Data2.

Формат RDS

В заключение нельзя не сказать о замечательном собственном R-формате файлов, в которые можно сохранять любые R-объекты. Такие файлы имеют расширение .rds и создаются с помощью команды saveRDS:

```
saveRDS(file = "Путь и имя файла.rds", Data)
# Сохранение в файл формата .rds объекта Data
```

согласно которой по указанному пути и с указанным именем будет создан rds-файл, содержащий объект Data. Таким способом можно сохранять совершенно любые объекты R, начиная от чисел, массивов, таблиц и заканчивая функциями и списками.

Обратно считать данные таких файлов позволяет команда readRDS:

```
Data <- readRDS("Путь и имя файла.rds") # Считать дан-
  # ные из файла формата .rds в Data
```

результатом которой станет запись в переменную Data того объекта, который ранее был записан в rds-файл.

Задание 3. Сохранить в файл data.rds вектор $\vec{a}=(1,2,3)$ и текст «Пушкин – наше всё!», а затем обратно считать эти данные из созданного файла.

Решение. Объявим в R вектор $\vec{a}=(1,2,3)$ и текст «Пушкин – наше всё!»:

```
a <- 1:3; a          # Задаем вектор a с координатами (1,2,3)
t <- "Пушкин – наше всё!"; t      # Задаем текст в пере-
  # менной t
```

Для того, чтобы одновременно записать эти два объекта в файл создадим объединяющий их список L:

```
L <- list(Vector = a, Text = t)          # Задаем список, со-
  # держащий вектор a и текст t
```

После чего записываем этот список L в файл:

```
saveRDS(file = "data.rds", L)          # Сохранение в файл
  # data.rds  списка  L
```

Обратите внимание на то, что мы не указали путь к файлу, и поэтому он будет создан в рабочей директории текущего сеанса R. Проверьте это. Узнать и изменить рабочую директорию можно через меню Tools – Global Options – default working directory... (см. рис. 119).

Теперь считаем данные из только что созданного файла командой:

```
Data <- readRDS("data.rds")     # Считать данные из фай-
  # ла data.rds в Data
Data                          # Посмотреть что находится в Data
```

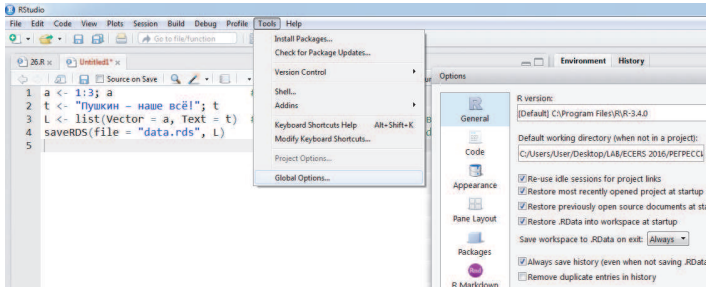


Рисунок 119

Но в таких случаях лучше воспользоваться командой glimpse из пакета dplyr, т.к. данные могут быть в общем случае громоздкими и не помещаться на экран (рис. 120):

```
glimpse(Data)          # Посмотреть структуру объекта Data
```

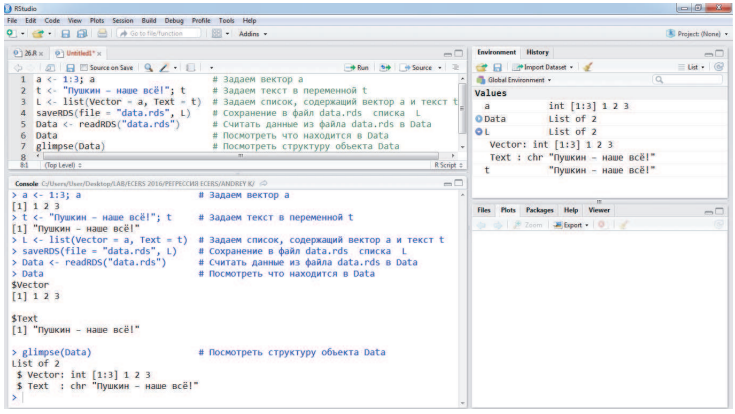


Рисунок 120

Как видим, считанный из файла data.rds объект Data содержит два поля: Vector и Text, т.е. можно непосредственно вернуть наш вектор командой Data$Vector и записанный текст командой Data$Text:

```
a <- Data$Vector; a          # Записать в а вектор
t <- Data$Text; t            # Записать в t текст
```

с отчетом в консоли

```
> a <- Data$Vector; a          # Записать в a вектор
[1] 1 2 3
> t <- Data$Text;t             # Записать в t текст
[1] «Пушкин – наше всё!»
```

Задания для самостоятельной работы

1. Дописать в файл data.xlsx (из разобранного выше задания 2) на новый лист «Cars» таблицу из зарезервированной в R переменной **cars** с данными о скоростях и длинах тормозных путей автомобилей Ford.

2. После загрузки библиотеки «ggplot2» в R становятся доступны данные под именем diamonds, в которых приведены статистические исследования алмазов. Создать xlsx-файл, содержащий данную таблицу diamonds. Указание: после того, как у вас ничего не получится, попробуйте все-таки использовать преобразование типов: *Data <- as.data.frame(diamonds)*

3. Скачать из интернета и сохранить в файле money.xlsx данные за последний полный месяц о курсе USD и EUR и записать их в файл money.rds не используя буфер обмена. Проверить результат, считав обратно данные из money.rds и отправив их в какой-нибудь новый xlsx-файл. Указание. В ряде случаев может потребоваться произвести автозамену точки на запятую в Excel, если данные о курсах валют содержат десятичным разделителем точку.

4. * Создать любым способом файлы matrix.xlsx и matrix.rds, содержащие в себе матрицу размера 100×100, элементы которой определяются формулой:

$$a_{ij} = \frac{2i - j}{i + j},$$

где i – номер строки, а j – номер столбца элемента матрицы.

Приложение Космос (для отличников)

Часто по итогу обработки Excel-данных вычислительными средствами R требуется записать результат не на новый лист, а в одну или несколько заранее определенных ячеек уже существующего листа, на котором, быть может, эти данные используются для последующих вычислений в других ячейках. Также часто необходимо считывать не весь лист, а определенную ячейку или их некоторую группу.

В этом разделе мы и обсудим возможности такого тонкого взаимодействия R с Excel, кстати, закладывающем основу полной автоматизации вычислений.

Задание А. Из файла data.xlsx с листа «Лист1» считать данные ячеек B1 и C4.

Решение. Напомним, что «Лист1» файла data.xlsx содержит фрагмент таблицы курса валют (рис. 121):

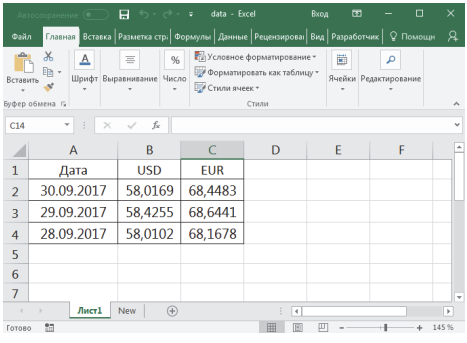


Рисунок 121

Причем, ячейка B1 имеет координаты (1, 2) и равна «USD», а ячейка C4 имеет координаты (4, 3) и равна 68,1678.

Изучите, но не копируйте и не запускайте в R, следующий ниже перечень команд, чтобы хоть немного

понять последующие за перечнем несколько строчек кода.

```
PATH <- "C:\\Users\\User\\Desktop\\data.xlsx"  # Путь
  # к файлу
Wbook <- loadWorkbook(PATH)        # Скачали книгу по
  # пути PATH в переменную Wbook
Sheet <- Wbook$getSheet("Лист1")  # Сохранить
  # "Лист1" книги Wbook в переменную Sheet
Row <- getRows(Sheet, 4)           # Сохранить 4-ую стро-
  # ку листа Sheet в переменную Row
Cell <- getCells(Row, 3)           # Сохранили 3-ю ячей-
  # ку строки Row в переменную Cell
X <- lapply(Cell, getCellValue, encoding="UTF-8")  # За-
  # писали содержимое ячейки Cell в переменную X
```

Итак, теперь скопируем и запустим в R небольшой, но весьма замысловатый, код, который решает задачу чтения содержимого ячеек B1 и C4:

```
PATH <- "C:\\Users\\User\\Desktop\\data.xlsx"  # Путь
  # к файлу
Wbook <- loadWorkbook(PATH)          # Скачали книгу
  # по пути PATH в переменную Wbook
Sheet <- Wbook$getSheet("Лист1")  # Сохранить
  # "Лист1" книги Wbook в переменную Sheet
# Сразу записываем значение ячейки (1, 2) на листе
  # Sheet в переменную B1:
B1 <- mapply(getCellValue, getCells(getRows(Sheet, 1), 2),
encoding="UTF-8"); B1 # Содержимое ячейки B1
# Сразу записываем значение ячейки (4, 3) на листе
  # Sheet в переменную C4:
C4 <- mapply(getCellValue, getCells(getRows(Sheet, 4), 3),
encoding="UTF-8"); C4 # Содержимое ячейки C4
```

С результатом в консоли:

```
> B1 <- mapply(getCellValue, getCells(getRows(Sheet,
1), 2), encoding="UTF-8"); B1 # Содержимое ячейки B1
```

```
  1.2
“USD”
> C4 <- mapply(getCellValue, getCells(getRows(Sheet,
4), 3), encoding=”UTF-8”); C4 # Содержимое ячейки
C4
    4.3
68.1678
```

Обратите внимание, что у считанных значений переменных B1 и C4 имеются атрибуты, равные координатам соответствующих ячеек: 1.2 и 4.3

Даже если не совсем понятно, как работает команда чтения, просто используйте конструкцию:

mapply(getCellValue, getCells(getRows(Sheet, i), j), encoding=”UTF-8”)

чтобы прочитать содержимое ячейки из строки *i* и столбца *j* на листе Sheet.

Полезно также знать, что группу рядом стоящих ячеек можно считать буквально одной строчкой. Например, чтобы считать всю таблицу значений валют (это 2–4 строки и 2–3 столбцы) в переменную Tabl, нужно запустить команду:

```
Tabl <- mapply(getCellValue, getCells(getRows(Sheet, 2:4),
2:3), encoding="UTF-8"); Tabl
```

Проделайте это и получите массив считанных ячеек:

```
> Tabl <- mapply(getCellValue,
getCells(getRows(Sheet, 2:4), 2:3),
encoding=”UTF-8”); Tabl
    2.2     2.3     3.2     3.3     4.2     4.3
58.0169 68.4483 58.4255 68.6441 58.0102 68.1678
```

А добавка строк кода

```
dim(Tabl) <- c(2, 3)   # Возвращаем размерность массиву:
  # 2 – количество столбцов, 3 – строк
Tabl <- t(Tabl); Tabl # Транспонируем в первоначальный вид
```

позволит увидеть в переменной Tabl массив считанных ячеек в исходном формате:

```
> dim(Tabl) <- c(2,3)
> Tabl <- t(Tabl); Tabl
         [,1]    [,2]
[1,] 58.0169 68.4483
[2,] 58.4255 68.6441
[3,] 58.0102 68.1678
```

Задание В. В файл data.xlsx на «Лист1» в ячейку D5 записать число π.

Решение. Ячейка D5 имеет координаты (5,4) и в представлении Java–объектов не существует, т.к. в файле она пустая во всех смыслах этого слова. Однако, создавать ее методами библиотеки «xlsx» достаточно трудоемко:

```
createRow(Sheet, i)                    # Создать i-ую строку
  # на листе Sheet
createCell(getRows(Sheet, i), j)       # Создать j-ую ячейку
  # в i-ой строке на листе Sheet
```

Причем, если строка или ячейка уже существовала с данными, то она будет затерта вновь создаваемой.

Гораздо проще заранее объявить требуемую ячейку на листе как существующую, например, просто раскрасив ее в желтый цвет (см. рис. 122).

Не забудем сохранить и обязательно закрыть файл data.xlsx перед записью в ячейку D5 числа :

```
PATH <- "C:\\Users\\User\\Desktop\\data.xlsx"  # Путь
  # к файлу
Wbook <- loadWorkbook(PATH)                    # Скачали
  # книгу в переменную Wbook
Sheet <- Wbook$getSheet("Лист1")               # Сохрани-
  # ли "Лист1" в переменную Sheet
Cell <- getCells(getRows(Sheet, 5), 4)         # Необходи
```

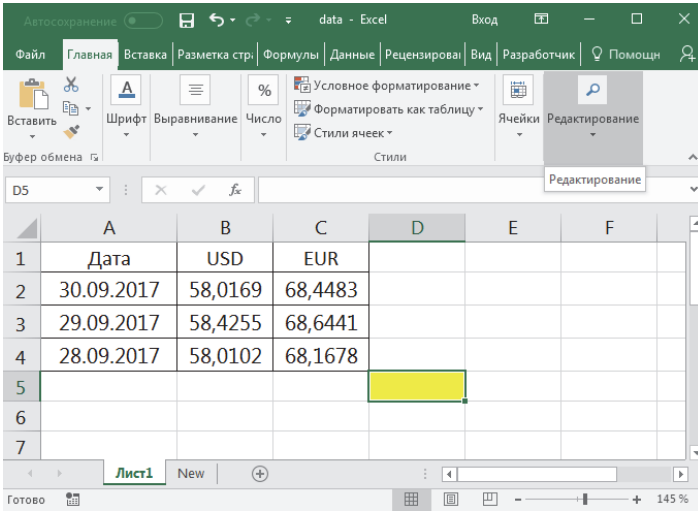


Рисунок 122

```
  # мая ячейка Cell (5,4) листа Sheet
mapply(setCellValue, Cell, pi)          # Записать
  # в ячейку Cell число pi
saveWorkbook(Wbook, PATH)               # Сохранить
  # измененную книгу
```

Не беспокойтесь о том, что функция записи в ячейку вернула NULL:

```
> mapply(setCellValue, Cell, pi)          # Записать
в ячейку D5 число 0.777
$`5.4`
NULL
```

Смело открывайте файл data.xlsx и проверяйте результат записи числа в ячейку D5 (см. рис. 123).

Замечание. Имейте в виду, что после каждого изменения с последующим сохранением Excel-файла требуется заново перезагружать книгу в R, т.е. выполнять команду:

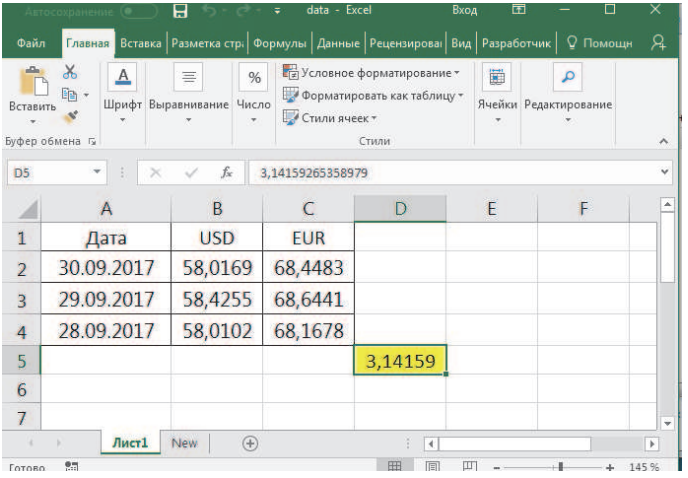


Рисунок 123

```
Wbook <- loadWorkbook(PATH)
# Скачали книгу в переменную Wbook
```

Следует резюмировать, что при записи объекта a (числа, текста) в ячейку (i, j) листа Sheet, можно ориентироваться на объединенную команду:

mapply(setCellValue, getCells(getRows(Sheet, i), j), a)

с последующим сохранением книги командой saveWorkbook(Wbook, PATH).

Практикум 12. Векторная алгебра (RStudio)

В этом разделе мы рассмотрим большинство операций, связанных с арифметическими векторами.

Задание векторов

Задание. Для векторов

$$\vec{a}=\begin{pmatrix}2\\-3\\4\\1\end{pmatrix},\vec{b}=\begin{pmatrix}-6\\9\\-12\\-3\end{pmatrix} \text{ и } \vec{p}=\begin{pmatrix}3\\2\\-1\\4\end{pmatrix}$$

вычислить следующие выражения:

a) $2\vec{a}-3\vec{b}+\vec{p}$ – линейная комбинация векторов

b) $\vec{a}\cdot\vec{b}$ и $\vec{a}\cdot\vec{p}$ – скалярное произведение векторов

c) $(\vec{a})^2=\vec{a}\cdot\vec{a}$ – квадрат вектора

d) $|\vec{a}|$, $|\vec{b}|$ и $|\vec{p}|$ – длины векторов

e) $\cos\left(\widehat{\vec{a},\vec{b}}\right)$ – косинус угла между векторами $\vec{a}$ и $\vec{b}$

f) $2\left(\vec{a},\vec{b}\right)\cdot\vec{p}+3\vec{b}\cdot(\vec{p})^2-|\vec{b}|\cdot\vec{b}$ – сложное выражение векторной алгебры

Решение. Подробно разберем каждую из перечисленных операций векторной алгебры, но сначала образуем в R указанные три вектора:

```
a <- c(2, -3, 4, 1)        # Сформировать вектор a из набора
  # чисел 2, -3, 4, 1
```

```
b <- c(-6, 9, -12, -3)   # Сформировать вектор b из набора
  # чисел -6, 9, -12, -3
p <- c(3, 2, -1, 4)      # Сформировать вектор a из набора
  # чисел 3, 2, -1, 4
a; b; p                  # Смотрим результат
```

Не забываем выделить этот фрагмент и кликнуть мышкой по кнопке меню Run или нажать сочетание клавиш Ctrl + Enter. Обязательно проверьте отчет компилятора в нижнем окне консоли на предмет ошибок ввода векторов:

```
> a; b; p
[1]  2 -3  4  1
[1]  -6   9 -12  -3
[1]  3  2 -1  4
```

Линейная комбинация векторов

а) вычислить $2\vec{a}-3\vec{b}+\vec{p}$

Для вычисления подобных выражений, называемых линейными комбинациями из-за того, что векторы складываются друг с другом и/или умножаются на числа, достаточно просто записать аналогичную строчку в R:

```
2*a-3*b+p              # Вычисление вектора, равного ука-
  # занной линейной комбинации
```

с результатом в консоли:

```
> 2*a-3*b+p      # Вычисление вектора, равного ука-
занной линейной комбинации векторов
[1]  25 -31  43  15
```

Напомним, что линейные операции над векторами осуществляются покоординатно, т.е. в нашем случае должен получиться вектор

$$(2a_1-3b_1+p_1;2a_2-3b_2+p_2;\ldots;2a_4-3b_4+p_4)$$

Попробуйте проверить этот результат вручную.

Скалярное произведение векторов

b) вычислить $\vec{a}\cdot\vec{b}$ и $\vec{a}\cdot\vec{p}$

В отличие от предыдущего пункта, здесь нельзя составить обычное умножение векторов, т.к. компилятор R покоординатно перемножит векторы, что не является скалярным произведением. Вообще, важно различать покоординатное произведение векторов (не используется в векторной алгебре, но удобно в программировании):

$$\left(a_1b_1;a_2b_2;\ldots;a_nb_n\right)$$

и скалярное произведение векторов (то что нам сейчас нужно):

$$\vec{a}\cdot\vec{b}=a_1b_1+a_2b_2+\ldots+a_nb_n\,.$$

Как видите, в результате получается не вектор, а число (скаляр) – отсюда и название скалярное произведение. Желая подчеркнуть эту разницу в линейной алгебре часто используют круглые скобки для обозначения скалярного произведения: $\vec{a}\cdot\vec{b}=\left(\vec{a};\vec{b}\right)$.

В языке R эти два приёма программируется следующим образом:

```
a*b              # Покоординатное произведение векто-
  # ров (не скалярное произведение!)
a%*%b            # Скалярное произведение векторов
```

с результатом:

```
> a*b            # Покоординатное произведение
векторов (не скалярное произведение!)
[1] -12 -27 -48  -3
> a%*%b          # Скалярное произведение векторов
     [,1]
[1,]  -90
```

Обратите внимание, что результат скалярного произведения представлен в виде матрицы 1×1, т.к. в языке R эта операция интерпретируется как частный случай произведения специальных матриц. Если мы

не хотим видеть результат в виде матрицы, мы можем вывести его как обычное число:

```
as.numeric(a%*%b)      # Скалярное произведение векторов
```

```
> as.numeric(a%*%b)
[1] -90
```

Аналогично получим скалярное произведение $\vec{a} \cdot \vec{p}$:

```
as.numeric(a%*%p)      # Скалярное произведение векторов
```

```
> as.numeric(a%*%p)
[1] 0
```

Кстати, в последнем случае результат оказался равным нулю, что говорит об ортогональности (перпендикулярности) векторов $\vec{a}$ и $\vec{p}$.

Впрочем, для нахождения скалярного произведения мы могли бы использовать простую конструкцию в виде суммы (sum) попарных произведений координат:

```
sum(a*p)      # Фактически тоже скалярное произведение
  # векторов
```

с тем же результатом.

c) вычислить $(\vec{a})^2 = \vec{a} \cdot \vec{a}$

Квадрат вектора $(\vec{a})^2$ понимается в векторной алгебре как скалярное произведение самого на себя $\vec{a} \cdot \vec{a}$, поэтому легко получить:

```
as.numeric(a%*%a)              # Квадрат вектора
```

```
> as.numeric(a%*%a)            # Квадрат вектора
[1] 30
```

Проверьте, что результат, действительно, равен сумме квадратов координат вектора:

$$(\vec{a})^2 = \vec{a} \cdot \vec{a} = a_1 a_1 + a_2 a_2 + \ldots + a_n a_n = a_1^2 + \ldots + a_n^2$$

Замечание. Ошибкой было бы записать a*a или a^2, т.к. в этом случае R выдал бы покоординатное выполнение указанных операций, а не их сумму: $(a_1^2, \ldots, a_n^2)$.

Длина вектора

d) вычислить $|\vec{a}|$, $|\vec{b}|$ и $|\vec{p}|$

Длиной вектора называют число, равное квадратному корню из $(\vec{a})^2$ и по смыслу это число является длиной отрезка, соединяющего начало и конец вектора, если его интерпретировать как направленный отрезок в евклидовой геометрии:

$$|\vec{a}| = \sqrt{a_1^2 + \ldots + a_n^2}$$

В языке R получить данное выражение можно несколькими способами. Первый из них – вызвать специальную функцию вычисления нормы элемента для вектора:

```
norm(a, type="2")   # Длина вектора a (обычная евклидова)
```

с результатом:

```
> norm(a, type=»2»)          # Длина вектора a (обычная евклидова)
[1] 5.477226
```

Второй способ – образовать, согласно формуле, корень из суммы квадратов элементов:

```
sqrt(sum(a^2))        # Альтернатива: длина вектора a
```

с тем же результатом:

```
> sqrt(sum(a^2))              # Альтернатива: длина вектора a
[1] 5.477226
```

Аналогично получаем для оставшихся векторов:

```
norm(b, type="2")    # Длина вектора b (обычная евклидова)
sqrt(sum(p^2))       # Альтернатива: длина вектора p
```

Косинус угла между векторами

e) вычислить $\cos\left(\widehat{\vec{a},\vec{b}}\right)$

Воспользуемся формулой для косинуса угла между арифметическими векторами

$$\cos\left(\widehat{\vec{a},\vec{b}}\right)=\frac{\vec{a}\cdot\vec{b}}{|\vec{a}|\cdot|\vec{b}|}$$

```
(a%*%b)/(norm(a, type="2")*norm(b, type="2"))
# Косинус угла между a и b
```

с результатом

```
> (a%*%b)/(norm(a,type="2")*norm(b,type="2"))
     [,1]
[1,]   -1
```

Обратим внимание, что косинус оказался равным -1, что говорит об угле между векторами, равном π. Это означает, что векторы коллинеарны (параллельны) и противоположно направлены. Кстати, это можно было заметить изначально, т.к. координаты векторов пропорциональны с отрицательным коэффициентом: $\vec{b}=-3\vec{a}$.

Конечно, чтобы формула не выглядела громоздкой, ее можно было запрограммировать поэтапно:

```
ab <- a%*%b                  # Образуем в переменной ab
  # скалярное произведение векторов a и b
L.a <- norm(a, type="2")   # Сохраняем длину вектора a
  # в переменной L.a
L.b <- norm(b, type="2")   # Сохраняем длину вектора b
  # в переменной L.b
ab/(L.a*L.b)                 # Получаем косинус угла
```

Произвольные выражения векторной алгебры

f) Вычислить $2\left(\vec{a},\vec{b}\right)\cdot\vec{p}+3\vec{b}\cdot\left(\vec{p}\right)^2-\left|\vec{b}\right|\cdot\vec{b}$

Вычислим отдельно три слагаемых выражения и образуем из них ответ:

```
S1 <- 2*as.numeric(a%*%b)*p      # Первое слагаемое
S2 <- 3*b*as.numeric(p%*%p)      # Второе слагаемое
```

```
S3 <- norm(b, type="2")*b          # Третье слагаемое
S1 + S2 - S3                       # Ответ
```

с результатом в виде вектора

```
> S1 + S2 - S3
[1] -981.4099  302.1149 -702.8199 -940.7050
```

Обратите внимание, что при вычислении скалярных произведений необходимо конвертировать ответ в число с помощью команды as.numeric(), иначе произойдет несовпадение типов (матрицы и векторы как бы перемешаются в одном выражении) и компилятор R выдаст ошибку.

Если вычисление скалярных произведений производится для многих пар векторов, то для этих целей удобно запрограммировать специальную пользовательскую функцию, скажем, под названием cosV:

```
cosV <- function(x,y) {          # Объявление имени функ-
  # ции cosV двух аргументов
  L.x <- norm(x, type="2"); L.y <- norm(y, type="2")  # На-
# хождение длин векторов
   cosV <- x%*%y/(L.x*L.y)                 # Нахождение
# косинуса угла
   return(as.numeric(cosV))    # Возвращение косинуса
# в качестве значения функции
}
cosV(a,b)                        # Вызов пользовательской
  # функции cosV
```

```
> cosV(a,b)   # Вызов пользовательской функции cosV
[1] -1
```

Задания для самостоятельной работы

1. Образовать и вывести на экран векторы $a(1,1,2)$ и $b(0,-4,3)$. Вычислить выражение

   a) $-4a+5b$

   b) $2(a,b)*a-3|b|*b$

2. Образовать и вывести на экран векторы $a(2,0,-3)$ и $b(6,1,4)$. Вычислить их длины и скалярное произведение. Ортогональны ли векторы?

3. Образовать и вывести на экран векторы $a(1,1,2,2,3,3,4,4)$ и $b(-2,-1,0,1,2,3,4,5)$. Какой вектор длиннее? Определить тупой, острый или прямой угол образован между векторами и найти его косинус.

4. Выбрать из матрицы A вектор-столбец максимальной длины, если матрица имеет вид:

$$A = \begin{pmatrix} 1 & 0 & 4 & -2 & 1 & 1 \\ 2 & 3 & 4 & 2 & 0 & -3 \\ 3 & 1 & 3 & 2 & 9 & 5 \\ 4 & 0 & 3 & 2 & 0 & -5 \\ 5 & 7 & 1 & -2 & 8 & 3 \\ 6 & 4 & 1 & 2 & 0 & 2 \end{pmatrix}.$$

5. Есть ли в матрице из предыдущего задания ортогональные столбцы? Если есть, то какие пары?

Ответы:

1. a) (-4,-24,7)

1. b) (4,64,-37)

2. |a|=3.605551; |b|=7.28011; (a,b)=0 – да, ортогональны

3. |a|=|b|=7.745967; (a,b)=50; cos(a,b)=0.8333333>0 – угол острый

4. 5-ый столбец

5. Да, есть ортогональные: 4-ый и 5-ый столбцы

Практикум 13. Алгебра матриц (RStudio)

В данном разделе мы будем рассматривать следующие темы алгебры матриц в R: линейные операции над матрицами (сложение матриц и умножение их на числа), умножение матриц и их транспонирование, вычисление определителей, определение ранга матриц и их размерности, нахождение обратных матриц, произвольные покоординатные вычисления.

Задание матриц

Объявим в R три матрицы:

$$A=\begin{pmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{pmatrix}, E=\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix} \text{ и } B=\begin{pmatrix}2 & -3\\ 2 & 0\\ 2 & 1\end{pmatrix},$$

вспомнив при этом основные конструкции и способы их задания:

```
# Заполнить матрицу 3x3 числами от 1 до 9 по строкам:
A <- matrix(1:9, nrow=3, ncol=3, byrow = TRUE)
A                    # Вывести на экран матрицу A
E <- diag(3)         # Записать в переменную E единичную
  # матрицу размера 3x3
E                    # Вывести на экран матрицу E
# Построить матрицу из двух столбцов: первый – по-
  # вторы числа 2 три раза, второй – вектор с координа-
  # тами (-3,0,1):
```

```
B <- cbind(rep(2, 3), c(-3, 0, 1))
B                    # Вывести на экран матрицу B
```

Выделим набранный скрипт и запустим компилятор R, указав мышью на клавишу Run или нажав Ctrl + Enter. В результате получим на экране консоли объявленные матрицы (см. рис. 124).

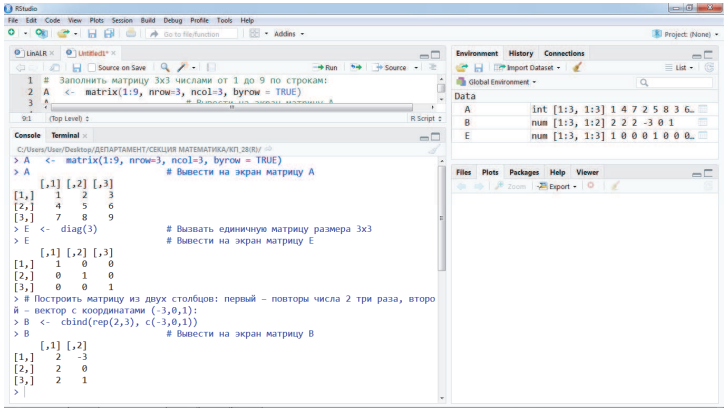


Рисунок 124

Убедитесь, что матрицы введены корректно.

Добавим к введенным матрицам A, E и B еще одну матрицу S:

$$S = \begin{pmatrix} 1 & -2 & 0 & 2 & 5 & 3 \\ -6 & 4 & 2 & -3 & 0 & 0 \\ 1 & 8 & -11 & 6 & 8 & 9 \end{pmatrix}.$$

Такую матрицу, как указывалось ранее, проще всего ввести в таблице Excel, затем скопировать ее в буфер обмена и запустить следующий код в R (см. также рис. 125):

```
# Чтение из буфера обмена excel-формата:
Data <- read.table("clipboard", h=FALSE, dec=",", sep =
"\t")
S <- data.matrix(Data)              # Объявить таблицу чи-
```

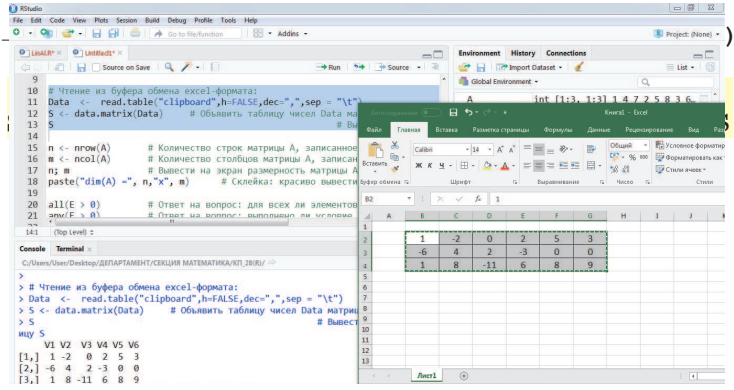


Рисунок 125

Можно заметить, что столбцы матрицы оказались проименованными, т.к. из буфера обмена данные были считаны в формате data.frame, что приводит к обязательному различию в названиях столбцов, пусть даже и формальному *V1*,*V2*,...

При этом обращение к отдельным элементам матриц, включая столбцы и строки, выглядит знакомым образом:

```
A[1, 2]          # Элемент a12 матрицы A (первая строка,
  # второй столбец)
B[2, ]           # Вторая строка матрицы B
S[ , 4]          # Четвертый столбец матрицы S (хотя и бу-
  # дет выведен на экран как строка!)
S[ , c(4,5)]     # Матрица, содержащая лишь 4-ый и 5-ый
  # столбцы матрицы S
S[ , -c(4,5)]    # Матрица S, из которой удалены столбцы
  # с номерами 4 и 5
```

Все приводимые далее примеры относятся к введенным матрицам *A*, *B*, *E* и *S*.

Размерность матрицы

Задание 1. Для матрицы A вывести ее размерность

Решение. Составим следующий код определения размерности матриц:

```
n <- nrow(A)          # Количество строк матрицы A, запи-
  # санное в переменную n
m <- ncol(A)          # Количество столбцов матрицы A,
  # записанное в переменную m
n; m                  # Вывести на экран размерность мат-
  # рицы A (n x m)
paste("dim(A) =", n, "x", m)
# Склейка: красиво вывести на экран размерность мат-
# рицы A
```

Обратите внимание на оператор paste, выводящий на экран консоли текстовую строку, состоящую из четырех склеенных кусков «dim(A) =», n, «x» и m.

Кванторы общности и существования

Можно было бы данный раздел назвать по-другому: проверка выполнения условий для всех элементов матрицы и проверка выполнения условий хотя бы для одного элемента матрицы.

Задание 2. Для матрицы *E* выяснить

a) Все ли её значения положительны?

b) Есть ли у неё положительные значения?

Решение. Для ответа на эти вопросы в общем случае нам понадобятся операторы: all(...) и any(...). Чтобы понять различия в этих логических операциях, составим проверку условия положительности для каждого элемента матрицы:

```
E > 0          # Проверяем каждый элемент матрицы E
  # на положительность
```

с результатом в окне консоли:

```
> E > 0                    # Проверяем каждый элемент
матрицы E на положительность
      [,1]  [,2]
[1,]  TRUE FALSE
[2,] FALSE  TRUE
```

Нетрудно видеть, что не все элементы матрицы E строго положительны, хотя имеются и такие. Так что, ответ на первый вопрос: FALSE, а на второй – TRUE.

Однако на практике матрица может быть настолько больших размеров, что вывести на экран результат для каждого её элемента не представляется возможным. Здесь как раз и будет уместно использовать специальные логические операторы all(...) и any(...). Посмотрите, как элегантно выглядят в коде ответы на наши два вопроса:

```
all(E > 0)      # Ответ на вопрос: для всех ли элементов
  # матрицы E выполнено условие?
any(E > 0)      # Ответ на вопрос: выполнено ли условие
  # хотя бы для одного элемента?
```

с тем же результатом:

```
> all(E > 0)    # Ответ на вопрос: для всех ли эле-
ментов матрицы E выполнено условие?
[1] FALSE
> any(E > 0)      # Ответ на вопрос: выполнено ли
условие хотя бы для одного элемента?
[1] TRUE
```

Транспонирование матриц

Задание 3. Найти транспонированную матрицу S^T.

Решение. Более простой по семантике операции в R не существует:

```
t(S)          # Транспонирование матрицы S
```

```
> t(S)                    # Транспонирование матрицы S
```

```
   [,1] [,2] [,3]
V1    1   -6    1
V2   -2    4    8
V3    0    2  -11
V4    2   -3    6
V5    5    0    8
V6    3    0    9
```

Сложение матриц и умножение их на числа

Задание 4. Найти линейную комбинацию матриц $W = 3A - 2E$.

Решение. Линейные операции над матрицами производятся покоординатно и выглядят абсолютно аналогично соответствующим операциям над векторами:

```
W <- 3*A-2*E     # Арифметические операции над мат-
  # рицами, сохраненные в матрице W
W                # Можно вывести на экран результат W
```

```
> W  <-  3*A-2*E     # Арифметические операции над
матрицами, сохраненные в матрице W
> W                  # Можно вывести на экран ре-
зультат W
     [,1] [,2] [,3]
[1,]    1    6    9
[2,]   12   13   18
[3,]   21   24   25
```

Здесь важно помнить, что размерности слагаемых должны быть одинаковыми. Так, например, невозможно сложить матрицы разных размерностей

$$A + B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} + \begin{pmatrix} 2 & -3 \\ 2 & 0 \\ 2 & 1 \end{pmatrix}:$$

```
A+B     # Попытка сложить матрицы разных размерностей
```

что немедленно приводит к сообщению об ошибке:

```
> A+B   # Попытка сложить матрицы разных размерностей
Error in A + B : non-conformable arrays
```

Произведение матриц

Задание 5. Найти матричные значения указанных выражений

a) $A \cdot B$
b) $S \cdot S^T$
c) A^2
d) A^6
e) $\sqrt{A}$

Решение. Матричное произведение в R отличается от покоординатного произведения и оформляется в специальных «процентных» скобках: %*%

```
A %*% B          # Произведение матриц A и B
S %*% t(S)       # Произведение матриц S и t(S)
A %*% A          # Матричный квадрат матрицы A²
```

с результатом в консоли

```
> A %*% B             # Произведение матриц A и B
     [,1] [,2]
[1,]   12    0
[2,]   30   -6
[3,]   48  -12
> S %*% t(S)          # Произведение матриц S и t(S)
     [,1] [,2] [,3]
[1,]   43  -20   64
[2,]  -20   65  -14
[3,]   64  -14  367
> A %*% A             # Матричный квадрат матрицы A²
     [,1] [,2] [,3]
[1,]   30   36   42
[2,]   66   81   96
[3,]  102  126  150
```

Кстати, в первом случае произведение матриц в обратном порядке $B \cdot A$ не существует из-за несоответствия размерностей. Напомним, что для матричного произведения необходимо, чтобы количество столбцов в первой матрице было равно количеству строк второй матрицы.

В третьем случае ошибкой было бы записать A^2, т.к. эту форму компилятор R воспринимает как покоординатное произведение элементов и представляет результат как A*A. Запустите следующий ниже код и посмотрите разницу между покоординатным квадратом A*A=A^2 и матричным возведением в степень A%*%A:

```
A*A             # Поэлементные квадраты чисел в матри-
  # це A. Можно и так: A^2
A%*%A           # Квадрат матрицы A (сравните с преды-
  # дущим результатом!)
```

Возведение в степень

Для вычисления старших степеней матрицы удобно использовать специальную библиотеку expm. Напомним, что при самом первом использовании какой-либо библиотеки на используемом компьютере необходимо выполнить первоначальную загрузку пакета из интернета командой install.packages(«Имя пакета») или использовать меню *Rstudio: Tools -> install packages*.

```
library(expm)       # Активация библиотеки expm
A%^%5               # Пятая степень матрицы A
sqrtm(A)            # Корень из матрицы A
```

```
> A%^%5                 # Пятая степень матрицы A
       [,1]   [,2]   [,3]
[1,] 121824 149688 177552
[2,] 275886 338985 402084
```

```
[3,] 429948 528282 626616
> sqrtm(A)              # Корень из матрицы A
                [,1]            [,2]             [,3]
    [1,] 0.4497564+0.7622786i 0.5526217+0.2067958i
0.6554871-0.3486869i
    [2,] 1.0185207+0.0841514i 1.2514702+0.0228291i
1.4844197-0.0384931i
    [3,] 1.5872851-0.5939759i 1.9503187-0.1611376i
2.3133523+0.2717007i
```

В последнем примере с матричным корнем sqrtm(A) результатом является одна из матриц, квадрат которой совпадает с матрицей A. Легко заметить, что полученный результат – комплекснозначная матрица.

Проверим результат вычисления одного из корней матрицы sqrtm(A):

```
sqrtm(A)%*%sqrtm(A)        # Возведем в квадрат матри-
  # цу sqrtm(A)
```

и получим исходную матрицу A:

```
> sqrtm(A)%*%sqrtm(A)
     [,1] [,2] [,3]
[1,] 1+0i 2+0i 3-0i
[2,] 4+0i 5-0i 6+0i
[3,] 7+0i 8+0i 9+0i
```

Определители матриц

Задание 6. Для матрицы A найти

a) Элемент a_{23}

b) M_{23} – минор элемента a_{23}

c) A_{23} – алгебраическое дополнение элемента a_{23}

d) Определитель $\det(A)$ матрицы A

Решение. Отработайте поочередно каждую строчку кода и проверьте совпадают ли ответы с теоретическими:

```
A              # Матрица A
```

```
> A                       # Матрица A
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

```
A[2, 3]                        # Элемент a23 матрицы A
```

```
> A[2, 3]                 # Элемент a23 матрицы A
[1] 6
```

```
A[-2,-3]       # Матрица A без 2-ой строки и 3-его столбца
```

```
> A[-2,-3]                # Матрица A без 2-ой строки
и 3-его столбца
     [,1] [,2]
[1,]    1    2
[2,]    7    8
```

```
det(A[-2,-3])           # Минор к элементу a23 матрицы A
```

```
> det(A[-2,-3])          # Минор к элементу a23 ма-
трицы A
[1] -6
```

```
(-1)^(2+3)*det(A[-2,-3])        # Алгебраическое дополне-
  # ние к элементу a23 матрицы A
```

```
> (-1)^(2+3)*det(A[-2,-3]) # Алгебраическое допол-
нение к элементу a23 матрицы A
[1] 6
```

```
det(A)              # Определитель A (в теории точный
  # ноль!)
```

```
> det(A)          # Определитель A (в теории точный
ноль!)
[1] -9.517127e-16
```

Здесь необходимо дать некоторые пояснения. Функция численного нахождения определителя матрицы det(A) реализует метод Гаусса – самый быстрый

из существующих для больших размерностей матриц. Однако, для целочисленных матриц результат может оказаться числом не целым.

Ниже мы приводим код пользовательской функции Det(A) – разложения определителя по первой строке, которая для целочисленых матриц вычисляет точное значение определителя:

```
Det <- function(A) {
 n <- nrow(A); m <- ncol(A)
 if (n != m) {return("Матрица не квадратная!")} else {
   if (n==1) {Rez <- A[1, 1]} else {
      Rez <- 0
      for (j in 1:n) {
         B <- as.matrix(A[-1, -j])
         Rez <- Rez + (-1)^(1+j)*A[1, j]*Det(B)
      }
    }
   return(Rez)
 }
}
# Пример использования
A <- matrix(1:9, 3, 3, byrow = TRUE)
A
Det(A)          # Разложение определителя по 1-ой строке
det(A)          # Метод Гаусса (встроенная в R функция)
```

Следует все же отметить, что для матриц больших размерностей (более 10x10) точный алгоритм Det(A) заметно уступает по скорости приближенному det(A).

Обратная матрица

Задание 7. Найти обратную матрицу к матрице

a) A

b) $A+E$

Решение. Обратная матрица в R вызывается базовой функцией solve(...), которая возвращает требуемый результат для любой невырожденной матрицы (det(A)≠ 0):

```
solve(A)              # Обратная матрица к A не существу-
  # ет, т.к. det(A)=0 (см. выше)
solve(A+E)            # Обратная матрица к A+E
```

с результатом:

```
> solve(A)            # Обратная матрица к A не су-
ществует, т.к. det(A)=0 (см. выше)
Error in solve.default(A) :
   system is computationally singular: reciprocal
condition number = 3.7011e-18
> solve(A+E)          # Обратная матрица к A+E
     [,1] [,2]          [,3]
[1,]   -6 -2.0  3.000000e+00
[2,]   -1  0.5  2.219904e-17
[3,]    5  1.0 -2.000000e+00
```

Проверьте, все ли корректно: матрично перемножьте (A+E) и solve(A+E), в результате чего должна получиться единичная матрица (или ее «машинный» аналог).

Ранг матрицы

Задание 8. Найти ранг матрицы

a) A

b) $A+E$

Решение. Для нахождения ранга матрицы потребуется специализированная библиотека Matrix, также требующая первоначальной установки:

```
library(Matrix)          # Активация библиотеки Matrix
rankMatrix(A)[1]         # Ранг матрицы A
rankMatrix(A+E)[1]       # Ранг матрицы A+E
```

```
> library(Matrix)              # Активация библиотеки
Matrix
```

```
> rankMatrix(A)[1]          # Ранг матрицы A
[1] 2
> rankMatrix(A+E)[1]        # Ранг матрицы A+E
[1] 3
```

Вместо заключения к практикуму 13

Напомним, что любой результат из R можно вернуть обратно в Excel в виде объекта X (X – число, вектор, матрица) следующей строкой кода:

```
# Скопировать в буфер обмена объект X в excel-формате:
write.table(X,"clipboard",quote=FALSE, col.names =
FALSE, row.names = FALSE, sep = "\t",dec=",")
# Не забыть выгрузить результат в Excel из буфера ко-
  # мандой Ctrl + v
```

Задания для самостоятельной работы

1. Для матриц

$$Q=\begin{pmatrix}-1 & 2 & 0\\ 3 & -1 & 0\\ 1 & 4 & 1\end{pmatrix}, E=\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix} \text{ и } P=\begin{pmatrix}0 & 0 & 0 & 0 & -3 & 4\\ 1 & 1 & 1 & 1 & 1 & 1\\ 2 & 2 & 2 & 2 & 2 & 2\end{pmatrix}$$

вычислить и записать результат в excel –файл

a) $\dim(P)$

b) $-4Q+5E$

c) $Q^T \cdot Q - E$

2. Вычислить

a) Q^3

b) $\det(Q)$ и $\det(Q^2)$. Как связаны эти числа между собой?

c) $\sqrt{Q-2E}$. Проверьте результат.

3. Найти обратную матрицу Q^{-1} и проверить результат.

4. Найти ранг матрицы P и сравнить с теоретическим значением.

5. * Написать функцию, вычисляющую след произвольной квадратной матрицы. Проверить ее работоспособность на матрице

$$G = \begin{pmatrix} 1 & 0 & 4 & -2 & 1 & 1 \\ 2 & 3 & 4 & 2 & 0 & -3 \\ 3 & 1 & 3 & 2 & 9 & 5 \\ 4 & 0 & 3 & 2 & 0 & -5 \\ 5 & 7 & 1 & -2 & 8 & 3 \\ 6 & 4 & 1 & 2 & 0 & 2 \end{pmatrix}.$$

Замечание. Следом квадратной матрицы A $(n \times n)$ называется число, обозначаемое Tr(A) и равное сумме ее диагональных элементов, т.е. $\mathrm{Tr}(A) = \sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + \ldots + a_{nn}$.

Практикум 14. Матричные уравнения (RStudio)

Посвятим этот раздел изучению вычислительных возможностей R при решении матричных уравнений и, в частности, систем линейных алгебраических уравнений.

Системы линейных алгебраических уравнений

Напомним, что любую систему линейных алгебраических уравнений можно представить в матричном виде. Например, система уравнений

$$\begin{cases} 2x-3y+z=-7 \\ x+4y-z=0 \\ -3x-y+z=12 \end{cases} \qquad (1)$$

эквивалента матричному уравнению

$$AX=B,$$

где матрицы A, B и X задаются следующим образом:

$$A=\begin{pmatrix} 2 & -3 & 1 \\ 1 & 4 & -1 \\ -3 & -1 & 1 \end{pmatrix}, \quad B=\begin{pmatrix} -7 \\ 0 \\ 12 \end{pmatrix} \text{ и } \quad X=\begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Не все подобные задачи удается решить численно, т.к. для существования и единственности решения системы типа (1) матрица A должна быть квадратной и невырожденной, т.е. количество неизвестных переменных (столбцов) должно равняться количеству урав-

нений (строк) и все эти уравнения должны быть линейно независимыми.

Теоретически, это означает, что определитель матрицы A должен быть отличен от нуля. Однако, на практике «машинный» ноль не всегда оказывается строгим нулем, что приводит к ряду ограничений даже для невырожденных матриц с близким к нулю определителем.

Задание 1. Решить приближенно систему (1)

$$\begin{cases} 2x-3y+z=-7 \\ x+4y-z=0 \\ -3x-y+z=12 \end{cases}$$

Решение. Введем соответствующие матрицы A и B на листе Excel и скопируем их через буфер обмена в R (см. рис. 126):

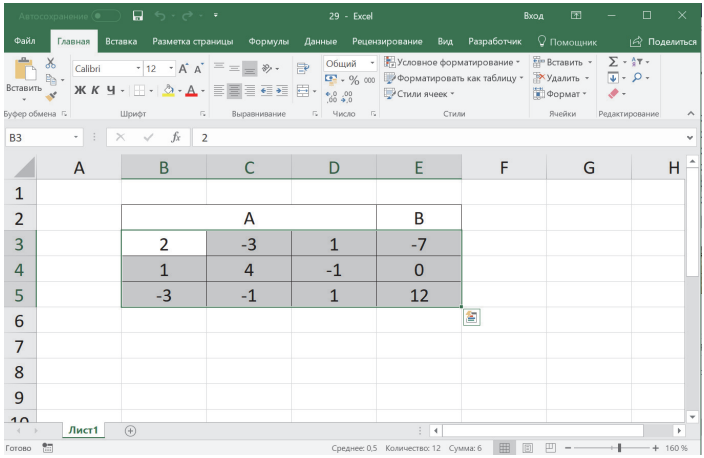


Рисунок 126

Здесь мы объединили на листе Excel две матрицы A и B в одну, чтобы их было проще скопировать в буфер обмена и считать в промежуточную матрицу W.

Далее считываем буфер обмена в R и образуем матрицы A и B (см. также рис. 127):

```
# В excel сформировать таблицу чисел и скопировать
  # в буфер обмена
Data <- read.table("clipboard", h=FALSE, dec=",", sep =
"\t")   # Чтение из буфера обмена excel-формата
W <- data.matrix(Data)          # Объявить таблицу чисел
  # Data матрицей W в R
#-------------------------------------------------------------------------
A <- W[ , 1:3]                  # Задать матрицу A как
  # первые 3 столбца таблицы W
B <- W[ , 4]                    # Задать матрицу B как 4-ой
  # столбец таблицы W
A                               # Посмотреть матрицу A
B                               # Посмотреть матрицу B
```

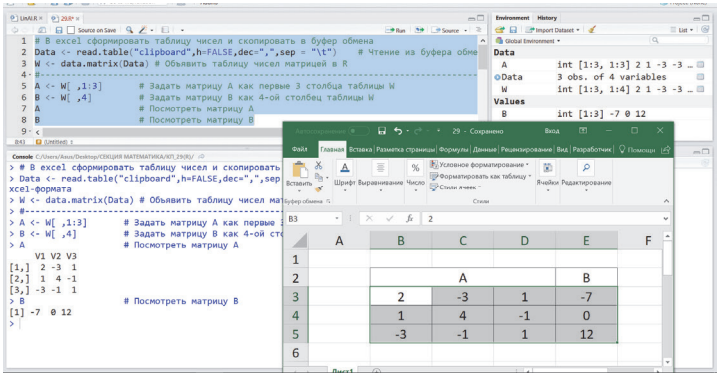


Рисунок 127

Теперь нам остается запустить специальную команду численного решения матричных уравнений $AX=B$. Выглядит этот код совсем несложно:

```
X <- solve(A, B)          # Найти приближенное решение
  # системы уравнений AX=B
X                         # Посмотреть решение X
```

с результатом:

```
> X  <-  solve(A,B)   # Найти приближенное решение системы уравнений AX=B
> X                   # Посмотреть решение X
V1 V2 V3
-3  2  5
```

Несмотря на то, что результат оказался совершенно точным, процедура solve не претендует на точное решение. Более того, для вырожденных матриц A (или близких к вырожденным) процедура выдает ошибку, связанную с обращением в бесконечность внутренних вычислений из-за близкого или равного нулю определителя матрицы A.

Матричные уравнения

Задание 2. Для матрицы A из предыдущего примера

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 1 & 4 & -1 \\ -3 & -1 & 1 \end{pmatrix}$$

решить приближенно матричное уравнение $AX = E$.

Решение. Легко заметить, что матричное уравнение определяет обратную матрицу $X = A^{-1}$. Получим ее обсуждаемым способом:

```
E <- diag(3)          # Вводим единичную матрицу E (3x3)
X <- solve(A, E)      # Решаем уравнение AX= E
X                     # Выводим результат в консоль
```

с результатом:

```
> E <- diag(3)     # Вводим единичную матрицу (3x3)
> X <- solve(A, E)          # Решаем уравнение AX=E
> X                    # Выводим результат в консоль
         [,1]       [,2]        [,3]
V1 0.2727273 0.1818182 -0.09090909
```

```
V2 0.1818182 0.4545455  0.27272727
V3 1.0000000 1.0000000  1.00000000
```

Замечание. В предыдущем параграфе мы находили обратную матрицу с помощью той же команды solve, но без указания второго аргумента (без единичной матрицы E). Убедитесь, что данная команда

```
X <- solve(A)        # Находим обратную матрицу к A
X                    #  Смотрим результат
```

действительно, приводит к тому же результату.

Задание 3. Для матрицы Q

$$Q = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

найти обратную Q^{-1}.

Решение. Точное значение определителя матрицы $\det Q = 0$, следовательно, матрица вырождена и её обратная матрица не определена (не существует). Кстати, «машинный» ноль определителя здесь вовсе не ноль, хотя и очень маленький, порядка $6{,}66 \cdot 10^{-16}$:

```
det(Q)        # Определитель матрицы Q
```

```
 > det(Q)    # Определитель матрицы Q
[1] 6.661338e-16
```

Посмотрите, как отреагирует процедура solve на нашу задачу:

```
# Задаем и отображаем матрицу:
Q <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE); Q
solve(Q)                    # Находим обратную матрицу к Q
```

В результате компилятор выдаёт вполне ожидаемую ошибку:

```
> Q <- matrix(1:9, nrow = 3, ncol = 3, byrow =
TRUE); Q
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> solve(Q)
Error in solve.default(Q) :
  system is computationally singular: reciprocal
condition number = 2.59052e-18
```

Замечание. В версии 3.4.2 и выше в данном сообщении будет указано на точную вырожденность матрицы Q.

Замечание для отличников. Если мы не хотим лишних сообщений об ошибках или хотим их конкретизировать, то можно перехватить программный текст Error и исправить на требуемый:

```
MySolve <- function(A){
  tryCatch(solve(A), error = function(x) print("Матрица
необратима!"))
}

# Пример использования функции MySolve:
MySolve(Q)      # Находим обратную матрицу или полу-
  # чаем сообщение о вырожденности
MySolve(A)      # Находим обратную матрицу или полу-
  # чаем сообщение о вырожденности
```

Задание 4. Рассмотрим в качестве следующего примера задачу о нахождении матрицы X, удовлетворяющую уравнению

$$A^{-1}XB^2 = E,$$

где $A = \begin{pmatrix} -1 & 2 & -3 \\ 2 & 9 & -6 \\ -3 & -6 & 0 \end{pmatrix}, B = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 0 \\ -3 & 4 & -5 \end{pmatrix}$, а E – единичная (3x3).

Решение. Для нахождения матрицы X необходимо обе части уравнения слева домножить на матрицу A:

$$A\cdot(A^{-1}XB^2) = A\cdot E \Leftrightarrow (AA^{-1})XB^2 = A \Leftrightarrow EXB^2 = A \Leftrightarrow XB^2 = A$$

и полученное уравнение справа домножить на квадрат обратной матрицы к B:

$$\left(XB^2\right)B^{-2} = AB^{-2} \Leftrightarrow X\left(B^2B^{-2}\right) = AB^{-2} \Leftrightarrow XE = AB^{-2} \Leftrightarrow X = AB^{-2}$$

В итоге нам необходимо реализовать вычисление для матрицы

$$X = AB^{-2} = A(B^{-1})^2 .$$

Введем данные в R из Excel аналогично через буфер обмена (см. рис. 128):

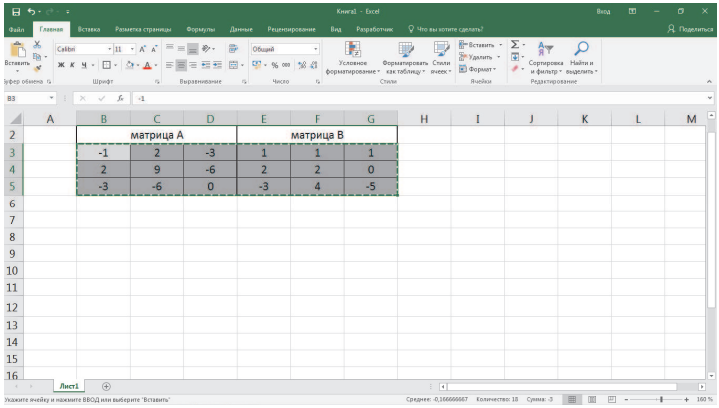


Рисунок 128

Здесь мы также временно объединили две матрицы в одну W и собираемся в R левую половину объявить матрицей A, а правую – матрицей B.

```
# В excel сформировать таблицу чисел и скопировать
  # в буфер обмена
Data <- read.table("clipboard", h=FALSE, dec=",", sep =
"\t")   # Чтение из буфера обмена
W <- as.matrix.data.frame(Data)  # Объявить таблицу
  # чисел матрицей в R
A <- W[ , 1:3]                   # Объявить матрицей A
  # первые три столбца матрицы W
B <- W[ , 4:6]                   # Объявить матрицей B
  # последние три столбца матрицы W
```

Далее нам остается вызвать соответствующее матричное произведение

$$X = A \cdot (B^{-1})^2$$

```
X <- A%*%(solve(B)%*%solve(B)); X # Вычислить ответ A*((B^-1)*(B^-1))
library(expm)                              # Подключаем
  # библиотеку для степеней матриц %^%
X <- A%*%solve(B%^%2); X                   # То же самое, но
  # короче... A*((B^-1)^2)
```

Если требуется выгрузить ответ X обратно в Excel, то необходимо добавить:

```
# Скопировать в буфер обмена матрицу X в excel-формате:
write.table(X,"clipboard", quote=FALSE, col.names =
FALSE, row.names = FALSE,
                    sep = "\t",dec=",")
```

(и не забудьте выгрузить ответ X обратно в Excel по сочетанию Ctrl + v) (см. рис. 129)

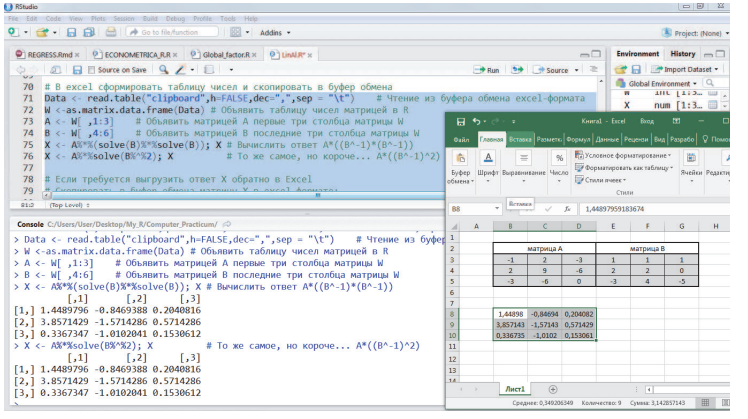


Рисунок 129

Обратите внимание на то, что оператор solve(B) возвращает обратную матрицу, а оператор solve(B%^%2) –

квадрат обратной матрицы. Вообще, используя определение обратной матрицы, несложно доказать, что квадрат обратной матрицы совпадает с обратной для квадрата матрицы:

$$\left(B^{-1}\right)^2 = \left(B^2\right)^{-1},$$

именно поэтому предыдущие выражения в алгебре матриц обозначают просто как B^{-2}.

Системы нелинейных алгебраических уравнений*

В заключение кратко остановимся на нелинейном случае. Рассмотрим специальную процедуру multiroot в пакете rootSolve (не забудьте скачать из репозитория), которая решает системы нелинейных уравнений

$$\begin{cases} f_1(x,y,z,\ldots) = 0 \\ \quad\vdots \\ f_m(x,y,z,\ldots) = 0 \end{cases} \qquad (1)$$

методом Ньютона-Рафсона.

Идея метода заключается в том, что нелинейные функции $f_i(x,y,z,\ldots)$ системы (1) раскладывают по формуле Тейлора до линейных слагаемых (до первого порядка) и находят решение уже линейной системы уравнений относительно приращений начальных аргументов $\Delta x, \Delta y, \Delta z, \ldots$, взяв за первоначальное приближение какой-либо вектор $(x_o, y_o, z_o, \ldots)$.

Естественно, найденное решение может оказаться далеко от истинного. Далее подставляют найденные приращения в исходное разложение и находят следующее уточнение корней. Повторяют такие итерации до тех пор, пока не достигнут заданной точности, а в случае превышения порога максимального количества итераций – останавливают процесс (обычно ограничиваются не более 100 итерациями).

Успехом применения данного метода служит удачный выбор начального приближения решения системы.

Задание 5. Найти точки пересечения прямой $x+y-2=0$ и окружности $(x-2)^2+y^2=4$.

Решение. В первую очередь для уверенного выбора начального приближения построим графики прямой и окружности:

```
x.line <- seq(-1, 5, length.out = 50)  # Для прямой разби-
  # ваем отрезок [1, 5] на 50 точек x.line
y.line <- 2-x.line                     # Вычисляем соответ-
  # ствующие орлинаты прямой y.line
t <- seq(0, 2*pi, length.out = 50)     # Задаем параметр-
  # угол t, описывающий окружность
x.circle <- 2 + 2*cos(t)               # Вычисляем абсцис-
  # сы окружности x.circle
y.circle <- 2*sin(t)                   # Вычисляем ордина-
  # ты окружности y.circle
plot(x.line, y.line, type = "l", lwd = 2, col = "blue")  # Стро-
  # им прямую
abline(h = 0, v = 0, lty = "84")                 # Отображаем
  # оси координат
lines(x.circle, y.circle, type = "l", lwd = 2, col = "red")
# Достраиваем окружность
```

Здесь для простоты мы воспользовались параметрическим уравнением окружности:

$$(x-x_o)^2+(y-y_o)^2=R^2 \quad \Leftrightarrow \quad \begin{cases} x=x_o+R\cos t \\ y=y_o+R\sin t \end{cases}, \ t\in[0,2\pi]$$

Возможно, вы видите у себя на экранах непропорциональный рисунок, в котором окружность деформирована. Это зависит от того, насколько раскрыто левое нижнее окно с графиком по отношению к другим окнам, т.к. в RStudio используется автоматическое масштабирование рисунков.

Из графиков на полученном рисунке 130 хорошо видны точки пересечения кривых. Левый корень системы имеет координаты примерно $A(0.5,1.5)$, а правый – $B(3.5,-1.5)$:

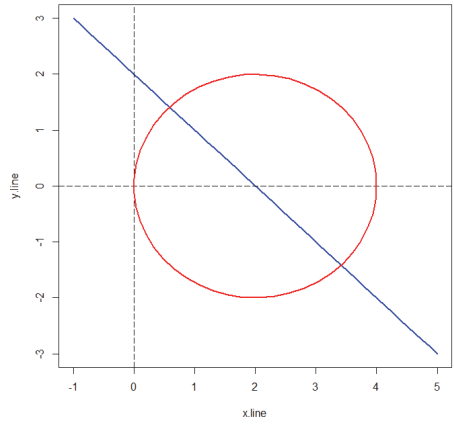


Рисунок 130

```
text(0.5, 1.5, "A")
text(3.5, -1.5, "B")
```

Найдем их уточненные значения. Объявим нашу систему уравнений

$$\begin{cases} line: & x+y-2=0 \\ circle: & (x-2)^2+y^2-4=0 \end{cases}$$

в виде специальной функции f, где вместо переменной x будем использовать $x[1]$, а вместо переменной y соответственно $x[2]$:

```
f <- function(x) { c(Line = x[1] + x[2] - 2, Circle = (x[1] - 2)^2 +
x[2]^2 - 4) }
```

Замечание. По сути, наша функция возвращает вектор, состоящий из двух чисел: насколько отличаются левые части для прямой и для окружности от нулей.

Далее активируем библиотеку rootSolve:

```
library(rootSolve)          # Активация пакета "rootSolve"
```

и запустим поиск корней системы с выбранными начальными приближениями.

В окрестности точки A:

```
multiroot(f, start = c(0.5, 1.5), ctol = 1e-8)    # Метод
  # Ньютона-Рафсона
```

```
> multiroot(f, start = c(0.5, 1.5), ctol = 1e-8)
# Метод Ньютона-Рафсона с начальным приближением
(0.5, 1.5)
$root
[1] 0.5857864 1.4142136     # Это найденное решение
системы (левая точка пересечения)

$f.root
          Line         Circle
-9.392487e-14  9.046097e-12 # Насколько отличаются
от нулей правые части уравнений

$iter                        # Количество проделан-
ных итераций
[1] 4

$estim.precis
[1] 4.570011e-12             # Достигнутая точность
```

В окрестности точки B:

```
multiroot(f, start = c(3.5, -1.5), ctol = 1e-8)  # Метод Нью-
  # тона-Рафсона
```

```
> multiroot(f, start = c(3.5, -1.5), ctol = 1e-8)
# Метод Ньютона-Рафсона с начальным приближением
(3.5, -1.5)
$root
[1]  3.414214 -1.414214     # Это найденное решение
системы (правая точка пересечения)

$f.root
          Line         Circle
```

```
-1.976197e-14  8.952838e-12 # Насколько отличаются
от нулей правые части уравнений

$iter
[1] 4            # Количество проделанных итераций

$estim.precis
[1] 4.4863e-12              # Достигнутая точность
```

Задания для самостоятельной работы

1. Решить систему уравнений

$$\begin{cases} -4x+z=11 \\ 3x+5y-2z=4 \\ 6x-3y+7z=-8 \end{cases}$$

и проверить результат.

2. Для матриц A и B

$$A=\begin{pmatrix} 1 & 1 & 8 \\ 8 & 0 & -2 \\ 3 & 3 & 5 \end{pmatrix} \quad \text{и} \quad B=\begin{pmatrix} 2 & 3 & 8 \\ 1 & 3 & 1 \\ 1 & -7 & 0 \end{pmatrix}$$

численно решить матричное уравнение и проверить результат

a) $AX=B$

b) $XA=B$

3. Для матриц A и B из предыдущего примера и единичной матрицы E (3x3) решить матричное уравнение

$$(A+E)^{-1}\cdot X\cdot B^3=(B-E)\cdot A^{-1}$$

и проверить результат.

4. Найдите точки возможного экстремума с точностью 10^{-7} для функции

$$f(x,y)=xy^2+\ln|x|+5\operatorname{arctg}y\ .$$

Проверьте результат, решив получающуюся систему аналитически.

Указание. Образуйте систему уравнений, содержащую обнуленные частные производные функции, и решите ее.

Ответ: $(-0.25, 2)$ *и* $(-4, 0.5)$

Практикум 15. Расширение double-арифметики

В данной части мы познакомимся с вычислениями в семантике обыкновенных дробей, которые позволяют сохранить точность результатов при оперировании с целыми или рациональными числами.

Алгебраически точное решение матричных уравнений*

В языке R разработана специальная библиотека *gmp*, которая позволяет оперировать не с десятичным форматом чисел, а с алгебраическим.

Поясним смысл сказанного, предварительно загрузив библиотеку gmp через меню Tools/Install Packages... и активировав ее кодом:

```
library(gmp)                # Библиотека расширенных типов
  # целых и рациональных чисел
```

Теперь попробуем продемонстрировать разницу между обычной double арифметикой *as.numeric* и расширенной с помощью специальных типов *as.bigz* и *as.bigq*.

Задание 1. Вычислить с максимальной точностью число 2^{200}.

Решение. Подключим максимальную выводимую на экран точность double арифметики:

```
options(digits = 22)       # Выводим на экран консоли мак-
  # симальное число разрядов
```

и вычислим число 2^{200}:

```
2^200
```

с результатом, записанным с помощью digits = 22 символов:

```
> options(digits = 22)
> 2^200
[1] 1.6069380442589903e+60
```

Это означает, что число 2^{200} мы представляем себе примерно, как:

$$2^{200} \approx 1{,}6069380442589903 \cdot 10^{60} =$$

$$= \overbrace{\underbrace{16069380442589903}_{17_{цифр}}000\ldots000}^{61_{цифр}} -$$

число из 61 цифры, первые 17 из которых мы видим, как значащие.

А остальные 44 цифры, вместо которых стоят нули, чему равны? Ответить на это вопрос здесь невозможно, т.к. мы использовали обычную double арифметику, и памяти, выделяемой на такие числа, недостаточно для получения более точного ответа.

Попробуем теперь воспользоваться специальным типом данных bigz, который выделяет под такие вычисления практически неограниченную память в рамках физически доступной на каждом конкретном компьютере (bigz – дословно «большие целые»):

```
as.bigz(2^200)       # Вычислить выражение со всеми
  # значащими цифрами
as.bigz(2)^200       # То же самое
```

с абсолютно точным результатом:

```
> as.bigz(2^200)     # Вычислить выражение со всеми
значащими цифрами
```

```
Big Integer ('bigz') :
[1] 160693804425899027554196209234116260252220299
3782792835301376
```

в котором хорошо видны все 61 значащие цифры числа!

Далее прокомментируем идею абсолютной точности рациональных чисел в алгебраической форме, т.е. чисел, которые можно представить в виде несократимой дроби $\frac{m}{n}$.

Потеря точности при вычислении таких дробей связана с возникновением периодических десятичных дробей, которые неизбежно приходится где-то обрывать. Однако, если результаты, включая промежуточные, представлять в виде обыкновенных дробей, то точность будет сохраняться – эту идею как раз и использует в своих алгоритмах специальный тип данных bigq (bigq – дословно «большие рациональные»).

Задание 2. Для матрицы A

$$A = \begin{pmatrix} 2 & -3 & 1 \\ 1 & 4 & -1 \\ -3 & -1 & 1 \end{pmatrix}$$

найти обратную матрицу с максимальной точностью.

Решение. Считаем матрицу из Excel (рис. 131):

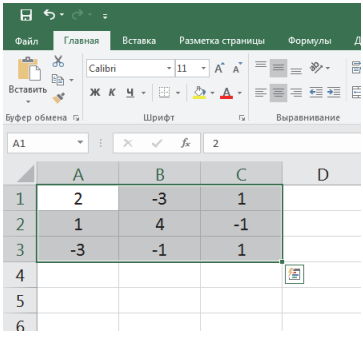


Рисунок 131

```
# В excel сформировать таблицу чисел и скопировать
  # в буфер обмена
Data <- read.table("clipboard", h=FALSE, dec=",", sep =
"\t")   # Чтение из буфера обмена excel-формата
A <- data.matrix(Data)     # Объявить таблицу чисел мат-
  # рицей в R
A                          # Посмотреть результат
```

и для сравнения укажем, что обычная максимальная точность double-арифметики дала бы такой ответ для обратной матрицы :

```
solve(A)        # Найти обратную матрицу в десятичном
  # представлении
```

с результатом:

```
> solve(A)       # Найти обратную матрицу в деся-
тичном представлении
                          [,1]                    [,2]
[,3]
V1     0.27272727272727276     0.18181818181818188
-0.09090909090909087
V2     0.18181818181818188     0.45454545454545464
0.27272727272727282
V3     1.0000000000000022      1.0000000000000022
1.0000000000000022
```

В случае представления результата через обыкновенные дроби типа bigq имеем:

```
solve(as.bigq(A))    # Найти обратную матрицу в обыкно-
  # венных дробях
solve.bigq(A)        # То же самое
```

с результатом:

```
> solve(as.bigq(A))       # Найти обратную матрицу
в обыкновенных дробях
Big Rational ('bigq') 3 x 3 matrix:
     [,1] [,2] [,3]
[1,] 3/11 2/11 -1/11
```

```
[2,] 2/11 5/11 3/11
[3,] 1    1    1
```

Проверим, насколько точной получится единичная матрица E, если перемножить AA^{-1}:

```
A%*%solve(as.bigq(A))  # Проверка точности
```

с абсолютно точным результатом:

```
> A%*%solve(as.bigq(A))  # Проверка точности
Big Rational ('bigq') 3 x 3 matrix:
     [,1] [,2] [,3]
[1,] 1    0    0
[2,] 0    1    0
[3,] 0    0    1
```

Задание 3. Решить систему уравнений

$$\begin{cases} 2x-3y+z=1 \\ x+4y-z=2 \\ -3x-y+z=3 \end{cases}$$

в обыкновенных дробях.

Решение. Матрица коэффициентов системы совпадает с матрицей A из предыдущего примера

$$A=\begin{pmatrix} 2 & -3 & 1 \\ 1 & 4 & -1 \\ -3 & -1 & 1 \end{pmatrix},$$

а столбец правой части $B=\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$.

Для сравнения приводим два решения: приближенное и точное (см. также рис. 132):

```
B <- 1:3                  # Задаем вектор B(1, 2, 3)
solve(A, B)               # Находим решение системы
  # AX=B в десятичном представлении
solve.bigq(A, B)          # Находим решение системы
  # AX=B в обыкновенных дробях
```

```
A%*%solve.bigq(A, B)      # Проверяем второй результат
  # (должен получиться столбец B)
```

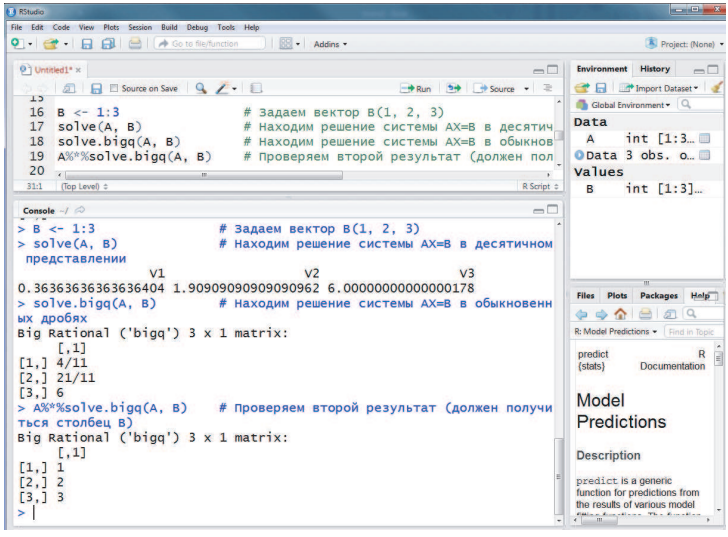


Рисунок 132

Разложение векторов по базису

Теперь мы готовы найти точное разложение векторов по базису для случая, если координаты всех векторов заданы в виде целых чисел или посредством обыкновенных дробей (рациональные числа).

Задание **4.** Разложить по базису

$$\begin{cases} \vec{e}_1 = (4;-7;8) \\ \vec{e}_2 = (8;0;21) \\ \vec{e}_3 = (11,-1,0) \end{cases}$$

a) вектор $\vec{a}(5;-16;150)$
b) вектор $\vec{b}(6;-3.5;14.5)$

Решение. Для того, чтобы найти разложение вектора $\vec{a}$ по базису $\vec{e}_1$, $\vec{e}_2$, $\vec{e}_3$, необходимо найти коэффициенты x_1, x_2 и x_3 в уравнении:

$$\vec{a} = x_1\vec{e}_1 + x_2\vec{e}_2 + x_3\vec{e}_3.$$

Если теперь вместо каждого вектора подставить столбец его координат, то мы получим алгебраическую систему уравнений с матрицами, составленными из столбцов координат соответствующих векторов. А именно:

$$A = \begin{pmatrix} 4 & 8 & 11 \\ -7 & 0 & -1 \\ 8 & 21 & 0 \end{pmatrix},\ B = \begin{pmatrix} 5 \\ -16 \\ 150 \end{pmatrix} \text{ и } X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}.$$

Поступая аналогично предыдущим вычислениям, образуем в Excel расширенную матрицу W и считываем ее в матрицы A и B:

```
# Работа с буфером обмена
Data <- read.table("clipboard", h=FALSE, dec=",", sep = "\t")
# Чтение из буфера обмена excel-формата
W <- data.matrix(Data)      # Объявить таблицу чисел
  # матрицей в R
A <- W[ , 1:3]; A       # Считываем матрицу A системы
B <- W[ , 4]; B         # Считываем правый столбец системы
```

с результатом, представленным на рисунке 133.

Далее остается запустить операцию решения системы:

```
solve(A, B)          # Приближенно решаем систему AX=B
solve.bigq(A, B)     # Точно решаем систему AX=B
```

с результатом для обоих случаев:

```
> solve(A, B)          # Приближенно решаем систему
AX=B
                V1                  V2                  V3
     3.0000000000000000  6.0000000000000000
-4.9999999999999991
```

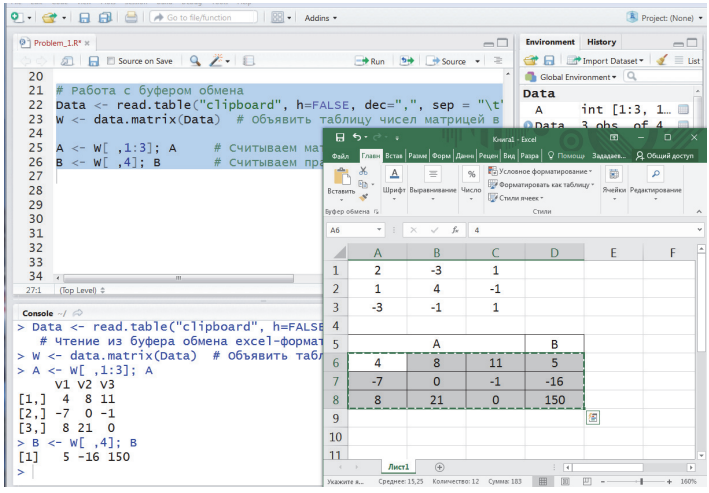


Рисунок 133

```
> solve.bigq(A, B)      # Точно решаем систему AX=B
Big Rational ('bigq') 3 x 1 matrix:
     [,1]
[1,] 3
[2,] 6
[3,] -5
```

Таким образом, искомое разложение по базису принимает вид:

$$\vec{a} = 3\vec{e}_1 + 6\vec{e}_2 - 5\vec{e}_3$$

Аналогично, для вектора $\vec{b}(6, -3.5, 14.5)$ получим:

```
B <- c(6, -3.5, 14.5); B
solve(A, B)          # Приближенно решаем систему AX=B
solve.bigq(A, B)     # Точно решаем систему AX=B
```

```
> B <- c(6, -7/2, 29/2); B
[1]  6.0 -3.5 14.5
> solve(A, B)          # Приближенно решаем систему
AX=B
                    V1                    V2
 5.0000000000000000e-01  5.0000000000000000e-01
                    V3
```

```
-2.0438670584833814e-17

> solve.bigq(A, B)     # Точно решаем систему AX=B

Big Rational ('bigq') 3 x 1 matrix:
     [,1]
[1,] 1/2
[2,] 1/2
[3,] 0
```

с ответом:

$$\vec{b} = \frac{1}{2}\vec{e}_1 + \frac{1}{2}\vec{e}_2 + 0\vec{e}_3 = \frac{1}{2}\vec{e}_1 + \frac{1}{2}\vec{e}_2$$

Заключение к практикуму 15

Заметим, что если в координатах вектора указать обыкновенную дробь, например, для вектора $\vec{b}\left(\frac{1}{3}, \frac{2}{7}, \frac{4}{5}\right)$ запустив строку кода:

```
b <- c(1/3, 2/7, 4/5)
b
```

```
> b <- c(1/3, 2/7, 4/5);
> b
[1] 0.3333333 0.2857143 0.8000000
```

то последующее решение solve.bigq(A, b) уже не будет точным, т.к. дроби $1/3$ и $2/7$ успеют потерять точность после округления в десятичном представлении double-арифметики.

Для сохранения точности необходимо такой вектор $\vec{b}$ изначально заявлять как bigq:

```
b <- c(as.bigq(1, 3), as.bigq(2, 7), as.bigq(4, 5))
b
```

```
> b <- c(as.bigq(1,3), as.bigq(2,7), as.bigq(4,5))
> b
```

```
Big Rational ('bigq') object of length 3:
[1] 1/3 2/7 4/5
```

Если же координаты вектора заданы как иррациональные числа, например, $\vec{b}\left(\sqrt{3},\sqrt{2},-\sqrt{5}\right)$, то ответом solve.bigq(A, b) станут обыкновенные дроби, наиболее близкие к истинным иррациональным результатам. То есть сам по себе ответ в виде обыкновенных дробей еще не говорит об абсолютной точности! Проверяйте результаты прямыми подстановками.

Замечание. Важно знать, что функция solve.bigq иногда отказывается работать с вполне хорошими невырожденными матрицами, ошибочно объявляя их вырожденными. В такие моменты всегда непосредственно проверяйте равенство нулю определителя det(A). В случае ошибки остается использовать лишь приближенное решение с помощью функции solve, которая с практической точки зрения ничем не уступает solve.bigq.

Впрочем, нет сомнений, что со временем подобная ошибка будет устранена авторами библиотеки.

Задания для самостоятельной работы

1. Разложить по базису

$$\begin{cases}\vec{e}_1=(3;9;12)\\ \vec{e}_2=(13;6;-5)\\ \vec{e}_3=(18;23;7)\end{cases}$$

a) вектор $\vec{a}(1;1;1)$
b) вектор $\vec{b}(1;3;4)$

2. Решить систему уравнений в обыкновенных дробях

$$\begin{cases}-4x+z=11\\ 3x+5y-2z=4\\ 6x-3y+7z=-8\end{cases}$$

и проверить результат.

3. Для матриц A и B

$$A=\begin{pmatrix}1 & 1 & 8\\ 8 & 0 & -2\\ 3 & 3 & 5\end{pmatrix} \quad \text{и} \quad B=\begin{pmatrix}2 & 3 & 8\\ 1 & 3 & 1\\ 1 & -7 & 0\end{pmatrix}$$

максимально точно решить матричное уравнение и проверить результат

$$AX = B$$
$$XA = B$$

4. Вычислить абсолютно точно и с точностью double-арифметики значение выражений:

a) 400^{300}

b) $200!$ Указание: используйте factorialZ(...)

c) C_{1000}^{500} Указание: используйте chooseZ(...)

Практикум 16. Спектральное и сингулярное разложение матриц (RStudio)

Этот раздел мы посвятим высшим аспектам линейной алгебры, связанным с ортогональными преобразованиями матриц и их представлению в виде сингулярных разложений. Обе трансформации играют чрезвычайно важную роль в анализе данных и, в частности, в регрессионном анализе.

Несмотря на кажущуюся теоретическую сложность в понимании самой сути соответствующих понятий, с вычислительной точки зрения в языке R все устроено очень и очень просто. Попробуем разобраться в основных идеях и способах их реализации.

Матрица линейного оператора

Рассмотрим произвольный линейный оператор A, преобразующий векторы какого-либо линейного пространства L. Напомним, что если линейный оператор задан матрицей A_e в некотором базисе $e_1, e_2, \ldots, e_n$, то зная координаты вектора x в этом базисе:

$$x = x_1 e_1 + x_2 e_2 + \ldots + x_n e_n,$$

легко получить координаты преобразованного вектора $y = Ax$. Для этого достаточно умножить матрицу оператора справа на столбец координат вектора x:

$$Y = A \cdot X$$

или в развернутом виде

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = A \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

Задание 1. Известно, что оператор поворота относительно начала координат двумерного вектора на угол $\frac{\pi}{3}$ против часовой стрелки в каноническом базисе задается матрицей поворота:

$$A = \begin{pmatrix} \cos\frac{\pi}{3} & -\sin\frac{\pi}{3} \\ \sin\frac{\pi}{3} & \cos\frac{\pi}{3} \end{pmatrix}.$$

Найти каким окажется вектор $a(7,-20)$ после такого поворота.

Решение. Умножим матрицу поворота на столбец координат поворачиваемого вектора $X = \begin{pmatrix} 7 \\ -20 \end{pmatrix}$:

$$Y = A \cdot X = \begin{pmatrix} \cos\frac{\pi}{3} & -\sin\frac{\pi}{3} \\ \sin\frac{\pi}{3} & \cos\frac{\pi}{3} \end{pmatrix} \begin{pmatrix} 7 \\ -20 \end{pmatrix},$$

что и даст нам координаты повернутого вектора.

В коде R получаем:

```
X <- c(7, -20)       # Вводим столбец координат вектора X
A <- cbind(c(cos(pi/3), sin(pi/3)), c(-sin(pi/3), cos(pi/3)))
  # Вводим матрицу поворота
Y <- A %*% X     # Вычисляем координаты повернутого
  # вектора Y=AX
Y                  # Выводим на экран столбец координат Y
```

с результатом в консоли:

```
> X <- c(7, -20)          # Вводим столбец координат
вектора X
```

```
> A <- cbind(c(cos(pi/3), sin(pi/3)), c(-sin(pi/3),
cos(pi/3)))  # Вводим матрицу поворота
> Y <- A %*% X          # Вычисляем координаты по-
вернутого вектора
> Y                     # Выводим на экран столбец
координат Y
          [,1]
[1,] 20.820508
[2,] -3.937822
```

Ниже на рисунке 134 представлен найденный поворот: синим – первоначальный вектор с координатами X, красным – повернутый вектор с координатами Y.

Код процедуры построения рисунка здесь следующий:

```
plot(0, 0, type = "p", col = "blue", xlim = c(-25, 25), ylim =
c(-25, 25))  # Строим начало координат и масштабируем
  # рисунок
abline(h = 0, v = 0, lty = "84")                    # Рисуем оси
  # координат пунктиром
arrows(0, 0, 7, -20, col = "blue",  lwd = 2)      # Строим
  # первый вектор X (стрелку)
arrows(0, 0, Y[1], Y[2], col = "red",  lwd = 2) # Строим
  # повернутый вектор Y (стрелку)
```

Возможно, на своем экране вы видите непропорциональный рисунок. Это зависит от того, насколько раскрыто по отношению к другим окнам в RStudio левое нижнее многофункциональное окно с графиком.

Замечание. Знайте, что всякий раз, когда изображение на экране компьютера поворачивается, каждая двумерная точка экрана претерпевает подобное преобразование.

Задание 2. Повернуть на угол $\frac{\pi}{3}$ против часовой стрелки относительно начала координат график функции

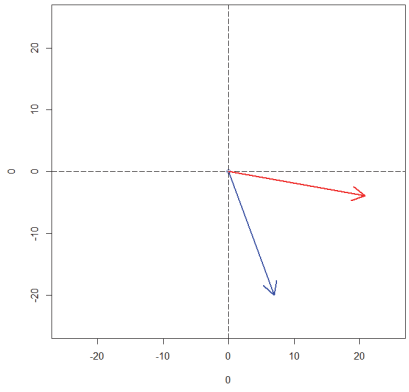


Рисунок 134

$$f(x)=x+\sin(3x).$$

Решение. Отличие данного задания от предыдущего заключается лишь в том, что повернуть надо не одну точку (одну пару координат), а множество точек, составляющих график функции.

Воспользуемся введенной матрицей поворота на угол $\frac{\pi}{3}$ и устроим повороты всех точек графика функции на отрезке, например, $[-5,5]$.

```
x <- seq(-5, 5, length.out = 100)   # Разбиваем отрезок [-5,
  # 5] на 100 точек
y <- x + sin(3*x)                   # Вычисляем значения
  # функции в точках разбиения
plot(x, y, type = "l", col = "blue", lwd = 2)  # Строим гра-
  # фик функции f(x)
abline(h = 0, v = 0, lty = "84")    # Рисуем оси координат
  # пунктиром
X <- rbind(x, y)                    # Образуем множество
  # пар точек, которые надо повернуть
Y <- A %*% X                        # Поворачиваем сразу все
  # точки матрицей поворотов
```

```
lines(Y[1, ], Y[2, ], type = "l", col = "red", lwd = 2)
  # Достраиваем повернутый график
```

с убедительным результатом (см. рис. 135):

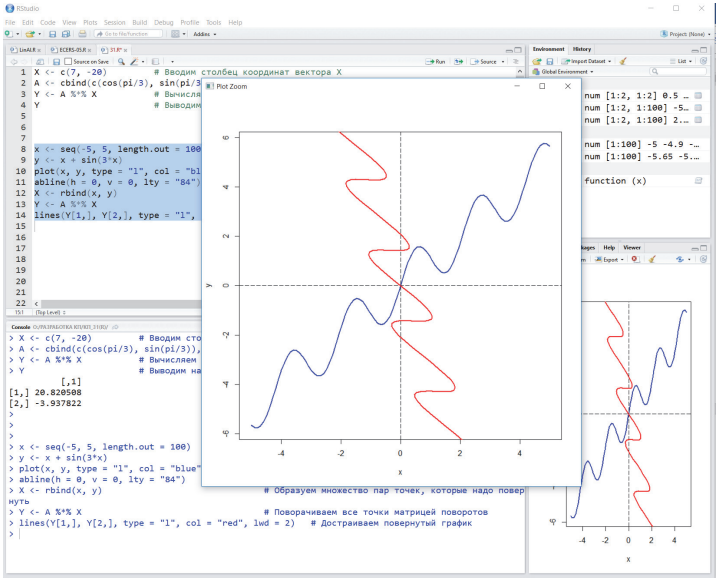


Рисунок 135

Преобразование матрицы линейного оператора

При переходе от одного базиса к другому матрица линейного оператора меняется, хотя действие самого оператора, разумеется сохраняется. Если переход от старого базиса, скажем, $e_1, e_2, \ldots, e_n$ к новому базису $f_1, f_2, \ldots, f_n$ задан матрицей перехода P, то связь между матрицами линейного оператора в этих базисах A_e и A_f задается следующим образом:

$$A_f = P^{-1} \cdot A_e \cdot P, \qquad (1)$$

$$A_e = P \cdot A_f \cdot P^{-1}. \qquad (2)$$

Может так оказаться, что в новом базисе матрица оператора выглядит проще, чем в исходном.

Задание 3. Матрица линейного оператора в базисе $e_1, e_2, \ldots, e_n$ имеет вид:

$$A_e = \begin{pmatrix} 23 & 0 & -6 \\ 24 & -1 & -6 \\ 74 & -4 & -18 \end{pmatrix},$$

а матрица перехода к новому базису $f_1, f_2, \ldots, f_n$ задана матрицей P:

$$P = \begin{pmatrix} 2 & 2 & 3 \\ 2 & 3 & 3 \\ 7 & 8 & 10 \end{pmatrix}.$$

Найти матрицу этого оператора A_f в новом базисе.

Решение. Введем указанные матрицы в Excel (рис. 136):

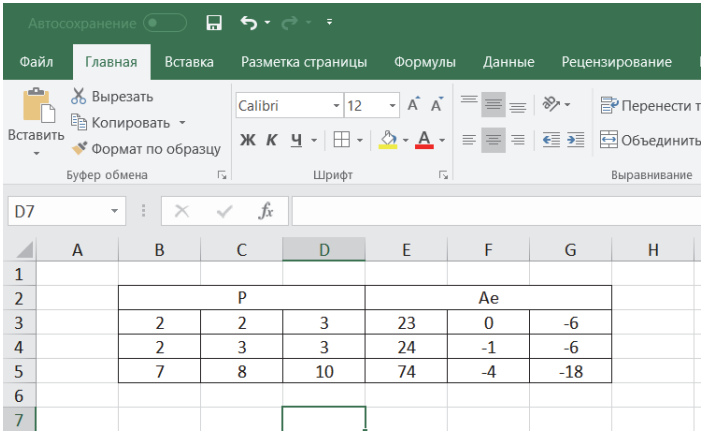


Рисунок 136

Воспользуемся далее буфером обмена и реализуем преобразование (1) $A_f = P^{-1} A_e P$:

```
Data <- read.table("clipboard",h=FALSE,dec=",",sep =
"\t")   # Чтение из буфера обмена excel W <- as.matrix.
```

```
data.frame(Data)                    # Объявляем таблицу чисел
  # Data матрицей W в R
P <- W[ , 1:3]; P                   # Первые три столбца матри-
  # цы W записываем в матрицу P
Ae <- W[ , 4:6]; Ae                 # Последние три столбца
  # матрицы W записываем в матрицу Ae
Af <- (solve(P) %*% Ae) %*% P    # Вычисляем матрицу
  # оператора Af в новом базисе
round(Af, 8)                        # Выводим округление ре-
  # зультата с точностью до 8 знаков
```

с результатом:

```
> Af <- (solve(P) %*% Ae) %*% P    # Вычисляем ма-
трицу оператора в новом базисе
> round(Af, 8)                  # Выводим округление
результата с точностью до 8 знаков
   V1 V2 V3
V1  2  0  0
V2  0 -1  0
V3  0  0  3
```

Как видим, матрица оператора в новом базисе выглядит очень простой, такой вид матриц называется диагональным.

В математике выделяют целый ряд специальных преобразований базиса, которые приводят матрицы операторов к наиболее простым видам. Мы рассмотрим некоторые из них и для краткости будем опускать слова базис и оператор, говоря просто лишь о самих преобразующихся матрицах.

Собственные числа и собственные векторы матриц

Напомним, что собственным числом (собственным значением) и соответствующим ему собственным вектором линейного оператора называются такое число

и такой ненулевой вектор , для которых справедливо равенство

$$Ax = \lambda x .$$

Если переписать это определение в координатной форме через матрицу оператора и столбец координат в каком-либо базисе, то мы получим аналог определения собственных значений и собственных векторов для матриц:

$$AX = \lambda X .$$

Задание 4. Матрица линейного оператора в некотором базисе имеет вид:

$$A_e = \begin{pmatrix} 23 & 0 & -6 \\ 24 & -1 & -6 \\ 74 & -4 & -18 \end{pmatrix}.$$

Найти собственные значения и собственные векторы данного оператора.

Решение. Учитывая, что матрица оператора A_e нами уже была введена в примере 3, нам остается только воспользоваться базовыми функциями в R для нахождения собственных значений и собственных векторов для матриц:

```
d <- eigen(Ae)$values; d          # Собственные значения
  # матрицы Ae
P <- eigen(Ae)$vectors; P         # Собственные векторы Ae,
  # стоящие в столбцах матрицы P
d[1]                              # Первое собственное значение
P[ , 1]                           # Первый собственный вектор
  # (первый столбец матрицы P)
```

с результатом:

```
> d <- eigen(Ae)$values; d   # Собственные значения
матрицы Ae
[1]  3  2 -1
> P <- eigen(Ae)$vectors; P  # Собственные векторы
Ae, стоящие в столбцах матрицы P
```

```
          [,1]      [,2]      [,3]
[1,] 0.2761724 0.2649065 0.2279212
[2,] 0.2761724 0.2649065 0.3418817
[3,] 0.9205746 0.9271726 0.9116846
> d[1]                          # Первое собственное
значение
[1] 3
> P[ , 1]                       # Первый собственный
вектор (первый столбец матрицы P)
[1] 0.2761724 0.2761724 0.9205746
```

Таким образом, мы нашли собственные числа для матрицы A_e, а значит и для оператора A. Найденные же собственные векторы матрицы A_e являются координатами собственных векторов для оператора A в соответствующем базисе.

Важное замечание! Использованная нами функция *eigen* возвращает в матрице P не просто столбцы собственных векторов, отвечающих собственным значениям. Если количество линейно независимых собственных векторов равно размерности матрицы, то представленные в столбцах линейно независимые собственные векторы образуют нормированный базис (длины векторов равны 1), а в случае симметричной матрицы – образуют ортонормированный базис, т.е. попарно векторы будут еще и ортогональны между собой. При этом сами собственные числа перечисляются по убыванию.

Проверьте, что длины каждого из собственных векторов в нашем случае равны единице, но ортогональности между векторами нет из-за несимметричности:

```
sum(P[ , 1]^2)              # Например, длина первого собст-
  # венного вектора
P[ , 2] %*% P[ , 3]         # Например, скалярное произведе-
  # ние (f2, f3)
```

Спектральное разложение

Если матрица линейного оператора вещественная и симметричная, то существует ортонормированный базис построенный из ее собственных векторов, в котором матрица оператора имеет диагональный вид, причем, на главной диагонали стоят соответствующие собственные значения.

В R получить такое разложение матрицы не представляет никакой проблемы, т.к. возвращаемая матрица из столбцов собственных векторов eigen(Ae)$vectors сама и является матрицей перехода P к новому ортонормированному базису, который диагонализирует оператор:

$$A_e = P \cdot A_f \cdot P^{-1} = P \cdot \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{pmatrix} \cdot P^{-1}$$

Такое разложение матриц часто называют спектральным, а матрицу перехода P – ортогональным преобразованием, т.к. фактически она определяет собой поворот одного ортонормированного базиса к другому ортонормированному.

Задание 5. Для матрицы линейного оператора в некотором ортонормированном базисе

$$A_e = \begin{pmatrix} 2 & 2 & -2 \\ 2 & 5 & -4 \\ -2 & -4 & 5 \end{pmatrix}$$

найти ортогональное преобразование, приводящее ее к диагональному виду.

Решение. Введем матрицу **Ae** через excel-формат, найдем ее собственные значения **d** и собственные векторы **P**, получим ее диагональный вид **D** в новом базисе и проверим истинность найденного разложения:

$$A_e = P \cdot D \cdot P^{-1}.$$

```
Data <- read.table("clipboard",h=FALSE,dec=",",sep = "\t")
# Чтение из буфера обмена
Ae <- as.matrix.data.frame(Data); Ae      # Объявляем
  # таблицу чисел Data матрицей Ae в R
d <- eigen(A)$values; d              # Собственные значения
  # матрицы Ae
P <- eigen(A)$vectors; P             # Собственные векторы
  # Ae, стоящие в столбцах матрицы P
P.inv <- solve(P)                    # Обратная матрица к P
D <- P.inv %*% (Ae %*% P); D    # Диагонализация ква-
  # дратной симметричной матрицы Ae
D <- diag(d); D                      # То же самое, но точно.
Ae                                   # Для сравнения выводим Ae
round(P %*% (D %*% P.inv), 8)        # И ее разложение
  #  A_e = P · A_f · P^-1
```

Результаты представлены на рисунке 187.

Интересно, что в этом случае матрица P, состоящая из столбцов ортонормированного базиса, обладает особыми свойствами:

$$P^T = P^{-1} \quad \text{и} \quad \left|\det(P)\right| = 1 .$$

Проверьте это в R для полученной матрицы P. Такие матрицы, кстати, называются ортогональными.

Замечание. Если матрица не симметричная, но линейно независимых собственных векторов хватает для построения базиса, то матрица P также будет приводить к диагональному виду в разложении $A_e = P \cdot A_f \cdot P^{-1}$, но чистым поворотом пространства данное преобразование уже может не быть. Этот случай мы рассмотрели в примере 4 с несимметричной матрицей.

Преобразование матрицы квадратичной формы

Обратите внимание, что из-за ортогональности матрицы перехода P в спектральном разложении, обратная матрица к P совпадает с транспонированной:

$$P^T = P^{-1} .$$

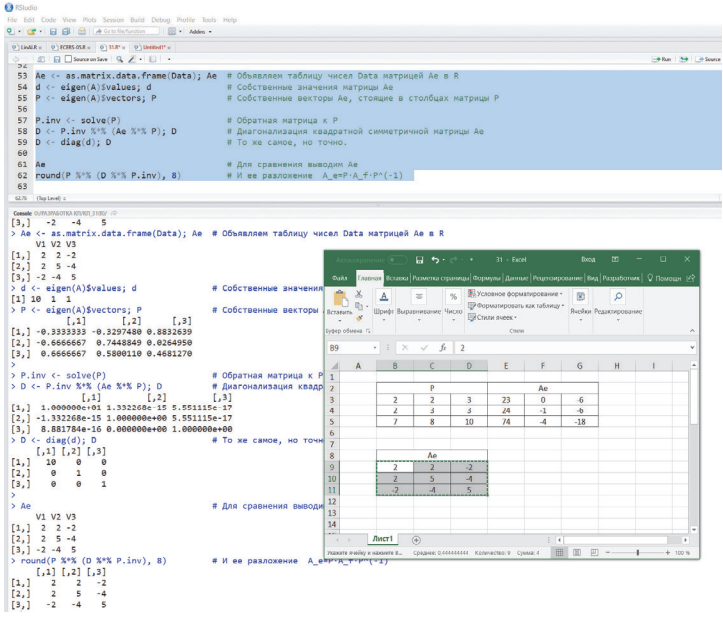


Рисунок 137

Таким образом, в этом случае формула преобразования матрицы оператора:

$$A_f = P^{-1} \cdot A_e \cdot P$$

может быть записана в виде:

$$A_f = P^T \cdot A_e \cdot P\,,$$

что совпадает с преобразованием матрицы квадратичной формы при соответствующей замене базиса. И фактически, таким ортогональным преобразованием P (поворотом пространства) мы приводим квадратичную форму к каноническому виду, причем без растяжений и сжатий. Вот почему ортогональные преобразования и спектральное разложение матриц получили такое широкое применение.

Задание 6. Привести квадратичную форму

$$\Phi(x_1, x_2, x_3) = 2x_1^2 + 4x_1x_2 - 4x_1x_3 + 5x_2^2 - 8x_2x_3 + 5x_3^2$$

методом ортогональных преобразований к каноническому виду.

Решение. Запишем матрицу квадратичной формы

$$A = \begin{pmatrix} 2 & 2 & -2 \\ 2 & 5 & -4 \\ -2 & -4 & 5 \end{pmatrix},$$

после чего нам остается найти ее спектральное разложение. Данную задачу для этой матрицы мы уже решили в предыдущем примере 5 (см. рис.3). Диагональная матрица квадратичной формы при этом получилась

$$D = \begin{pmatrix} 10 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

что приводит к следующему каноническому виду квадратичной формы в новых переменных:

$$\Phi(y_1, y_2, y_3) = 10y_1^2 + y_2^2 + y_3^2.$$

Сингулярное разложение матриц*
(Singular Values Decomposition, SVD)

Как мы уже отмечали выше, не все вещественные квадратные матрицы допускают спектральное разложение:

$$A = P \cdot \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{pmatrix} \cdot P^{-1},$$

где P – ортогональная матрица, составленная из собственных векторов матрицы A, а $\lambda_1,\ldots,\lambda_n$ – собственные значения матрицы A.

Одним из частично обобщающих разложений является сингулярное разложение матриц. По теореме

Дж. Форсайт для любой квадратной вещественной матрицы A найдутся такие две ортогональные матрицы U и V той же размерности, что будет выполняться следующее разложение, называемое сингулярным:

$$A = U \cdot \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \lambda_n \end{pmatrix} \cdot V^T \cdot$$

Причем, всегда можно так выбрать матрицы U и V, чтобы числа, стоящие на главной диагонали $\lambda_1,\ldots,\lambda_n$ располагались по убыванию.

В сингулярном разложении принята следующая терминология: столбцы матрицы U называют левыми сингулярными векторами, а столбцы матрицы V – правыми сингулярными векторами. Значения $\lambda_1,\ldots,\lambda_n$ называют сингулярными значениями матрицы A.

Для симметричных вещественных неотрицательных квадратных матриц спектральное и сингулярное разложения совпадают.

Замечание. Можно показать, что столбцы матрицы U являются собственными векторами матрицы AA^T, а квадраты сингулярных чисел $\lambda_1^2,\ldots,\lambda_n^2$ – ее собственными значениями. Аналогично столбцы матрицы V являются собственными векторами матрицы A^TA, а квадраты сингулярных чисел так же являются ее собственными значениями.

Задание 6. Для матрицы

$$A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

найти ее спектральное разложение.

Решение. Введем матрицу **A** через excel-формат, найдем ее сингулярные значения **d**, левую и правую матрицы разложения **U** и **V**, получим диагональный вид **D** в сингулярном разложении и проверим истинность найденного разложения по формуле:

$$A = U \cdot \mathrm{D} \cdot V^{\mathrm{T}}.$$

```
Data <- read.table("clipboard",h=FALSE,dec=",",sep =
"\t")   # Чтение из буфера обмена excel
A <- as.matrix.data.frame(Data); A   # Объявляем табли-
  # цу чисел Data матрицей A в R
S <- svd(A)                          # Производим сингуляр-
  # ное разложение матрицы A
d <- S$d; d                          # Вектор (массив) всех
  # сингулярных чисел
U <- S$u; U                          # Левая матрица U в син-
  # гулярном разложении
V <- S$v; V                          # Правая матрица V
  # в сингулярном разложении
D <- diag(S$d); D                    # Диагональная матрица
  # D в сингулярном разложении
U %*% D %*% t(V)                     # A = U D V' сингулярное
  # разложение A (проверка)
round(U %*% D %*% t(V), 8)     # то же самое, но на-
  # гляднее
```

с результатом (см. рис. 138)

Обратите внимание, что полученное сингулярное разложение всегда наглядно отражает ранг матрицы A. Действительно, получающаяся диагональная матрица разложения

$$D = \begin{pmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

имеет ранг 2, что по свойству сингулярного разложения совпадает с рангом исходной матрицы A.

Замечание. Сингулярное разложение оказывается определено и для неквадратных матриц. Единственное отличие состоит в том, что матрица V не будет квадратной и уж тем более ортогональной. Попробуйте получить сингулярное разложение для матрицы:

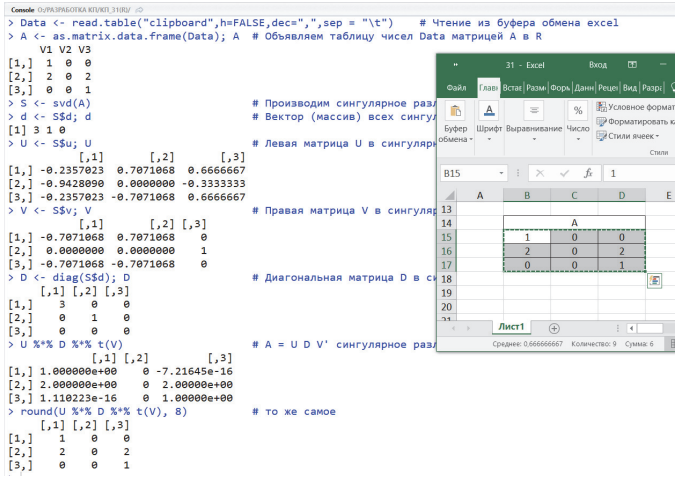


Рисунок 138

A <- matrix(1:12, 3, 4, byrow = TRUE); A

т.е. найти соответствующие матрицы U, V и D. Обязательно проверьте результат – сингулярное разложение матрицы должно совпадать с самой матрицей:

$$A = U \cdot \mathrm{D} \cdot V^{\mathrm{T}}.$$

Задания для самостоятельной работы

1. Как будет преобразован двумерный вектор $\vec{a}(-5,3)$ под действием линейного оператора, заданного в каноническом базисе матрицей

$$A_e = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Сделайте рисунок и попробуйте объяснить словами: что «делает» оператор с векторами.

2. Как изменится график арктангенса на отрезке $[-15,15]$ под действием линейного оператора с матрицей

$$A_e = \begin{pmatrix} 1 & 0 \\ 0 & -0.4 \end{pmatrix},$$

заданной в каноническом базисе. Ответить на вопрос сравнительным изображением графиков арктангенса и его трансформированного образа.

Указание. Воспользуйтесь аналогией с примером *2, в котором мы поворачивали график функции с помощью матрицы оператора.*

3. Найдите собственные значения и собственные векторы для матриц

a) $A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$

b) $B = \begin{pmatrix} 2 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 4 \\ 1 & 0 & 0 & 6 & 1 \\ 0 & 3 & 1 & 0 & 0 \\ 0 & 1 & 5 & 0 & 1 \end{pmatrix}$

c) $Q = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix}$. При каких значениях φ собственные числа вещественны?

4. Для матрицы получить спектральное и сингулярное разложения и сравнить полученные диагональные виды между собой. Чему равен ранг матрицы?

a) $A = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$

a) $F = \begin{pmatrix} 1 & 8 & -4 \\ 8 & 0 & 3 \\ -4 & 3 & 5 \end{pmatrix}$

c) $G = \begin{pmatrix} 1 & -2 & 3 & 1 \\ -2 & 0 & 1 & 1 \\ -1 & -2 & 4 & 2 \\ 3 & -2 & 2 & 0 \end{pmatrix}$

5. Привести квадратичную форму

$$\Phi(x_1, x_2, x_3, x_4) = 4x_1x_3 - 6x_2x_4$$

к каноническому виду методом ортогональных преобразований.

Практикум 17. Задачи линейной оптимизации (RStudio)

Последний раздел нашего учебника по R посвящен решению задач линейной оптимизации.

В идеале было бы хорошо заранее разобраться и прорешать несколько типов таких задач чисто аналитически. Это помогло бы не только понять основную идею решения таких задач, но и ощутить всю вычислительную мощь с одновременной простотой используемых команд в R.

Внимательно изучите каждый комментарий к строчкам приводимого кода. Для желающих основательно разобраться в линейном программировании и понять детали его экономического и физического смысла в конце раздела приведены ссылки на соответствующую литературу [1], [2].

Стандартная задача линейного программирования

Задание 1. Решить стандартную задачу линейного программирования

$$f(x_1, x_2) = 10x_1 + 20x_2 \to \max$$

$$\begin{cases} x_1 + 3.5x_2 \le 350 \\ 2x_1 + 0.5x_2 \le 240 \\ x_1 + x_2 \le 150 \\ x_2 \ge 60 \\ x_1, x_2 \ge 0 \end{cases}$$

P.S. Задача заимствована из пособия [1] (пример 1.1 задача о костюмах, стр. 12)

Решение. Разобьем решение задачи линейного программирования на несколько этапов.

Объявление в **R** задачи линейного программирования. Загружаем пакет *lpSolveAPI* из репозитория CRAN и отрабатываем следующий ниже код R.

```
library(lpSolveAPI)              # Активируем библиотеку
  # линейного программирования
M <- make.lp(ncol = 2)           # Объявляем количество
  # неотрицательных переменных в M
name.lp(M, "Example")            # Объявляем название
  # "Example" для задачи (модели) M
colnames(M) <- c("X1", "X2")     # Объявляем названия
  # переменных в модели M
lp.control(M, sense = "max")$sense   # Объявляем задачу
  # на максимум модели M
set.objfn(M, c(10, 20))          # Задаем целевую функцию:
  # f = 10*X1 + 20*X2 для модели M
add.constraint(M, c(1, 3.5), "<=", 350)   # Задаем ограни-
  # чение: 1*X1+3.5*X2 <= 350 для M
add.constraint(M, c(2, 0.5), "<=", 240)   # Аналогично
add.constraint(M, c(1, 1), "<=", 150)     # Аналогично
add.constraint(M, c(0, 1), ">=", 60)      # Аналогично
rownames(M) <- c("A", "B", "C", "D")   # Называем огра-
  # ничения в модели M
M                                # Смотрим какая модель M в итоге
  # получилась / можно print(M)
```

Запускаем введенный код на компиляцию с результатом:

```
> M                # Смотрим какая модель M в итоге получилась / можно print(M)
Model name: Example
            X1    X2
Maximize    10    20
A            1   3.5  <=  350
B            2   0.5  <=  240
C            1     1  <=  150
D            0     1  >=   60
Kind       Std   Std          # тип – стандартная
Type      Real  Real          # переменные – вещественнозначные, double
Upper      Inf   Inf          # верхние границы изменения переменных
Lower        0     0          # нижние границы изменения переменных
```

Решение задачи линейного программирования. Далее решаем составленную задачу линейного программирования M:

```
solve.lpExtPtr(M)       # Если 0, то ошибок нет – достигнуто оптимальное решение
get.variables(M)        # Оптимальный план
get.objective(M)        # Достигнутый max
```

```
> solve.lpExtPtr(M)  # Если 0, то ошибок нет – достигнуто оптимальное решение
[1] 0
> get.variables(M)
[1] 70 80             # Оптимальные значения переменных X1 и X2 в модели M
> get.objective(M)
[1] 2300              # Значение максимума целевой функции f в модели M
```

Разумеется, можно сохранить полученное решение задачи M в соответствующие переменные:

```
X1.opt <- get.variables(M)[1]; X1.opt      # Оптимальное
  # значение для X1
X2.opt <- get.variables(M)[2]; X2.opt      # Оптимальное
  # значение для X2
f.max <- get.objective(M); f.max           # Значение целе-
  # вой функции на оптимальном решении
```

Здесь также могут быть полезны следующие дополнительные возможности по анализу характера полученного оптимального решения.

Оценка дефицита ресурсов:

```
# Значения правых частей неравенств системы:
b <- get.constr.value(M); b                # Изначально задан-
  # ные в системе неравенства
b.opt <- get.constraints(M); b.opt         # Достигнутые значе-
  # ния на оптимальном решении
round(abs(b - b.opt), 10)                  # Строка дефицита
  # (исчерпанность) ресурсов
```

```
> b <- get.constr.value(M); b        # Изначально
заданные в системе неравенств
[1] 350 240 150  60
> b.opt <- get.constraints(M); b.opt # Достигнутые
на оптимальном решении
[1] 350 180 150  80
> round(abs(b - b.opt), 10)          # Строка дефи-
цита ресурсов
[1]  0 60  0 20
```

Оценка устойчивости коэффициентов целевой функции:

```
# Минимальные и максимальные коэфф. в целевой
  # функции, сохраняющие оптимальность:
min <- get.sensitivity.obj(M)$objfrom; min
max <- get.sensitivity.obj(M)$objtill; max
# Диапазоны коэффициентов целевой функции, сохра-
```

```
  # няющие оптимальность:
cbind(min, max)       # i-ая строка – это диапазон устой-
  # чивости для i-ого коэффициента
```

```
> min <- get.sensitivity.obj(M)$objfrom; min
[1]  5.714286 10.000000
> max <- get.sensitivity.obj(M)$objtill; max
[1] 20 35
> cbind(min, max)   # i-ая строка – диапазон устой-
чивости для i-ого коэффициента
           min max
[1,]  5.714286  20   # диапазон устойчивости опти-
мальности для 1-ого коэфф.
[2,] 10.000000  35   # диапазон устойчивости опти-
мальности для 2-ого коэфф.
```

Решение двойственной к **M** задачи:

```
get.sensitivity.rhs(M)$duals      # Оптимальный план
  # двойственной задачи
# Минимальные и максимальные коэфф. в целевой
  # функции, сохраняющие оптимальность:
min.dual <- get.sensitivity.rhs(M)$dualsfrom; min.dual
max.dual <- get.sensitivity.rhs(M)$dualstill; max.dual
# Диапазоны коэффициентов целевой функции, сохра-
  # няющие оптимальность:
cbind(min.dual, max.dual)   # i-ая строка – это диапазон
  # устойчивости для i-ого коэффициента
```

```
> get.sensitivity.rhs(M)$duals       # Оптимальный
план двойственной задачи (Y1, Y2, Y3 и Y4)
[1] 4 0 6 0 0 0
>  min.dual  <-  get.sensitivity.rhs(M)$dualsfrom;
min.dual
[1]  3e+02 -1e+30  1e+02 -1e+30 -1e+30 -1e+30
>  max.dual  <-  get.sensitivity.rhs(M)$dualstill;
max.dual
```

```
[1]   5.250000e+02   1.000000e+30   1.730769e+02
1.000000e+30 1.000000e+30 1.000000e+30
> cbind(min.dual, max.dual)   # i-ая строка – это
диапазон устойчивости для i-ого коэффициента
     min.dual     max.dual
[1,]    3e+02 5.250000e+02
[2,]   -1e+30 1.000000e+30
[3,]    1e+02 1.730769e+02
[4,]   -1e+30 1.000000e+30
[5,]   -1e+30 1.000000e+30
[6,]   -1e+30 1.000000e+30
```

Справка по возникающим ошибкам:

```
?solve.lpExtPtr(M)              # Вызов справки по ошиб-
 # кам при решении задачи ЛП
```

0: «optimal solution found» – найдено оптимальное решение
1: «the model is sub-optimal» – модель не имеет оптимального решения
2: «the model is infeasible» – max/min не достижим (возможно область пуста, возможно область не является ограниченной)
3: «the model is unbounded» – max/min не достижим, т.к. область неограниченна
4: “the model is degenerate” – модель вырождена
5: «numerical failure encountered» – ошибка вычислений, обычно связана с противоречивыми добавками в модель на стадии ее повторной прогонки
6: «process aborted» – процесс прерван (возможно пользователем)
7: «timeout» – истекло время, отводимое на необходимые вычисления, вероятно произошло зацикливание между двумя опорными решениями
9: «the model was solved by presolve» – модель была решена на стадии выбора опорного решения (вероятно имеется в виду)

10: «the branch and bound routine failed» – разветвления границы области слишком нетипичны, возможно их очень много
11: «the branch and bound was stopped because of a break-at-first or break-at-value» – нахождение угловых точек было прервано из-за изменения модели между сеансами обращения к ее решению (вероятно имеется в виду)
12: «a feasible branch and bound solution was found» – найдено одно из возможных решений (вероятно имеется в виду)
13: «no feasible branch and bound solution was found» – не найдено ни одного опорного решения (вероятно имеется в виду).

Целочисленное линейное программирование

Задание 2. Решить задачу линейного целочисленного программирования

$$f(x_1, x_2, x_3, x_4) = 4x_1 + 3x_2 + 5x_3 + 6x_4 \to \min$$

$$\begin{cases} x_1 + x_2 + 2x_3 + 3x_4 = 123 \\ 2x_1 + x_2 + x_3 = 471 \\ x_3 \le 300 \\ x_4 \le 400 \\ x_1, x_2, x_3, x_4 \in Z \end{cases}$$

Решение. Составим следующий ниже код на R.

Объявление в **R** задачи целочисленного линейного программирования (внимательно изучите комментарии к каждой строке):

```
M.Int <- make.lp(ncol = 4)          # Объявляем количество
  # неотрицательных переменных в M.Int
name.lp(M.Int, "Example Int")                  # Объявляем
  # название задачи для M.Int
colnames(M.Int) <- c("X1","X2","X3","X4")  # Объявля-
  # ем названия переменных в M.Int
```

```
lp.control(M.Int, sense = "min")$sense          # Объявляем
  # задачу на максимум модели M.Int
set.objfn(M.Int, c(4, 3, 5, 6))  # Задаем целевую функцию:
  # f = 4*X1 + 3*X2 + 5*X3 + 6*X4

# Задаем первое ограничение: 1*X1+1*X2+2*X3+3*X4
  # >= 100 для модели M.Int:
add.constraint(M.Int, c(1, 1, 2, 3), "=", 123)
add.constraint(M.Int, c(2, 1, 1, 0), "=", 471)   # Аналогично
rownames(M.Int) <- c("A", "B")          # Называем огра-
  # ничения в модели M.Int
set.type(M.Int, c(1,2,3,4), "integer")   # Указываем номе-
  # ра целых переменных

# Указываем диапазоны изменения всех четырех пере-
  # менных, при этом
# lower – вектор нижних границ переменных, upper –
  # правых. Inf – символ бесконечности:
set.bounds(M.Int, lower = c(-Inf, -Inf, -Inf, -Inf), upper =
c(Inf, Inf, 300, 400))
M.Int      # Смотрим какая модель M.Int в итоге получилась
```

Запускаем код на компиляцию с результатом:

```
> M.Int
Model name: Example Int
            X1    X2    X3    X4
Minimize     4     3     5     6
A            1     1     2     3  =  123
B            2     1     1     0  =  471
Kind       Std   Std   Std   Std
Type       Int   Int   Int   Int
Upper      Inf   Inf   300   400
Lower     -Inf  -Inf  -Inf  -Inf
```

Решение задачи линейного программирования. Далее решим составленную задачу линейного программирования M.Int:

```
solve.lpExtPtr(M.Int)       # Если 0, то ошибок нет –
  # достигнуто оптимальное решение
```

```
get.variables(M.Int)          # Оптимальный целочислен-
  # ный план
get.objective(M.Int)          # Достигнутый минимум
```

```
> solve.lpExtPtr(M.Int)
[1] 0
> get.variables(M.Int)
[1]   84    3  300 -188
> get.objective(M.Int)
[1] 717
```

Попробуйте прогнать тот же код, но без указания целочисленности решения, т.е. предварительно «закомментируйте» строку символом хэша #:

```
# set.type(M.Int, c(1,2,3,4), "integer")      # Указываем
  # номера целых переменных
```

и сравните ответ

```
> solve.lpExtPtr(M.Int)
[1] 0
> get.variables(M.Int)
[1]   85.5    0.0  300.0 -187.5
> get.objective(M.Int)
[1] 717
```

с полученным ранее целочисленным вариантом.

Вопрос для отличников: Если сравнивать ответы для этих двух задач, то можно заметить что, хотя оптимальные планы задач и получились разными, значение целевой функции на оптимальных решениях одинаково 717. О чем это говорит?

Транспортная задача

Задание 3. Решить транспортную задачу закрытого типа.

Склады A_i	Тарифы c_{ij} на перевозку со склада A_i в магазин B_j ед. объема продукции					Запасы складов
	B_1	B_2	B_3	B_4	B_5	
A_1	1	2	3	4	5	160
A_2	4	3	2	7	3	150
A_3	5	5	4	2	8	220
Потребности магазинов	140	90	100	50	150	$\Sigma = 530$

Решение. Кратко напомним, что искомым оптимальным планом в транспортной задаче является матрица X, состоящая из чисел x_{ij} – количества перевезенной продукции со склада A_i в магазин B_j, а целевой функцией F служит общая стоимость такой перевозки:

$$F = c_{11}x_{11} + \ldots + c_{35}x_{35} = 1x_{11} + 2x_{12} + 3x_{13} + \ldots + 8x_{35} \to \min .$$

Данная задача является задачей закрытого типа, т.к. сумма всех потребностей магазинов равна сумме всех запасов складов $\Sigma = 530$. Поэтому, системы линейных ограничений по складам и по магазинам здесь выглядят в виде уравнений (точных равенств):

$$\begin{cases} A_1 : x_{11} + x_{12} + x_{13} + x_{14} + x_{15} = 160 \\ A_2 : x_{21} + x_{22} + x_{23} + x_{24} + x_{25} = 150 \\ A_3 : x_{31} + x_{32} + x_{33} + x_{34} + x_{35} = 220 \\ \\ B_1 : x_{11} + x_{21} + x_{31} = 140 \\ B_2 : x_{12} + x_{22} + x_{32} = 90 \\ B_3 : x_{13} + x_{23} + x_{33} = 100 \\ B_4 : x_{14} + x_{24} + x_{34} = 50 \\ B_5 : x_{15} + x_{25} + x_{35} = 150 \\ \\ x_{ij} \ge 0 \end{cases} .$$

Объявление в R задачи линейного программирования. Можно было бы поступить аналогично предыдущим задачам линейного программирования, но мы воспользуемся другим специальным ресурсом – еще одной полезной библиотекой *lpSolve* (см. также рис. 139).

```
library(lpSolve)                    # Активируем библиотеку
  # линейного программирования
# В excel вводим матрицу тарифов и бросаем в буфер
  # обмена.
# Читаем из буфера обмена данные в Data:
Data <- read.table("clipboard",h=FALSE,dec=",",sep = "\t")
# Чтение из буфера обмена excel-формата
P <- as.matrix(Data); P             # Объявляем таблицу Data
  # матрицей P в R
```

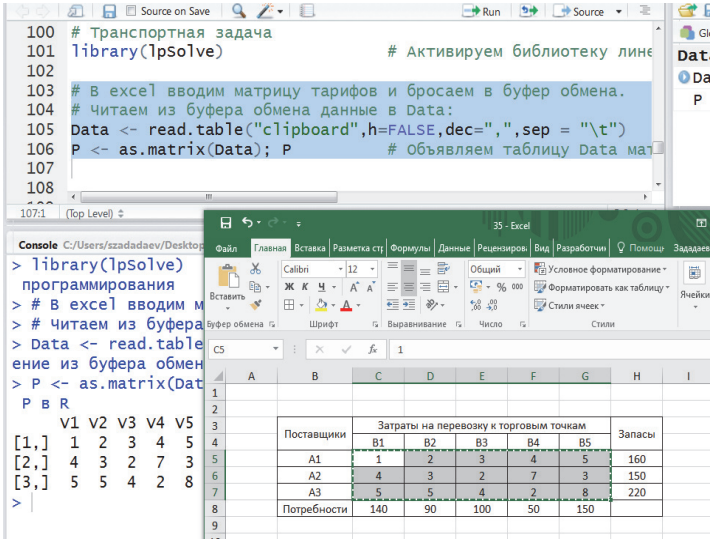


Рисунок 139

Далее вводим ограничения по запасам и потребностям, знаки ограничений (везде двойное равно ==) и объявляем транспортную задачу в терминах R:

```
SignA <- c("==", "==", "==")          # Знаки в ограниче-
  # ниях складов Ai / а еще лучше: rep("==", 3)
SignB <- c("==", "==", "==", "==", "==")      # Знаки
  # в огранчниях магазинов Bj / rep("==", 5)
SumA <- c(160, 150, 220)              # Запасы по складам Ai
SumB <- c(140, 90, 100, 50, 150)              # Потребности
  # магазинов Bj
# Записываем составленную транспортную задачу в пере-
  # менную M.Tr:
M.Tr <- lp.transport(cost.mat = P, direction = "min",
        row.signs = SignA, row.rhs = SumA,
        col.signs = SignB, col.rhs = SumB)
```

Не забудьте набранный код отправить на компиляцию (Ctrl+Enter).

Решение транспортной задачи. Далее решаем составленную транспортную задачу M.Tr:

```
M.Tr$status     # Если 0 - решение найдено
M.Tr$solution   # Оптимальный план перевозок (матрица X)
M.Tr$objval     # Минимальные суммарные транспорт-
  # ные расходы на перевозку
```

с результатом в консоли

```
> M.Tr$status     # Если 0 - решение найдено
[1] 0
> M.Tr$solution     # Оптимальный план перевозок
(матрица X)
     [,1] [,2] [,3] [,4] [,5]
[1,]  140   20    0    0    0
[2,]    0    0    0    0  150
[3,]    0   70  100   50    0
> M.Tr$objval      # Минимальные суммарные транс-
портные расходы на перевозку
[1] 1480
```

Полный перечень произведенных вычислений по транспортной задаче M.Tr можно получить командой:

```
Str(M.Tr)
```

```
> str(M.Tr)
List of 20
 $ direction     : int 0
 $ rcount        : int 3
 $ ccount        : int 5
 $ costs        : num [1:16] 0 1 4 5 2 3 5 3 2 4 ...
 $ rsigns        : int [1:3] 3 3 3
 $ rrhs          : num [1:3] 160 150 220
 $ csigns        : int [1:5] 3 3 3 3 3
 $ crhs          : num [1:5] 140 90 100 50 150
 $ objval        : num 1480
 $ int.count     : int 15
 $ integers     : int [1:15] 1 2 3 4 5 6 7 8 9 10 ...
 $ solution      : num [1:3, 1:5] 140 0 0 20 0 70
0 0 100 0 ...
 $ presolve      : int 0
 $ compute.sens  : int 0
 $ sens.coef.from: num [1:3, 1:5] 0 0 0 0 0 0 0 0
0 0 ...
 $ sens.coef.to  : num [1:3, 1:5] 0 0 0 0 0 0 0 0
0 0 ...
 $ duals         : num [1:3, 1:5] 0 0 0 0 0 0 0 0
0 0 ...
 $ duals.from    : num [1:3, 1:5] 0 0 0 0 0 0 0 0
0 0 ...
 $ duals.to      : num [1:3, 1:5] 0 0 0 0 0 0 0 0
0 0 ...
 $ status        : int 0
 - attr(*, “class”)= chr “lp”
```

часть из которых мы рассмотрели выше.

Замечание. Транспортная задача открытого типа будет отличаться от этой вместо равенств (==) нестрогими неравенствами (<=) в определении ограничений по суммарным запасам или потребностям, смотря каких больше. Подробнее см. [1] в конце этого раздела.

Задания для самостоятельной работы

1. Решить задачу линейного программирования смешанного типа

$$f(x_1,x_2,x_3,x_4,x_5)=2x_1-x_2+x_3+3x_4+7x_5\to\min$$

$$\begin{cases} 3x_1+4x_2+9x_3-3x_4\le 200 \\ x_1+6x_2+x_3+2x_4+x_5\ge 60 \\ x_1+x_2+x_3-x_4+3x_5=40 \\ x_1\ge 15 \\ x_2\le 50 \\ -40\le x_4\le 25 \\ x_1,x_2,x_4\in R, \\ x_3,x_5\in Z \end{cases}$$

Ответ:

```
Model name: Example 1
            X1    X2    X3    X4    X5
Minimize     2    -1     1     3     7
A            3     4     9    -3     0  <=  200
B            1     6     1     2     1  >=   60
C            1     1     1    -1     3   =   40
Kind       Std   Std   Std   Std   Std
Type      Real  Real   Int  Real   Int
Upper      Inf    50   Inf    25   Inf
Lower       15  -Inf  -Inf   -40  -Inf
>
> solve.lpExtPtr(M.Int)
[1] 0
> get.variables(M.Int)
[1]  17  50 -19 -40 -16
> get.objective(M.Int)
[1] -267
```

2. * Решить транспортную задачу открытого типа

Склады A_i	Тарифы c_{ij} на перевозку со склада A_i в магазин B_j ед. объема продукции						Запасы складов
	B_1	B_2	B_3	B_4	B_5	B_6	
A_1	3	1	7	1	2	3	250
A_2	2	3	1	4	4	3	120
A_3	6	1	3	5	1	2	150
A_4	4	4	3	2	4	2	100
Потребности магазинов	90	60	50	300	40	70	

Совет: для таблиц числовых данных буфер обмена сработает и из MS Word с теми же параметрами в R.

Указание: для открытого типа какое-то из уравнений по суммарным потребностям или запасам превратится в нестрогое неравенство.

Ответ:

```
> M.Tr$status      # Если 0 - решение найдено
[1] 0
> M.Tr$solution     # Оптимальный план перевозок (матрица X)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    0    0  250    0    0
[2,]   90    0   30    0    0    0
[3,]    0   60   20    0   40   30
[4,]    0    0    0   50    0   40
> M.Tr$objval      # Минимальные суммарные транспортные издержки
[1] 860
```

Рекомендуемая литература к практикуму 17

1. Экономико-математическое моделирование: практическое пособие по решению задач в Excel и R / И.В. Орлова, М.Г. Бич. – 3-е изд., испр. и доп. – М.: Вузовский учебник: ИНФРА-М, 2018. – 190с.

2. Методы оптимальных решений. Практикум: учебное пособие / коллектив авторов; под ред. В.А.Колемаева, В.И.Соловьева. – Москва: КНОРУС, 2017. – 194с.

Заключение

Теперь, когда в целом мы имеем достаточное представление о языке программирования R, можно обратить свое внимание на истинное предназначение в математике этого замечательного языка – на его применение в анализе данных.

Здесь можно выделить несколько направлений, в которых удобно получать и сами результаты, и их наглядное визуальное представление – это теоретико-вероятностное моделирование событий и процессов; описательная статистика и проверка статистически гипотез; эконометрические методы, в т.ч. исследование регрессионных моделей; анализ временных рядов; методы кластерного, факторного и дискриминантного анализа; случайные деревья, леса и многое другое.

Вообще, специализированных библиотек для R существует невероятно большое количество. Вопрос остается лишь в выборе научной области на основе своего прикладного интереса.

Глоссарий

Важно: в приведенном глоссарии указаны номера глав (практикумов), а не номера страниц.

Учебное издание

Зададаев Сергей Алексеевич

МАТЕМАТИКА НА ЯЗЫКЕ R

Учебник

Дизайн обложки *Середы Т.В.*
Компьютерная верстка *Середы Т.В.*

Подписано в печать 06.03.2018
Формат 60×84/16. Объем 20,3 п.л.
Тираж 1000 экз. Заказ № 708

Издательство «Прометей»
115035, г. Москва, ул. Садовническая, д. 72, стр. 1
Тел./факс: 8 (495) 799-54-29
E-mail: info@prometej.su

ISBN 978-5-907003-59-0