The Analysis of Data

Volume 2

Computing

Guy Lebanon

Google Research and
Georgia Institute of Technology

First Edition

TAOD Press

August 30, 2012

To Anat Lebanon

Contents

Chapter 1

Hardware

Chapter 2

The Operating System

Chapter 3

C and C++ Programming

Chapter 4

R Programming

This chapter provides a self-contained introduction to the R programming language. The following chapter provides an introduction to graphing data with R. A reader who understands these two chapters can not only understand the code segments throughout the book, but also modify the code segments and run additional experiments.

R is similar to Matlab and Python in the following ways:

- They run inside an interactive shell in their default setting. Often this shell is a complete graphical user interface (GUI).
- They emphasize storing and manipulating data as multidimensional arrays.
- They interface packages featuring functionalities, both elementary, such as matrix operations, SVD, eigenvalues solvers, random number generators, and optimization routines; and specialized, such as kernel smoothing and support vector machines.
- They exhibit execution time slower than that of C, C++, and Fortran, and are thus poorly suited for analyzing massive[1] data.
- They can interface with native C or C++ code, supporting limited large scale data analysis by implementing computational bottlenecks in C or C++.

The three languages differ in the following ways:

- R and Python are open-source and freely available for Windows, Linux, and Mac, while Matlab requires an expensive license.
- R, unlike Matlab, features ease in extending the core language by writing new packages, as well as in installing packages contributed by others.

[1]The slowdown can be marginal or significant, depending on implementation. Vectorized code may improve computational efficiency by performing basic operations on entire arrays rather than on individual array elements inside nested loops.

- R features a large group of motivated contributors who enhance the language by implementing high-quality packages[2].
- Developers designed R for statistics, yielding a syntax much better suited for computational data analysis, statistics, and data visualization.
- Creating high quality graphs in R requires less effort than it does in Matlab and Python.
- R is the primary programming language in the statistics, biostatistics, and social sciences communities, while Matlab is the main programming language in engineering and applied math, and Python is a popular general purpose scripting language.

We chose R primarily due to licensing, the active user community, and the ease of creating high quality graphs.

4.1 First Commands

The first step is to download and install a copy of R on your computer. R is available freely for Windows, Linux, and Mac at the R Project website http://cran.r-project.org/.

The two most common ways to run R are

- inside a GUI, by double clicking the R icon on Windows or Mac or by typing `R -g Tk &` on Linux, and
- in a terminal prompt, by typing `R` in a Linux or Mac terminal.

Other ways to run R include

- using the Emacs Speaks Statistics (ESS) package from within Emacs,
- using a third party GUI such as R-Studio (available from http://www.rstudio.org; see Figure **??** for a screen shot), and
- running R from a client that accesses an R server application.

The easiest way to quit R is to type `q()` at the prompt or to close the corresponding GUI window.

Like other programming languages, R code consists of a sequential collection of commands. Often, every command in R is typed in its own line. Alternatively, one line may contain multiple commands, separated by semicolons.

[2]Figure **??** demonstrates that the number of packages grows exponentially (see also **?**). All contributed packages go through a quality control process, and enforcement of standards, and have common documentation format.

```
## Error:  cannot open the connection
## Error:  object 'Packages' not found
## Error:  object 'Packages' not found
```

Figure 4.1: A linear growth on the log scale of the number of contributed R packages indicates an exponential growth. See **?** for more details and an interesting description of the open source social movement behind R. The straight line shows linear regression fit.

Figure 4.2: Screenshot of the R-Studio R development environment available from http://www.rstudio.org.

```
a = 4
a = 4
b = 3
c = b
```

Since R, unlike C++ or Java, is not strongly typed, we can assign a variable without expressing a type and can even change a variable's type within the same session. Thus, the following is quite legal.

```
a = 3.2
a = "string"
```

We can display the value of any variable using the `print()` function or by simply typing its name. Comments require a hash sign # and occupy the remainder of the current line.

```
a = 4
print(a)

## [1] 4

a  # same thing

## [1] 4

cat(a)  # same thing

## 4

cat("value of variable a is:", a, "\n")

## value of variable a is: 4
```

As shown above, we set variables with the assignment operator =. An alternative is the operator <-, as in `a <- 4` or `4 -> a`. We use the former in this book since = is more similar to assignment in other languages. The operator <<- sets global variables.

Strings in R are sequences of case sensitive characters surrounded by single or double quotes. To get help on a specific function, operator, or data object type `help(X)` where `X` is a string. Similarly, `example(X)` shows an example of the use of the function `X`. The function `help.start(X)` starts html-based documentation within a browser, which is sometimes easier to navigate. Searching the help documentation using `help.search(X)` is useful if you cannot recall the precise string on which you want to seek help. The function `RSiteSearch(X)` searches the R discussion boards for information concerning the string `X` within a browser. In addition to the commands above, searching the web using a search engine often provides useful results.

Here are some important commands with short explanations.

```
x = 3  # assigning value 3 to variable x
y = 3 * x + 2  # basic variable assignment and arithmetic
ls()  # lists variable names in workspace memory
ls(all.names = TRUE)  # lists all variable names including
hidden ones
ls.str()  #  annotated list of variable names
# saves all variables to a file
save.image(file = "fname")
# saves specified variables
save(x, y, file = "fname")
rm(x, y)  # clears variables x and y from memory
rm(list = ls())  # clears all variables in workspace memory
load(varFile)  # loads variables from file back to the workspace
history(15)  # displays 15 most recent commands
```

The precise interaction of the command line or R GUI tool (like R-Studio) depends on the operating system and GUI program. In general, pressing the up arrow key and down arrow key allows searching through the command history for previous commands. This can be very useful for fixing typos or for executing slight modifications of a long command line. In the Linux and Mac terminal, Control-R conveniently takes a text pattern and returns the most recent command containing that pattern.

Upon exiting R, the command line prompts the user to save the workspace memory. Saving places all current variables in a file named `.RData` in the current directory. Starting R automatically uploads that file if it exists in the current directory, retrieving the set of variables from the previous session in that directory. Inside R, the user can change directories or view the current directory using `setwd(X)` and `getwd()`, respectively. The function `system(X)` executes shell commands.

```
setwd("~")  # changes to home directory
# displays all files in current directory
list.files(all.files = TRUE)
```

```
system("ls -al")  #  same using the system command in Linux or
Mac
```

As stated earlier, R features easy installation of both core R and third party packages. The function `install.packages(X)` installs the functions and datasets in the package `X` from the internet. After installation the function `library(X)` brings the package into scope, thus making available `X`'s functions and variables. This two-stage process mitigates namespace overlap among the currently loaded libraries[3]. Typically, an R programmer would install many packages on his or her computer, but have only a limited number in scope at any particular time. A list of available packages, their implementation and documentation is available at http://cran.r-project.org/web/packages/. These packages often contain interesting and demonstrative datasets. The function `data` lists available datasets in a particular package.

```
# installs package
install.packages("ggplot2")
# installs package from a particular mirror
install.packages("ggplot2", repos = "http://cran.r-project.org")
# installs a package from source, rather
# than binary
install.packages("ggplot2", type = "source")
library("ggplot2")  # brings package into scope
# displays all datasets in the package
# ggplot2
data(package = "ggplot2")
installed.packages()  # displays a list of installed packages
update.packages()  # updated currently installed packages
```

R attempts to match a command-reference name by searching the current working environment and packages in scope for a variable or function matching the name (recall the `library` function), using the earliest match. The function `search` exhibits that search path. The first entry is typically `.GlobalEnv`, representing the working environment. This permits the masking of some variables or functions that are in scope, but are found further down the search path.

```
pi

## [1] 3.142

pi = 3  # redefines variable pi
pi  # .GlobalEnv match

## [1] 3

rm(pi)  # removes masking variables
pi

## [1] 3.142
```

[3]The number of available packages is over 1000 at the time of this writing; see Figure ??.

The function sink(X) records the output to the file X instead of the display, useful for creating a log-file for later examination. To print the output both to screen and to a file, use the following variation.

```
sink(file = "outputFile", split = TRUE)
```

The examples above show how to run R interactively. To execute R code written in a text file foo.R (.R is the conventional file name suffix for R code) use either

- the command source("foo.R") within R,
- the command R CMD BATCH foo.R from a Linux or Mac command line,
- the command Rscript foo.R within a Linux or Mac command line,
- a command line executable script file, with appropriate permissions, with

  #!/usr/bin/Rscript

  as the top line.

The last three options permit passing parameters to the script, for example using R CMD BATCH --args arg1 arg2 foo.R or Rscript foo.R arg1 arg2. To retrieve the arguments inside the script, use the function call args=commandArgs(TRU to return the arguments as a list of strings (see below on how to handle lists in R). The fourth option above has the advantage of allowing Linux style input and output redirecting via foo.R < inFile > outFile and other shell tricks.

4.2 Scalar Data Types

Variable categories in R include scalars and collections. Scalar types include numeric, integer, logical, string, dates, and factors. Numeric and integer variables represent real numbers and integers, respectively. A logical or binary variable is a single bit whose value in R is TRUE or FALSE. Strings are ordered sequences of characters. Dates represent calendar dates. Factor variables represent values from an ordered or unordered finite set. Some operations can trigger casting between the various types. Functions such as as.numeric can perform explicit casting.

```
a = 3.2
b = 3  # both are numeric types
b

## [1] 3

typeof(b)  # function returns type of object

## [1] "double"
```

```
c = as.integer(b)  # cast to integer type
c

## [1] 3

typeof(c)

## [1] "integer"

c = 3L  # alternative to casting: L specifies integer
d = TRUE
d

## [1] TRUE

e = as.numeric(d)  # casting to numeric
e

## [1] 1

f = "this is a string"  # string
f

## [1] "this is a string"

ls.str()  # show variables and their types

## a :  num 3.2
## b :  num 3
## c :  int 3
## d :  logi TRUE
## e :  num 1
## f :  chr "this is a string"
```

Here are some commands illustrating ordered and unordered factors.

```
current.season = factor("summer", levels = c("summer",
    "fall", "winter", "spring"), ordered = TRUE)
current.season  # ordered factor

## [1] summer
## 4 Levels: summer < fall < ... < spring

levels(current.season)  # display the levels of g

## [1] "summer" "fall"   "winter" "spring"

my.eye.color = factor("brown", levels = c("brown",
    "blue", "green"), ordered = FALSE)
my.eye.color

## [1] brown
## Levels: brown blue green
```

4.3 Vectors, Arrays, Lists, and Dataframes

Vectors, arrays, lists and dataframes are collections of scalar or other collection-type variables. A vector is a one-dimensional ordered collection of variables of the same type. An array is a multidimensional generalization of vectors of which a matrix is a two-dimensional special case. Lists are ordered collections of variables of potentially different types. The list signature is the ordered list of variable types from the list. A dataframe is an ordered collection of lists of the same signature.

In the case of multidimensional arrays, the indices appear inside square brackets, where missing indices indicate a full selection of that dimension. A minus sign indicates selection of all but the specified values. Below are some examples of handling vectors and arrays.

```
x = c(4, 3, 3, 4, 3, 1)  # c() concatenates arguments to create
a vector
x

## [1] 4 3 3 4 3 1

length(x)

## [1] 6

2 * x + 1  # element-wise arithmetic

## [1] 9 7 7 9 7 3

# Boolean vector (default is FALSE)
y = vector(mode = "logical", length = 4)
y

## [1] FALSE FALSE FALSE FALSE

# numeric vector (default is 0)
z = vector(length = 3, mode = "numeric")
z

## [1] 0 0 0

q = rep(3.2, times = 10)  # repeat value multiple times
q

##  [1] 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2 3.2

w = seq(0, 1, by = 0.1)  # values in [0,1] in 0.1 increments
w

##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
## [11] 1.0
```

```
# 11 evenly spaced numbers between 0 and 1
w = seq(0, 1, length.out = 11)
w

##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
## [11] 1.0

any(w <= 0.5)  # is it true for some elements?

## [1] TRUE

all(w <= 0.5)  # is it true for all elements?

## [1] FALSE

which(w <= 0.5)  # for which elements is it true?

## [1] 1 2 3 4 5 6

# create an array with TRUE/FALSE reflecting
# whether condition holds
w <= 0.5

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [7] FALSE FALSE FALSE FALSE FALSE

r = (w <= 0.5)  # Boolean vector showing TRUE where w<=1/2
r

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [7] FALSE FALSE FALSE FALSE FALSE

w[w <= 0.5]  # extracting from w entries for which w<=0.5

## [1] 0.0 0.1 0.2 0.3 0.4 0.5

subset(w, w <= 0.5)  # an alternative with the subset function

## [1] 0.0 0.1 0.2 0.3 0.4 0.5

w[w <= 0.5] = 0  # zero out all components smaller or equal to
0.5
w

##  [1] 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.7 0.8 0.9
## [11] 1.0
```

Arrays are multidimensional generalizations of vectors; the `dim` attribute specifies the dimension. Matrices are simply two dimensional arrays.

```
z = seq(1, 20, length.out = 20)
x = array(data = z, dim = c(4, 5))
x

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20

x[2, 3]  # refer to the second row and third column

## [1] 10

x[2, ]  # refer to the entire second row

## [1]  2  6 10 14 18

x[-1, ]  # all but the first row - same as x[c(2,3,4),]

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    2    6   10   14   18
## [2,]    3    7   11   15   19
## [3,]    4    8   12   16   20

y = x[c(1, 2), c(1, 2)]  # 2x2 top left submatrix
2 * y + 1  # note element-wise operation

##      [,1] [,2]
## [1,]    3   11
## [2,]    5   13

y %*% y  # matrix product (both arguments are matrices)

##      [,1] [,2]
## [1,]   11   35
## [2,]   14   46

x[1, ] %*% x[1, ]  # inner product (both vectors have the same
dimensions)

##      [,1]
## [1,]  565

t(x)  # matrix transpose

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
## [4,]   13   14   15   16
## [5,]   17   18   19   20
```

```
outer(x[, 1], x[, 1])  # outer product

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    4    6    8
## [3,]    3    6    9   12
## [4,]    4    8   12   16

rbind(x[1, ], x[1, ])  # vertical concatenation

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    1    5    9   13   17

cbind(x[1, ], x[1, ])  # horizontal concatenation

##      [,1] [,2]
## [1,]    1    1
## [2,]    5    5
## [3,]    9    9
## [4,]   13   13
## [5,]   17   17
```

Lists are ordered collections which permit positions to hold variables of different types. To ease the tracking of the semantics of variables in the list, we typically assign names to the various positions within the list. Regarding element access, if `L` is a list, `L[i]` is a list containing the i element and `L[[i]]` is the i element.

```
L = list(name = "John", age = 55, manager = FALSE,
    no.children = 2, children.ages = c(15, 18))
names(L)  # displays all position names

## [1] "name"          "age"
## [3] "manager"       "no.children"
## [5] "children.ages"

L[2]  # list containing first element - displays name and value

## $age
## [1] 55

L[[2]]  # value held in second position

## [1] 55

L$name  # value in list corresponding to name (first position)

## [1] "John"
```

```
L["name"]  # same thing

## $name
## [1] "John"

L$children.ages[2]  # same as L[[5]][2]

## [1] 18
```

A dataframe is an ordered sequence of lists sharing the same signature. A dataframe often serves as a table whose rows correspond to data examples (samples from a multivariate distribution) and whose columns correspond to dimensions or features.

```
vecn = c("John Smith", "Jane Doe")
veca = c(42, 45)
vecs = c(50000, 55000)
# create a data frame
R = data.frame(name = vecn, age = veca, salary = vecs)
R

##          name age salary
## 1 John Smith  42  50000
## 2   Jane Doe  45  55000

# modify column names
names(R) = c("NAME", "AGE", "SALARY")
R

##          NAME AGE SALARY
## 1 John Smith  42  50000
## 2   Jane Doe  45  55000
```

The core R package `datasets` contains many interesting and demonstrative datasets, such as the `iris` dataset, whose first four dimensions are numeric measurements describing flower geometry, and whose last dimension is a string describing the flower species.

```
names(iris)  # lists the dimension (column) names

## [1] "Sepal.Length" "Sepal.Width"
## [3] "Petal.Length" "Petal.Width"
## [5] "Species"

head(iris, 4)  # show first four rows

##   Sepal.Length Sepal.Width Petal.Length
## 1          5.1         3.5          1.4
## 2          4.9         3.0          1.4
## 3          4.7         3.2          1.3
## 4          4.6         3.1          1.5
##   Petal.Width Species
## 1         0.2  setosa
## 2         0.2  setosa
## 3         0.2  setosa
## 4         0.2  setosa
```

```
iris[1, ]  # first row
```

```
##   Sepal.Length Sepal.Width Petal.Length
## 1          5.1         3.5          1.4
##   Petal.Width Species
## 1         0.2  setosa
```

```
iris$Sepal.Length[1:10]  # sepal length of first ten samples
```

```
##  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9
```

```
# allows replacing iris$Sepal.Length with
# the shorter Sepal.Length
attach(iris, warn.conflicts = FALSE)
mean(Sepal.Length)  # average of Sepal.Length across all rows
```

```
## [1] 5.843
```

```
colMeans(iris[, 1:4])  # means of all four numeric columns
```

```
## Sepal.Length  Sepal.Width Petal.Length
##        5.843        3.057        3.758
##  Petal.Width
##        1.199
```

The subset function is useful for extracting subsets of a dataframe.

```
# extract all rows whose Sepal.Length
# variable is less than 5 and whose species
# is setosa
subset(iris, Sepal.Length < 5 & Species != "setosa")
```

```
##     Sepal.Length Sepal.Width Petal.Length
## 58           4.9         2.4          3.3
## 107          4.9         2.5          4.5
##     Petal.Width    Species
## 58          1.0 versicolor
## 107         1.7  virginica
```

```
# count number of rows corresponding to
# setosa species
dim(subset(iris, Species == "setosa"))[1]
```

```
## [1] 50
```

The function summary provides a useful statistical summary of the different dataframe columns. R automatically determines whether the variables are numeric, such as Sepal.Length, or factors, such as Species. For numeric variables, the summary function displays the minimum, maximum, mean, median, and the 25% and 75% percentiles. For factor variables, the summary function displays the number of dataframe rows in each of the factor levels.

```
summary(iris)
```

```
##   Sepal.Length   Sepal.Width
##  Min.   :4.30   Min.   :2.00
##  1st Qu.:5.10   1st Qu.:2.80
##  Median :5.80   Median :3.00
##  Mean   :5.84   Mean   :3.06
##  3rd Qu.:6.40   3rd Qu.:3.30
##  Max.   :7.90   Max.   :4.40
##   Petal.Length   Petal.Width
##  Min.   :1.00   Min.   :0.1
##  1st Qu.:1.60   1st Qu.:0.3
##  Median :4.35   Median :1.3
##  Mean   :3.76   Mean   :1.2
##  3rd Qu.:5.10   3rd Qu.:1.8
##  Max.   :6.90   Max.   :2.5
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
```

With appropriate formatting, we can create a dataframe using a text file. We can load the text

```
Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
         5.1         3.5          1.4         0.2     setosa
         4.9         3.0          1.4         0.2     setosa
```

into a dataframe in R using the read.table function.

```
# from a text file in the local directory
Iris = read.table("irisFile.txt", header = TRUE)
# or else from a text file on the internet
Iris = read.table("http://www.exampleURL.com/irisFile.txt",
    header = TRUE)
```

We can examine and edit dataframes and other variables within a spreadsheet-like environment using the edit function.

```
edit(iris)  # examine data as spreadsheet
iris = edit(iris)  # edit dataframe/variable
newIris = edit(iris)  # edit dataframe/variable but keep
original
```

4.4 If-Else, Loops, and Functions

The flow of control of R code is very similar to that of other programming languages. Below are some examples of if-else, loops, function definitions and function calls.

```
a = 10
b = 5
c = 1
if (a < b) {
    d = 1
} else if (a == b) {
    d = 2
} else {
    d = 3
}
d

## [1] 3
```

The logical operators in R are similar to those in C, C++, and Java. Examples include `&&` for AND, `||` for OR, `==` for equality, and `!=` for inequality. For-loops repeat for a pre-specified number of times, with each loop assigning a different component of a vector to the iteration variable. Repeat-loops repeat until a `break` statement. While-loops repeat until a `break` statement or until the loop condition is not satisfied.

```
sm = 0  # repeat for 100 iteration, with num taking values 1:100
for (num in seq(1, 100, by = 1)) {
    sm = sm + num
}
sm  # same as sum(1:100)

## [1] 5050

repeat {
    sm = sm - num
    num = num - 1
    if (sm == 0)
        break  # if sm==0 then stop the loop
}
sm
```

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

```
a = 1
b = 10
# continue the loop as long as b>a
while (b > a) {
    sm = sm + 1
    a = a + 1
    b = b - 1
}
sm
```

```
## [1] 5
```

Functions in R are similar to those in C, C++, Java, and Matlab. When calling a function the arguments flow into the parameters according to their order at the call site. Alternatively, arguments can appear out of order if the caller supplies parameter names.

```
# parameter bindings by order
foo(10, 20, 30)
# out of order parameter bindings
foo(y = 20, x = 10, z = 30)
```

Omitting an argument assigns the default value of the corresponding parameter.

```
# passing 3 parameters
foo(x = 10, y = 20, z = 30)
# x and y are missing and are assigned
# default values
foo(z = 30)
# in-order parameter binding with last two
# parameters missing
foo(10)
```

Out of order parameter bindings and default values simplify calling functions with long lists of parameters, when many of those parameters take default values.

```
# The following function raises the first
# argument to the power of the second. The
# first argument is named bas and has
# default value 10.  The second parameter is
# named pow and has default value 2.
myPower = function(bas = 10, pow = 2) {
    res = bas^pow  # raise base to a power
    return(res)
}
myPower(2, 3)  # 2 is bound to bas and 3 to pow (in-order)
```

```
## [1] 8

# same binding as above (out-of-order
# parameter names)
myPower(pow = 3, bas = 2)

## [1] 8

myPower(bas = 3)  # default value of pow is used

## [1] 9
```

As in other languages, R passes variables by value; as a consequence, changing the passed arguments inside the function does not modify their respective values in the calling environment. Variables defined inside functions are local, and thus are unavailable after the function completes its execution. The returned value is the last computed variable or the one specified in a `return` function call. Returning a list or dataframe permits returning multiple values.

```
x = 2
myPower2 = function(x) {
    x = x^2
    return(x)
}
y = myPower2(x)  # does not change the value of x outside the
function
x

## [1] 2

y

## [1] 4
```

It is best to avoid loops when programming in R. There are two reasons for this: simplifying code and computational speed-up. Many mathematical computations on lists, vectors, or arrays may be performed without loops using component-wise arithmetic.

```
a = 1:10
# compute sum of squares using a for loops
c = 0
for (e in a) c = c + e^2
c

## [1] 385

# same operation using vector arithmetic
sum(a^2)
```

```
## [1] 385

# time comparison with a million elements
a = 1:1e+06
c = 0
system.time(for (e in a) c = c + e^2)

##    user  system elapsed
##   0.779   0.004   0.783

system.time(sum(a^2))

##    user  system elapsed
##   0.005   0.002   0.007
```

Another way to avoid loops is to use the function `sapply`, which applies a function passed as a second argument to the list, data-frame, or vector, passed as a first argument. This leads to simplified code, though the computational speed-up may not apply in the same way as it did above.

```
a = seq(0, 1, length.out = 10)
b = 0
c = 0
for (e in a) {
    b = b + exp(e)
}
b

## [1] 17.34

c = sum(sapply(a, exp))
c

## [1] 17.34

# sapply with a user defined function
# f(x)=exp(x^2)
sum(sapply(a, function(x) {
    return(exp(x^2))
}))

## [1] 15.07

# or more simply
sum(sapply(a, function(x) exp(x^2)))

## [1] 15.07
```

4.5 Debugging and Profiling

Core R features some debugging and profiling functionality. Additional debugging and profiling functionality is available through third party packages, as described at the end of this section.

We can use the `print` and `cat` functions to display values of relevant variables at appropriate positions to debug code. The `print` function displays the value of a variable, while the `cat` function displays the concatenation of values of several variables passed as parameters.

The `browser` function suspends the execution of the R code, providing a prompt through which the programmer can evaluate variables or execute new R code for debugging. Other options include stepping through line by line (by typing `n`), continuing execution (by typing `c`), or halting execution (by typing `Q`).

Listing 4.1: debugging with the browser function (R code)

```r
foo2=function(i) {a=i+1;b=a+1;browser();a=b+1;return(b)}
> foo2(3)
Called from: foo2(3)
Browse[1]> a  # display the value of a
[1] 4
Browse[1]> b  # display the value of b
[1] 5
Browse[1]> n  # executes next command
debug: a = b + 1
Browse[2]> c  # continue execution
[1] 5
```

The function `debug` binds the `browser` function to a user-specified function, invoking a debug session at each call site of said function. The `undebug` cancels the binding.

Listing 4.2: debugging with the debug function (R code)

```r
foo3=function(i) {a=i+1;b=a+1;a=b+1;return(b)}
> debug(foo3)
> foo3(1)
debugging in: foo3(1)
debug: {
    a = i + 1
    b = a + 1
    a = b + 1
    return(b)
}
Browse[2]> n  # execute first command
debug: a = i + 1
Browse[2]> n  # execute next command
debug: b = a + 1
```

```
Browse[2]> a  # display value of a
[1] 2
Browse[2]> p=2*a+2  # define a new variable
Browse[2]> p
[1] 6
Browse[2]> n  # executes next command
debug: a = b + 1
Browse[2]> b
[1] 3
Browse[2]> c  # continue execution
exiting from: foo3(1)
[1] 3
> undebug(foo3)
```

The `trace` function accepts the name of a function as a parameter and prints the function call with the passed arguments each time the function executes. If an error occurs, the variable `.Traceback`, a character array containing the currently active function calls, appears in the workspace.

Additional debugging functionality is available from contributed packages. In particular, the `debug` package **?** provides capabilities similar to those of standard graphical debuggers, including a graphical window showing the current command, setting conditional breakpoints, and the opportunity to continue debugging after an error. Another package is `edtdbg`, which integrates R debugging with external text editors such as vim.

Profiling refers to diagnosing which parts of the code are responsible for heavy computational efforts and memory usage. The function `Rprof(X)` starts a profiling session and saves the information to the file `X`. A call `Rprof(NULL)` indicates the end of the profiling session and `summaryRprof` shows the profiling summary.

```
# start profiling
Rprof("diagonsisFile.out")
A = runif(1000)  # generate a vector of random numbers in [0,1]
B = runif(1000)  # generate another random vector
C = eigen(outer(A, B))
Rprof(NULL)  # end profiling
summaryRprof("diagonsisFile.out")

## $by.self
##                       self.time self.pct
## "eigen"                    3.52    85.02
## "all.equal.numeric"        0.36     8.70
## "as.vector"                0.16     3.86
## "t.default"                0.08     1.93
## "mean"                     0.02     0.48
##                       total.time total.pct
## "eigen"                     4.14    100.00
## "all.equal.numeric"         0.62     14.98
## "as.vector"                 0.16      3.86
## "t.default"              0.08      1.93
## "mean"                   0.02      0.48
##
## $by.total
##                       total.time total.pct
## "eigen"                     4.14    100.00
## "<Anonymous>"               4.14    100.00
## "block_exec"                4.14    100.00
## "call_block"                4.14    100.00
## "doTryCatch"                4.14    100.00
## "eval.parent"               4.14    100.00
## "eval"                      4.14    100.00
## "evaluate"                  4.14    100.00
## "FUN"                       4.14    100.00
## "knit_fun"                  4.14    100.00
## "knit"                      4.14    100.00
## "lapply"                    4.14    100.00
## "local"                     4.14    100.00
## "mapply"                    4.14    100.00
## "process_file"              4.14    100.00
## "process_group.block"       4.14    100.00
## "try"                       4.14    100.00
## "tryCatch"                  4.14    100.00
## "tryCatchList"              4.14    100.00
## "tryCatchOne"               4.14    100.00
## "unlist"                    4.14    100.00
## "withCallingHandlers"       4.14    100.00
## "withVisible"               4.14    100.00
## "all.equal.numeric"         0.62     14.98
## "all.equal"                 0.62     14.98
## "isSymmetric.matrix"        0.62     14.98
## "as.vector"                 0.16      3.86
## "t.default"                 0.08      1.93
## "attr.all.equal"            0.08      1.93
## "mode"                      0.08      1.93
## "t"                         0.08      1.93
## "mean"                      0.02      0.48
##                       self.time self.pct
## "eigen"                    3.52    85.02
## "<Anonymous>"              0.00     0.00
## "block_exec"               0.00     0.00
## "call_block"               0.00     0.00
## "doTryCatch"               0.00     0.00
## "eval.parent"              0.00     0.00
## "eval"                     0.00     0.00
## "evaluate"                 0.00     0.00
## "FUN"                      0.00     0.00
## "knit_fun"                 0.00     0.00
## "knit"                     0.00     0.00
## "lapply"                    0.00     0.00
## "local"                     0.00     0.00
## "mapply"                    0.00     0.00
## "process_file"              0.00     0.00
## "process_group.block"       0.00     0.00
## "try"                       0.00     0.00
## "tryCatch"                  0.00     0.00
## "tryCatchList"              0.00     0.00
## "tryCatchOne"               0.00     0.00
## "unlist"                    0.00     0.00
## "withCallingHandlers"       0.00     0.00
## "withVisible"               0.00     0.00
## "all.equal.numeric"         0.36     8.70
## "all.equal"                 0.00     0.00
## "isSymmetric.matrix"        0.00     0.00
## "as.vector"                 0.16     3.86
## "t.default"                 0.08     1.93
## "attr.all.equal"            0.00     0.00
## "mode"                      0.00     0.00
## "t"                         0.00     0.00
## "mean"                      0.02     0.48
##
## $sample.interval
## [1] 0.02
##
## $sampling.time
## [1] 4.14
```

The second report displays execution time within each function. Total time is time spent in the function and its respective call subgraph; self time is the time spent only in the function itself. Notice, though that the function `eigen` is responsible for nearly 100% of the total time, most of the execution time occurs within other functions called from within `eigen` (the `.Call` responsible for most of the self time represents compiled C code — see the next section). This is clearly the computational bottleneck, as the outer product and the generation of random samples are not responsible for any substantial total time.

See http://cran.r-project.org/doc/manuals/R-exts.pdf for more information on `Rprof` and a discussion on profiling memory usage. The packages `proftools` and `profr` provide additional profiling functionality including analyzing the call graphs.

4.6 Interfacing with C

R is inherently an interpreted language; that is, R compiles each command at run time, resulting in many costly context switches and difficulty in applying standard compiler optimization techniques. Thus, R programs likely will not execute as efficiently as compiled programs in C, C++, or Fortran.

The computational slowdown described above typically increases with the number of elementary function calls or commands. For example, R code that randomly samples two matrices, multiplies them, then computes eigenvalues typically will not suffer a significant slowdown compared to similar implementations in C, C++, or Fortran. On the other hand, R code containing many nested loops is likely to be substantially slower.

For example, consider the code below, which compares two implementations of matrix multiplication. The first uses R's internal matrix multiplication and the second implements it through three nested loops, each containing a scalar multiplication.

```r
n = 100
nsq = n * n
# generate two random matrices
A = matrix(runif(nsq), nrow = n, ncol = n)
B = matrix(runif(nsq), nrow = n, ncol = n)
system.time(A %*% B)  # built-in matrix multiplication

##    user  system elapsed
##   0.001   0.000   0.001

matMult = function(A, B, n) {
    R = matrix(data = 0, nrow = n, ncol = n)
    for (i in 1:n) for (j in 1:n) for (k in 1:n) R[i,
        j] = R[i, j] + A[i, k] * B[k, j]
    return(R)
}
# nested loops implementation
system.time(matMult(A, B, n))

##    user  system elapsed
##   2.941   0.001   2.942
```

The first matrix multiplication is faster by several orders of magnitude even for a relatively small $n = 100$. In fact, the first may be very similar to the second implementation with three nested loops; the key difference is that the built-in matrix multiplication runs compiled C code.

Clearly, it is better, if possible, to write R code containing relatively few loops and few elementary R functions. Since core R contains a rich library of elementary functions, one can often follow this strategy. In some cases, however, this vectorization approach is not possible. A useful heuristic is to identify computational bottlenecks using a profiler, then re-implement the offending R code in a compiled language such as C or C++. The remaining R code will interface with the reimplemented bottleneck code via an external interface. Assuming that a small percent of the code is responsible for most of the computational inefficiency (as is often the case), this strategy can produce substantial speedups with little effort.

The Writing R Extensions manual available at http://cran.r-project.org/doc/manuals/R-exts.pdf describes the `.C` function for interfacing compiled C code from within R. The manual also describes the `.Call` interface (which provides additional functionality), and interfaces for C++ and Fortran. We focus in this section on the relatively simple `.C` interface.

We assume that the reader is familiar with C programming. The C code implementing the computational bottleneck resides in a function accepting multiple pointers and returning `void`. The pointers in the input arguments point to memory containing arrays that the R code passes to the C code or arrays returned by the C code. The command `R CMD SHLIB filename.c` compiles the C code and creates a file `filename.so`. The function `dynload` loads that file into the current R session. Finally, the function `.C` invokes C functions within the shared object, returning a list of arrays containing the memory at the locations pointed to by the pointer arguments.

For example, consider the task of computing $c^n_{i=1} = \sum_{j=1}^n (a_j + i)^{b_j}$ for all $i, j = 1, \ldots, n$ given two vectors a and b of size n. The C code to compute the result appears below. The first two pointers point to arrays containing the vectors a and b, the third pointer points to the length of the arrays, and the last pointer points to the area where the results should appear. Note the presence of the pre-processor directive `include<R.h>`.

Listing 4.3: computing array coefficients (C code)

```c
#include <R.h>
#include <math.h>

void fooC(double* a, double* b, int* n, double* res)
{
  int i,j;
  for (i=0;i<(*n);i++) {
    res[i]=0;
    for (j=0;j<(*n);j++)
      res[i] += pow(a[j]+i+1,b[j]);
  }
}
```

If we save this code as a file named `fooC.c`, we can compile using the following command entered in the Linux or Mac prompt (this command calls the gcc or cc compiler):

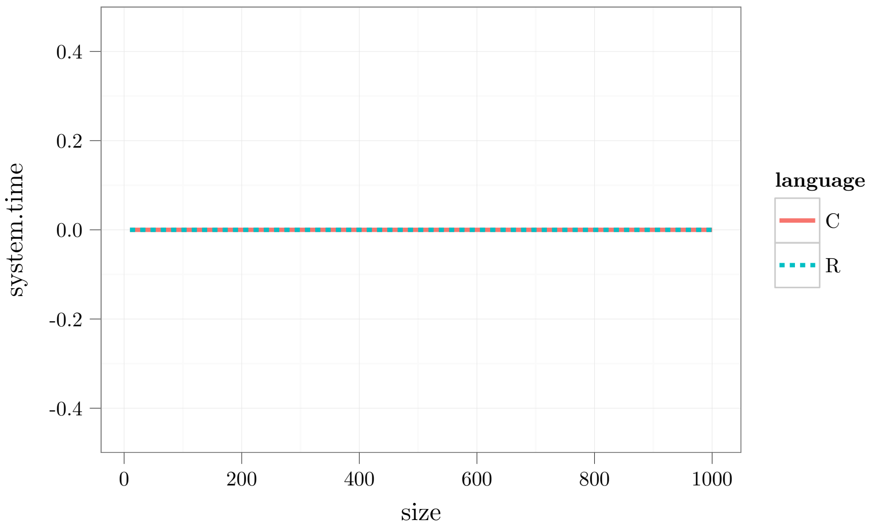
```
R CMD SHLIB fooC.c
```

The compilation creates a file `fooC.so`. Placing `fooC.so` in the same directory as the R code, we can dynamically load the shared object with `dynload`. Note that below, the result expectedly appears in the fourth element of the list returned by the `.C` function.

We display below a graph contrasting the run-time for both implementations as a function of the size of the arrays n. The R implementation is much slower for large n than the .C implementation that interfaces compiled C code from R. We include the code below that generates the graph, though the `qplot` function will only be described in the next chapter.

```
dyn.load("fooC.so")  # load the compiled C code

## Error:  unable to load shared object '/home/lebanon/TAODv2/Rnw/fooC.so':
## dlopen(/home/lebanon/TAODv2/Rnw/fooC.so, 6):  image not found

n = 10
A = seq(0, 1, length = n)
B = seq(0, 1, length = n)
C = rep(0, times = n)
L = .C("fooC", A, B, as.integer(n), C)

## Error:  C symbol name "fooC" not in load table

ResC = L[[4]]  # extract 4th list element containing result
ResC

## [1] 2


# same in R
fooR = function(A, B, n) {
    res = rep(0, times = n)
    for (i in 1:n) {
        for (j in 1:n) res[i] = res[i] + (A[j] +
            i)^(B[j])
    }
    return(res)
}
ResR = fooR(A, B, n)
ResR

##  [1] 13.34 17.48 21.21 24.71 28.03 31.24
##  [7] 34.34 37.37 40.33 43.24

k = 10
sizes = seq(10, 1000, length = k)
Rtime = rep(0, k)
Ctime = Rtime
i = 1
for (n in sizes) {
    A = seq(0, 1, length = n)
    B = seq(0, 1, length = n)
    C = rep(0, times = n)
    Ctime[i] = system.time(.C("fooC", A, B, as.integer(n),
        C))
    Rtime[i] = system.time(fooR(A, B, n))
    i = i + 1
}
```

```
## Error:  C symbol name "fooC" not in load table

## Timing stopped at: 0 0 0.001

DF = stack(list(C = Ctime, R = Rtime))
names(DF) = c("system.time", "language")
DF$size = sizes
print(qplot(x = size, y = system.time, lty = language,
    color = language, data = DF, size = I(1.5),
    geom = "line"))
```



4.7 Customization and Other Topics

When R starts it executes the function `.First()` in the .Rprofile file in the home directory (if it exists). This is a good place to put user preferred options. Similarly, R executes the function `.Last` in the same file at the end of any R session. The function `options` adjust the behavior of R in many ways, for example to include more digits when displaying the value of a numeric variable.

```
.First = function() {
    options(prompt = "R >", digits = 6)
    library("ggplot2")
}
.Last = function() {
    cat(date(), "\nBye\n")
}
```

In Linux or Mac, we can execute R with flags; for example, `R -q` starts R without printing the initial welcome message.

We conclude with the following comments:

- R is a functional language in that all statements are interpreted as functions. Thus, rather than using `return a` in C++ or `help X` in Matlab, R invokes `return(a)` and `help(X)`. Furthermore, even operators are actually function calls. For example, `a=b` converts to the function call `'='(a,b)`, and `a[3]` converts to the function call `'['(a,3)` (try it out!).

- R is an object-oriented language; users can define new objects and modify the way in which common functions operate on them.

- R supports lazy evaluation; that is, R computes values as late as possible in execution. For example, R copies large arrays only when needed and performs complex computations only when the result becomes necessary.

- The value `NA`, meaning Not Available, denotes missing values. When designing data analysis functions, one should handle `NA` values explicitly. Many functions feature the argument `na.rm` which, if `TRUE`, operates on the data after removing any `NA` values.

- When using arithmetic operations between arrays of different sizes, the smaller array is extended as needed, with new elements created by recycling old ones. Similarly, storing a value in a non-existing element expands the array as needed, padding with `NA` values.

```
a = c(1, 2)
b = c(10, 20, 30, 40, 50)
a + b

## [1] 11 22 31 42 51

b[7] = 70
b

## [1] 10 20 30 40 50 NA 70
```

- We can access multidimensional array elements using a single index. The single index counts elements by traversing the array by columns, then rows, then other dimensions where appropriate.

```
A = matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
A

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

```
A[3]  # counting by columns A[3]=A[1,2]
```

```
## [1] 3
```

- Since the R dollar operator is similar to the period operator in many other languages, periods can appear in variable names. Thus, the period can delimit words within a variable name; for example, `my.parallel.clustering` is a legitimate name for a function or a dataframe variable.

4.8 Notes

R programming books include free resources such as the official introduction to R manual http://cran.r-project.org/doc/manuals/R-intro.html (replace html with pdf for pdf version) and the language reference http://cran.r-project.org/doc/manuals/R-lang.html (replace html with pdf for pdf version). Additional manuals on writing R extensions, importing data, and other topics are available at http://cran.r-project.org/doc/manuals/. Many additional books are available via commercial publishers.

R packages are available from http://cran.r-project.org. Each package features a manual in a common documentation format; many packages feature additional tutorial documents known as vignettes. Navigating the increasing repository of packages can be overwhelming. The Task Views help to aggregate lists of packages within a particular task or area. The list of Task Views is available at http://cran.r-project.org/web/views/. Another useful tool is http://crantastic.org which shows recently contributed or updated packages, user reviews, and ratings. Finally, the freely available R-Journal at http://journal.r-project.org/ contains high quality refereed articles on the R language, including many descriptions of contributed packages.

4.9 Exercises

1. Type the R code in this chapter into an R session and observe the results.
2. Implement a function that computes the log of the factorial value of an integer using a for loop. Note that implementing it using $\log(A)+\log(B)+\cdots$ avoids overflow while implementing it as $\log(A \cdot B \cdot \cdots)$ creates an overflow early on.
3. Implement a function that computes the log of the factorial value of an integer using recursion.
4. Using your two implementations of log-factorial in (2) and (3) above, compute the sum of the log-factorials of the integers $1, 2, \ldots, N$ for various N values.

5. Compare the execution times of your two implementations for (4) with an implementation based on the official R function `lfactorial`. You may use the function `system.time` to measure execution time. What are the growth rates of the three implementations as N increases? Use the command `options(expressions=500000)` to increase the number of nested recursions allowed. Compare the timing of the recursion implementation as much as possible, and continue beyond that for the other two implementations.

Chapter 5

Graphing Data with R

Chapter 6

Preprocessing Data