Introduction to computational programming
Using *R*

David M. Rosenberg
University of Chicago
Committee on Neurobiology

Version control information:
Last changed date: 2009-09-15 18:34:08 -0500 (Tue, 15 Sep 2009)
Last changes revision:24
Version: Revision 24
Last changed by: root

September 15, 2009

Overview

This exercise is designed to serve as a practical introduction to the computational tools that will be used throughout this course. It assumes no previous knowledge of numerical analysis nor experience in computer programming.

In order to help distinguish between *code*, example output, computer commands and textual information, the following conventions will be used (both here and in later computational exercises).

R input

Commands to be entered into the *R* interpreter will be presented in *syntax-highlighted* typewriter font, with the ">" character marking the beginning of each line. Here is an example:

```
> 3 + 5
> help.start()
> load('myData.RData')
```

R output

Output from the *R* interpreter when shown, will be displayed directly after the corresponding input lines using the same font but in a different color and without the leading ">".

```
> 3 + 5
```

```
[1] 8
```

```
> randomData <- rnorm(n=100)
> summary(randomData)
```

```
      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-2.53900 -0.90210 -0.09291 -0.16160  0.62420  2.04800
```

Computer commands / keyboard keys

Following standard conventions, keyboard commands/shortcuts will be printed inline with the text in black typewriter font. Combinations of keys which must be pressed simultaneously are separated by hyphens. Keys to be pressed sequentially are separated by spaces. "Modifier keys" (which vary in name from keyboard to keyboard) are denoted using a capital `C` or `M`. Keyboard notation is summarized below:

- *Control:* Typically the "control" key abbreviated as `C-`
- *Meta:* Usually the "alt" on standard keyboards and the "command" on apple keyboards, abbreviated as `M-`
- *Enter:* Variously termed "enter", "return", "carriage return", "linefeed", and "newline", abbreviated as `[CR]`
- *Directional arrows:* the arrow keys are represented by `[LEFT]`, `[RIGHT]`, `[UP]`, and `[DOWN]` respectively.
- *Other keys:* Other keys are represented similarly, such as `[Esc]`, `[F1]` and `[TAB]`.

For example `C-c` means to simultaneously press the "Control key" and the letter "c". `C-x C-c` means to first simultaneously press the "Control" key and the letter "x", then to simultaneously press the "Control" key and the letter "c", and `[Esc] : q !` means to sequentially press "escape", the "colon" (requires `[shift]`), "q" and the exclamation point (requires `[shift]`).

Make sure to pay special to similar looking characters such as

- Single- (´), double- (¨) and "back-" (`) quotes
- Parentheses (( )), brackets ([ ]) and braces ({ })

Graphical menus navigation is represented by placing boxes around menu and button names, such as `File` - `Quit`.

Source text

Large sections of source code and file contents will be displayed similarly to *R* code with the following exceptions.

1. The select will be surrounded by a box.
2. No prompts will be displayed (see section 2.1.5)

3. A header comment (see section 4.2) will give the name of the file, URL to download it and other metadata

Here is an example *R* source file, defining a function needed in Exercise 2 (see II).

```
#!/usr/bin/env rr
# encoding: utf-8
# sumDigits.R
#
# sumDigits - a function which takes as input a number and returns the
#             sum of its digits
#
# Example:
#       > sumDigits(15)
#       [1] 6
#       > sumDigits(c(10, 122, 134))
#       [1] 1 5 8
#
sumDigits <- function(x) {
  return(sum(as.integer(strsplit(as.character(x), '')[[1]])))
}
```

Part I

Tutorial

1 Getting Started

While not strictly necessary, many students find it helpful to have access to *R* and associated tools on their own computers. Fortunately, *R* is *free software*[1], and available for most computing platforms.

1.1 GNU *R*

The R homepage `http://r-project.org` provides compiled binaries for Windows, OS X, and linux platforms as well as the source distributions (for other platforms). The following are platform specific installation instructions for the most common scenarios.

[1] By calling *R free software*, we are saying both that:

1. You don't have to pay to use *R* (free as in beer)
2. You are free to examine and improve *R* as you like (free as in speech)

.

1.1.1 Mac OSX

The Mac OSX binary distribution of *R* can be downloaded from `http://streaming.stat.iastate.edu/CRAN/bin/macosx/` as a `.dmg` file. After downloading the image, simply open the `.dmg` file and drag the `R.app` icon into your `Applications` folder.

Once you have done this, starting *R* is as easy as double-clicking the `R.app` icon in your `Applications` folder. Alternatively, you may run *R* in a console window by opening `Terminal.app` (located in the `Utilities` subfolder of `Applications`) and typing `R`[2].

Running `R.app` provides you with some additional GUI functionality, provided through the menu interface, such as a *R* source editor (`File` - `New Document`), a package installer (`Packages & Data` - `Package Installer`) and easy access to package guides (`Help` - `Vignettes`).

1.1.2 Windows

Installing *R* under windows is accomplished by downloading the windows binary installer from `http://streaming.stat.iastate.edu/CRAN/bin/windows/base/`, opening the installer and following the on-screen directions. Upon completion of the installer (and possibly rebooting), you should have an icon labelled `R 2.9.2` on your desktop (and possibly in the `Start` menu as well).

To start a new *R* session, simply double-click on the `R 2.9.2` icon.

1.1.3 Linux

Installing *R* on a linux system can generally be performed using your distribution-specific package manager (`rpm/yum` for RedHat-type distributions, `apt` for Debian based distributions such as Ubuntu).

If your distribution does not provide *R* packages, you can download the compressed sources from `http://streaming.stat.iastate.edu/CRAN/bin/linux/ubuntu/` and compile them yourself[3].

TODO: Dependency issues

1.1.4 Other options

Should none of the above options prove successful for you, alternative methods of running *R* do exist.

TODO: other methods

- Java i.e. Biocep
- remote (ssh)

[2] If you have trouble with this, it may be due to having the default `PATH` set incorrectly. See me for details.
[3] If you need help with this, see me.

1.2 Text Editor

A *Text editor* is a program that lets you edit *plain text* documents (such as *R* source code) without inserting any formatting or other markup (as you would find in a document edited by Microsoft Word.) Additionally, all of the text editors described below provide additional capabilities to aid in the writing of *R* source code.

At first glance, the use of a text editor may seem superfluous; why edit your code elsewhere instead of typing it directly into *R*. The answer to this is threefold:

1. **Repeatability:** The act of typing code directly into an interpreter is innately error prone. Additionally, you will often find "chunks" of code which you find yourself using over and over. In order to speed up this process and ensure that the same code is used every time, it is beneficial to save the "chunk" in a code file. A text editor is the proper tool for this.
2. **Communication:** Having code stored in a text file makes it easy to share between users.
3. **Analysis:** Having code stored in a text file enables easy post-hoc analysis and modification.

With these benefits in mind, I recommend that each student find a text editor that they become comfortable with. The following are some suggestions:

1.2.1 Cross-platform

Cross-platform tools are tools which are available on multiple operating systems (i.e. Mac OSX, Windows, etc). Of the three cross-platform text editors listed below, two deserve special mention. *Vi(m)* and *Emacs* are the two most popular text editors in the world. They can be found on most modern operating systems without installing any software (with the exception of windows). They are both very mature tools with a lot of features, but both carry a significant learning curve. If you use Mac OSX or Linux, I would highly encourage you to take a look at one (or both) of them even if you don't end up using it as your "primary" text editing tool.

- **vi(m)** is (arguably) easier to use and learn than *Emacs*, and is available (as a source package) at `ftp://ftp.vim.org/pub/vim/unix/vim-7.2.tar.bz2`.
- **Emacs**, though somewhat more difficult to get started with, is a more full-featured tool and has a special add-on package called *ESS (Emacs Speaks Statistics)* which provides high-level integration with *R*.
- **jedit** is a relatively new Java-based cross-platform editor which can be downloaded (all platforms) from `http://prdownloads.sourceforge.net/jedit/jedit42install.jar`.

1.2.2 Mac OSX

The following *OS X* specific text-editors deserve special mention.

- **TextMate**, available at `http://macromates.com/` is a very easy-to-use and powerful text editor that I *highly* recommend to anyone running *OS X*.

- **MacVim**, available at `http://code.google.com/p/macvim/` is an enhanced version of *Vi(m)* which provides additional GUI capabilities and ease-of-use enhancements.
- **Aquamacs**, available at `http://aquamacs.org/` is an enhanced version of *Emacs* which provides GUI integration, ease-of-use enahancements, and includes many add-on packages such as *ESS*.

1.2.3 Windows

The default windows text editor, *Notepad*, provides only the bare minimum of features. Recommended alternatives include:

- **Gvim**, available at `ftp://ftp.vim.org/pub/vim/pc/gvim72.exe`, provides the power of the *vi* editor to windows users as well as an easier to use GUI.
- **Emacs** for windows can be downloaded from `http://ftp.gnu.org/pub/gnu/emacs/windows/emacs-23.1-bin-i386.zip`. I have no experience using *emacs* under windows.
- **e-texteditor** is a *TextMate* clone (see 1.2.2), providing many of the same features and the ability to use *TextMate* extensions. It is available from `http://www.e-texteditor.com/`.
- **notepad++** is another popular Windows text-editor with which I have no experience. It can be downloaded from `http://notepad-plus.sourceforge.net/uk/site.htm`.

2 Your first *R* session

Lets dive right into your first *R* session. If you are in the lab, Click on the `Finder` icon, click `Applications` in the left sidebar, find the Open `R.app` icon, and double-click it. You should be greeted by a message similar to

```
1  R version 2.10.0 Under development (unstable) (2009-06-03 r48708)
2  Copyright (C) 2009 The R Foundation for Statistical Computing
3  ISBN 3-900051-07-0
4
5  R is free software and comes with ABSOLUTELY NO WARRANTY.
6  You are welcome to redistribute it under certain conditions.
7  Type 'license()' or 'licence()' for distribution details.
8
9    Natural language support but running in an English locale
10
11 R is a collaborative project with many contributors.
12 Type 'contributors()' for more information and
13 'citation()' on how to cite R or R packages in publications.
14
15 Type 'demo()' for some demos, 'help()' for on-line help, or
16 'help.start()' for an HTML browser interface to help.
17 Type 'q()' to quit R.
18
19 >
```

2.1 Interpreter

Try entering the following commands into the *R* interpreter.

```
> 3
> 3 + 5
> 1:50
> x <- 1:5
> x / 2
```

You should see the following result:

```
> 3
```

```
[1] 3
```

```
> 3 + 5
```

```
[1] 8
```

```
> 1:50
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

```
> x <- 1:50
> x / 2
```

```
 [1]  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
[16]  8.0  8.5  9.0  9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0
[31] 15.5 16.0 16.5 17.0 17.5 18.0 18.5 19.0 19.5 20.0 20.5 21.0 21.5 22.0 22.5
[46] 23.0 23.5 24.0 24.5 25.0
```

Lets go through this one line at a time.

```
> 3
```

```
[1] 3
```

The *R* interpreter runs in what is called a *Read-Evaluate-Print* loop. It *reads* in commands as you type them, *evaluates* those commands, and finally *prints* the result to the screen. Here, you entered the number `3`, which was evaluated to `3`, and printed to the screen. The `[1]` preceding the `3` in the output indicates that there is only one result.

```
> 3 + 5
```

```
[1] 8
```

Here, the *R* interpreter evaluated the expression `3 + 5` and printed the result, `8`.

```
> 1:50
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

This expression introduces an important concept in *R*. The expression `1:5` means (to *R*) "all whole numbers between 1 and 50, inclusive" and represents a *vector*[4] or collection of values. In order to display this result on the screen, the numbers from 1 to 50 are split over several lines. Each line begins with a number in brackets, which denotes the "number" of each result.

```
> x <- 1:50
```

This expression introduces two additional important concepts. The first is that of a *variable*. A *variable* is a symbol which has a value assigned to it. Here `x` is a variable. The second concept is that of *assignment*[5]. It is the most basic of of variable operations, and is represented by the characters `<-`. The assignment operator works by taking the expression to its right (`1:50`), and assigning it to the variable to its left (`x`). From this point forward, typing `x` by itself is *exactly* the same as typing `1:50`. Try it. enter the following:

```
> x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

The result such be exactly the same as when you typed `1:50`.

```
> x / 2
```

```
 [1]  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
[16]  8.0  8.5  9.0  9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0
[31] 15.5 16.0 16.5 17.0 17.5 18.0 18.5 19.0 19.5 20.0 20.5 21.0 21.5 22.0 22.5
[46] 23.0 23.5 24.0 24.5 25.0
```

Here we show that the variable `x` can be used just like a number, and that basic operations (such as division) operate on all elements of a vector.

2.1.1 Example 1

Here is another example session for you to try, exploring further features of the *R* interpreter.

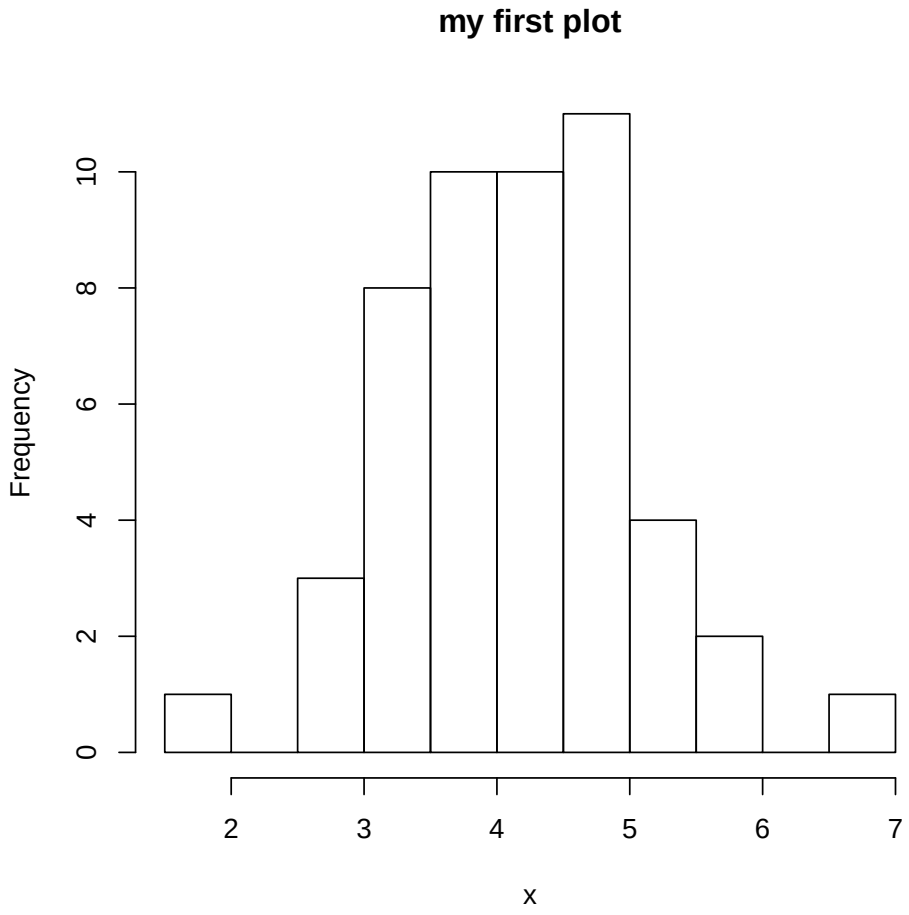
```
> x <- rnorm(50, mean=4)
> x
```

```
 [1] 5.017031 4.560544 4.289889 3.031587 4.381764 2.939694 1.656084 4.217989
 [9] 4.672228 5.823131 3.645826 4.552368 5.462737 4.110087 3.432377 4.602679
[17] 4.698746 3.520772 4.246894 3.006378 3.795803 5.675920 4.067957 3.594787
[25] 4.920940 2.886859 4.671890 3.133632 5.004439 5.342411 4.716417 4.076530
[33] 3.708200 4.554709 4.467479 3.657676 3.761465 4.136495 2.899990 4.718618
[41] 3.937597 3.178198 3.377057 3.131617 3.876911 4.326266 4.974828 6.535345
[49] 3.723242 3.198111
```

[4] This will be elaborated on 3.2.

[5] If you have used other programming languages before, this may seem strange (traditionally, `=` was used for assignment). Although *R* will generally permit you to use `=` instead of `<-` for assignment, this practice is strongly discouraged.

```
> mean(x)
[1] 4.118404
> range(x)
[1] 1.656084 6.535345
> hist(x)
> ?hist
> hist(x, main='my first plot', )
```



The *R* interpreter includes several useful features.

2.1.2 Tab completion

Tab completion is a process where a partially entered command is completed by the interpreter by looking for any possible command containing the text you have typed. Tab completion is performed by pressing the `[TAB]` key after typing part of a command. Try it. Suppose you couldn't remember the name of the command for finding square roots. Type into the interpreter `sq` and, without typing anything else, press the `[TAB]` key. You should find that `sq` has been expanded to `sqrt`! If there is more than one possible completion for the text you have entered, tab-completion will list all possible completions if you press `[TAB]` twice. Try getting all completions of `exp`. The result should look something like this[6].

```
> exp
```

```
exp
exp      expand.grid      expand.model.frame      expm1    expression
```

Tab-completion can also be used to find the arguments to a function. Consider the `plot` function. It has far too many options to remember. But, but typing `plot(` and pressing the `[TAB]` key, *R* will show you all possible arguments to the function `plot`[7].

```
> plot(
```

```
  [1] ""                "..."               "ci.lty="
  [4] "do.points="      "log="              "panel.last="
  [7] "x="              "absVal="           "ci.type="
 [10] "edge.root="      "lty.intervals="    "panel="
 [13] "xaxt="           "add.smooth="       "ci="
 [16] "edgePar="        "lty.predicted="    "par.fit="
 [19] "xlab="           "add="              "col.01line="
 [22] "formula="        "lty.separator="    "pch="
 [25] "xlim="           "angle="            "col.hor="
 [28] "frame.plot="     "lty="              "plot.type="
 [31] "xpd="            "ann="              "col.intervals="
 [34] "freq="           "lwd="              "predicted.values="
 [37] "xval="           "ask="              "col.points="
 [40] "grid="           "main2="            "qqline="
 [43] "xy.labels="      "asp="              "col.predicted="
 [46] "hang="           "main="             "range.bars="
 [49] "xy.lines="       "axes="             "col.range="
 [52] "horiz="          "mar.multi="        "separator="
 [55] "y="              "border="           "col.separator="
 [58] "id.n="           "mar="              "set.pars="
 [61] "yax.flip="       "caption="          "col.vert="
 [64] "intervals="      "max.mfrow="        "sub.caption="
 [67] "yaxt="           "center="           "col="
 [70] "label.pos="      "mgp="              "sub="
 [73] "ylab="           "cex.caption="      "conf="
```

[6] In the OSX GUI, a pop-up menu showing possible completions will be shown instead of printing the possibilities to the screen.

[7] In the OSX GUI, tab-completion for function arguments does not work. Instead, the syntax for the function (including arguments) is always shown along the bottom of the GUI window.

```
 [76] "labels.id="       "nc="              "subset="
 [79] "ylim="            "cex.id="          "cook.levels="
 [82] "labels="          "nodePar="         "type="
 [85] "zero.line="       "cex.main="        "dLeaf="
 [88] "leaflab="         "oma.multi="       "verbose="
 [91] "cex.points="      "data="            "legend.text="
 [94] "oma="             "verticals="       "ci.col="
 [97] "density="         "levels="          "panel.first="
[100] "which="
```

Finally, tab-completion can be used to complete filenames when used inside single or double quotation marks.

2.1.3 History

A second useful feature of the *R* interpreter is the *command history*. *R* keeps a record of every command you have entered since opening the *R* interpreter. At the `>` prompt, you can use the `[UP]` and `[DOWN]` arrows to move backwards and forwards through your command history. Try it.

You can view your full command history by using the command `history()` (press textttq to quit the history browser). Finally, you can use the command `savehistory()` to write your history to a text file for later inspection (default name is `.Rhistory`).

2.1.4 Comments

In *R*, the pound character (`#`) is used to denote comments. In general anything after between a `#` and the end of a line is ignored. This allows explanations and textual notes to be included directly in *R* source code. The only time that a `#` does not indicate a comment is when it is surrounded by single- or double- quotation marks.

```
> # This is a comment
> '# This is not a comment'
```

```
[1] "# This is not a comment"
```

2.1.5 Prompt

As you have seen, the standard *R* prompt is the `>` character. Occasionally, you will see the plus sign (`+`) shown as a prompt instead. This is a *continuation prompt* and signifies that the command entered on the previous line is not complete. This allows you to break long commands over multiple lines.

```
> 3 + 5 +
+ 2
```

```
[1] 10
```

```
> sqrt(
+   3.141)
```

```
[1] 1.772287
```

Occasionally, other (self-explanatory) prompts will be shown, such as

```
Hit <Return> to see the next plot:
```

2.1.6 Exploring the current environment

Sometimes, it is easy to loose track of the variables you have defined. *R* provides a simple command to view them: `ls()`.

```
> ls()
```

```
[1] "oldRout"          "oldRset"          "plotCompletions" "randomData"
[5] "startuptext"      "sumDigits"        "x"
```

```
> ls(all.names=TRUE)
```

```
 [1] ".myRset"          ".Random.seed"     ".required"       ".Rout"
 [5] "oldRout"          "oldRset"          "plotCompletions" "randomData"
 [9] "startuptext"      "sumDigits"        "x"
```

2.1.7 Getting help

One of *R* ś greatest strengths is the availability of online help inside the interpreter. When inside the *R* help system, you can move up and down with the arrow keys, and return to the interpreter by pressing `q`.

2.1.8 Help browser

There are two general ways to invoke the *R* help system (described below).

- Using the `help` command:

```
> help('plot')
```

```
plot                  package:graphics                  R Documentation

Generic X–Y Plotting

Description:

     Generic function for plotting of R objects.  For more details
     about the graphical parameter arguments, see 'par'.

Usage:
```

- Using the `?` shortcut:

```
> ?plot
```

In general, you can receive help for any command *thisCommand* using either `help('thisCommand')`[8] or with `?thisCommand`.

In increase the scope of your search of the help system, try

```
> help.search('plot')
> ??plot
```

Finally, you can invoke the *HTML* help browser (if available) using the command `help.start()`.

2.1.9 Examples

Another feature of the online help systems is the `example()` function. *R* will print and execute any code shown in the "Examples" section of a commandś online help.

```
> example('Arithmetic')
```

```
Arthmt> x <- -1:12

Arthmt> x + 1
 [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13

Arthmt> 2 * x + 3
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27

Arthmt> x %% 2 #-- is periodic
 [1] 1 0 1 0 1 0 1 0 1 0 1 0 1 0

Arthmt> x %/% 5
 [1] -1  0  0  0  0  0  1  1  1  1  1  2  2  2
```

2.1.10 Session

An *R* session consists of the commands and variables created from the time you start the *R* interpreter until you quit. *R* provides several facilities for managing, saving, and restoring sessions.

- `save.image()` - This command saves the current session to a file named `.RData` in the current directory.
- `save()` - This command saves an *R* object (given as an argument) to a file.
- `load()` - This command takes as an argument a file (such as `.RData`) and restores it into the current session.
- `source()` - This command executes all *R* commands in the file given as an argument.

[8] Note the use of quotation marks.

- `dump()` - This command "dumps" the current session as a series of commands to a file given as an argument. The file produced by `dump()` make be loaded using `source()`.

2.1.11 Quitting

To quit the *R* interpreter, use the command `q()`. *R* will then ask you if you want to save the current session. If you do, `save.image()` will be used to write the current session to disk.

Note that, by default, *R* *always* loads the file `.RData` on startup if it finds one in its working directory. This means that *R* will always load the most recently save session image. To remove this file (within *R*), you may issue the command `unlink('.RData')`.

To clear the current session, issue the command `rm(list=ls())`.

2.1.12 Aborting

Occasionally, you may find that *R* seems "stuck." Perhaps you mistakenly created an infinite loop or entered a prohibitively complicated command. If you would like to cancel in-progress computation, press `C-c` to abort the running computation[9]. You can also use `C-c` to clear a partially entered command and start at a clean `>` prompt. In the event this doesn't work, the operating system process manager can be used to halt the offending process[10]

3 Exploring *R*

3.1 Example 2: Calculator

Included below are a couple of quick examples highlighting the use of *R* as a calculator. A more complete listing of basic calculation functions is provided in table 1.

```
> # Arithmetic
> 3 / 5
```

```
[1] 0.6
```

```
> 301 + 50000003
```

```
[1] 50000304
```

```
> 0.0005 * 0.0001
```

```
[1] 5e-08
```

[9] In the OSX GUI, the `[Esc]` key is used instead

[10] How this is done is operating system dependent. In windows, `C-M-[del]` will bring up the Task Manager, which may be used to terminate *R* . In Linux/MacOSX, the command `sudo killall -9 {R,R.app}`, entered at a Terminal prompt will terminate *R* .

category	functions
arithmetic	`+,-,*,/,%%`
exponential	`exp(), log(), log10(), log2()`
trigonometric	`cos(), sin(), tan(), acos(),asin(),atan() cosh(), sinh(), tanh(), acosh(), asinh(), atanh()`
approximation	`abs(), sign(), sqrt(), floor(), ceiling(), trunc(), round(), signif()`

Table 1: Table of standard mathematical functions

```
> -0.0001 ** 9
```

```
[1] -1e-36
```

```
> -0.0001 ^ 9
```

```
[1] -1e-36
```

```
> ## exponentiation can be represented with either ** or ^
> 3 + 5 * 2
```

```
[1] 13
```

```
> (3 + 5) * 2
```

```
[1] 16
```

```
> ## special operations are called by name
> sin(3)
```

```
[1] 0.14112
```

```
> sqrt(5)
```

```
[1] 2.236068
```

```
> ## complex numbers are supported when written as x + yi
> -1 + 0i
```

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

```
> sqrt(-1 + 0i)
```

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

```
> ## constants can be called by name or expression (varies)
> pi
```

```
[1] 3.141593
```

```
> exp(1)
```

```
[1] 2.718282
```

3.2 Example 3: Variables

As discussed in 2, variables play a key part in computational programming. In *R* , variables are defined using the **variable**<-value syntax. Here, we will discuss some of the nuances and common pitfalls encountered when using variables in *R* .

3.2.1 Names

Variable names in *R* consist of letters, numbers, underscores, and periods (“.”), subject to the following constraints.

- Variable names cannot start with an underscore or a number. Variable names *generally* should not begin with a period[11].
- Variable names are case sensitive. This means that the names `myVariable` and `myvariable` are *not* the same.
- There are a small number of reserved variable names (listed in table **??**) which cannot be used. In general, however, the names of *R* functions are *not* protected. Thus, it is possible to define a variable named `print` which will prevent you from using the `print()` function. Beware of this.

Reserved Names

`if`	`TRUE`	`else`	`FALSE`
`repeat`	`NULL`	`while`	`Inf`
`function`	`NaN`	`for`	`NA`
`in`	`NA_integer_`	`next`	`NA_real_`
`break`	`NA_complex_`		

Table 2: Reserved names in *R* .

In addition to these restrictions, style[12] provides an additional list of *suggestions*.

- Variable names should be descriptive. For example a variable containing the number of students in a class: `numStudents` - good, `ns` - bad.
- Variable names should not be written in all capital letters and should not start with capital letters. `numStudents` - good, `NUM_STUDENTS` - bad, `num_students` - good, `numstudents` - less good

[11] Variables with names beginning with a period are “hidden” from the you and many of *R* ś internal functions.
[12] TODO: Talk about style / reference to style

3.2.2 Types

Variables are classified by *type*, that is, what "kind" of information they contain. Just as names tell you "who" a variable is, types tell "what" a variable is. In *R* there a number of different *types* suited for the analysis of different problems. Here we shall explore only the most universal of *types* provided in *R* . The basic *types* are summarized in table 3.

type name	description	example & usage notes
numeric	number values	`3, 5, pi, 0.001, 10e6, 0+3i`
logical	boolean values	`TRUE, FALSE`
character	sequence of letters, numbers and punctuation	`"David Rosenberg", "MyCharacterString"` *NOTE:* data of type character must be surrounded by either single or double quotation marks.
function	a built-in or user defined function	`exp, print, help`

Table 3: Basic data types

3.2.3 Attributes and coercion

It is possible (and sometimes necessary) to treat variables of one type as if they were of another type. The process of converting data from one type to another is called *coercion*, and is used most frequently when importing data from other programs (such as Microsoft Excel) into *R* . There are a set of commands in *R* for performing variable coercion with names such as `as.numeric()` and `as.character()`. If you need to use these functions, consult the online documentation for guidance.

One special case of coercion that does not require any commands is coercion from type *numeric* to *logical*. A numeric variable equal to 0 is treated as `FALSE` when used as a logical. All other numeric variables are treated as `TRUE`.

TODO: Do attributes deserve mention here?

3.2.4 Vectors and Matrices

It is often useful to use several "values" to describe a single "thing" or to refer to a group of related variables with a single name. *R* provides three constructs for creating multi-valued variables (all of which must be of the same type).

Vectors are the most basic of these "compound" variables and are, in fact, the primary way in which *R* handles data. Individual values in a vector (sometimes called members) are each assigned a unique integer or index. The easiest way to think of a vector is as an ordered sequence.

$$(a_n) = a_1, a2, \ldots$$

Individual members of a vector are referenced by using brackets as follows

```
> firstVector <- letters[1:10];
> firstVector    # if no index is given, a variable name referes to all members

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

> firstVector[5]     # The fifth member of firstVector

[1] "e"

> firstVector[5] <- 'E';  # Members can be assigned by indexing as well
> firstVector

 [1] "a" "b" "c" "d" "E" "f" "g" "h" "i" "j"
```

Vectors are typically constructed using either the `c()` command or using a special `start:end` notation.

```
> secondVector <- c(1,4,9,16)          # c() is used to make vectors
> secondVector                         #

[1]  1  4  9 16

> biggerVector <- c(secondVector, 25) # and to add to them
> numVector <- 1:5                     # a vector of integers can by made using
> numVector                            #       the start:end notation

[1] 1 2 3 4 5

> numVector[c(1,3,5)]                  # You can use vectors to index vectors

[1] 1 3 5
```

In general, most functions operations accept vectors just like regular variables and work in an element-wise fashion. Exceptions to this rule include functions that only make sense with multi-valued input (such as `max()`) and some user-generated functions.

```
> firstVector <- 1:15
> firstVector.1 <- firstVector / pi
> firstVector.1

 [1] 0.3183099 0.6366198 0.9549297 1.2732395 1.5915494 1.9098593 2.2281692
 [8] 2.5464791 2.8647890 3.1830989 3.5014087 3.8197186 4.1380285 4.4563384
[15] 4.7746483

> firstVector.2 <- sin(firstVector.1)
> firstVector.2

 [1]  0.31296180  0.59448077  0.81627311  0.95605566  0.99978466  0.94306673
 [7]  0.79160024  0.56060280  0.27328240 -0.04149429 -0.35210211 -0.62733473
[13] -0.83953993 -0.96739776 -0.99806251
```

```
> length(firstVector)            # length() is used to find out how many members

[1] 15

>                                # a vector has
> max(firstVector)               # largest member

[1] 15

> sum(firstVector)               # sum of all members

[1] 120

> firstVector > pi/2             # a logical vector is returned

 [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[13]  TRUE  TRUE  TRUE
```

Matrices are similar to vectors except that they have an additional dimension of indexing. Matrices are generated with the `matrix()` function and can be merged using the commands `rbind()` and `cbind()`. Two important points about matrices to remember are:

1. *R* defaults to *column-major* format for matrices[13]
2. Arithmetic operations are normally performed elementwise on matrices. Matrix multiplication and division are performed using `%*%` and `%/%`

```
> matrix(data=c(1:12), nrow=4, ncol=3)     # Note how the values go DOWN first

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

> matrix(data=c(1:12), nrow=4, ncol=3, byrow=TRUE)    # byrow=TRUE avoids this

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

> matrix(c(1:12), 4, 3, byrow=TRUE)        # The xxx= are optional

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

[13] TODO: row-major vs. column-major

```
> matrix(c(1:12), 4, byrow=TRUE)          # only one of nrow, ncol is needed
```

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

```
> matrix(c(1:12), ncol=3, byrow=TRUE)
```

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

```
> myMatrix <- matrix(c(1:12), 4, byrow=TRUE)
> t(myMatrix)                             # t() transposes a matrix
```

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

```
> myMatrix
```

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

```
> myMatrix[4,3]                    # members are referenced by two indices
```

```
[1] 12
```

```
> myMatrix[4,]                # rows/columns are extracted by omitting an index
```

```
[1] 10 11 12
```

```
> myMatrix[,3]
```

```
[1]  3  6  9 12
```

```
> myMatrix[c(1:2), c(1:3)] # vector indexing works as before
```

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

```
> rbind(myMatrix, myMatrix)        # rbind() joins matrices vertically
```

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

```
> cbind(myMatrix, myMatrix)        # cbind() joins matrices horizontally
```

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

```
> myMatrix / 5                     # Operations are always performed element-wise
```

```
      [,1] [,2] [,3]
[1,]  0.2  0.4  0.6
[2,]  0.8  1.0  1.2
[3,]  1.4  1.6  1.8
[4,]  2.0  2.2  2.4
```

```
> myMatrix * myMatrix              # (Even multiplication)
```

```
      [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
[3,]   49   64   81
[4,]  100  121  144
```

```
> myMatrix %*% t(myMatrix)         # unless the %*% notation is used
```

```
      [,1] [,2] [,3] [,4]
[1,]   14   32   50   68
[2,]   32   77  122  167
[3,]   50  122  194  266
[4,]   68  167  266  365
```

Arrays are similar to matrices and vectors except that they allow any number of dimensions.

3.2.5 Lists and data.frames

1. Lists
   (a) indexing
   (b) extracting
   (c) nesting
2. data.frame
   (a) concept
   (b) relation to lists
   (c) importance

3.2.6 Special values

- NULL
- NaN
- NA

3.3 Boolean Logic

One of the fundamental concepts of computer programming, and one often unfamiliar to non-programmers, is the concept of *boolean logic*. Boolean algebra is a simple mathematical system containing two values (**True** and **False**) and three operations (**AND**, **OR** and **NOT**). The following table outlines how boolean logic is represented in *R* .

	R representation	context	meaning
true	`TRUE`	`a`	`a` is true
false	`FALSE`	`b`	`b` is true
not	`!`	`!a`	inverse of `a`
and	`&` and `&&`	`a & b`	`a` and `b`
or	`|` and `||`	`a	b`	`a` or `b`

The conventions for using these operations are bit unusual.

- **AND** (`&`) takes two arguments, `a` and `b`. One is placed before the `&` and one after. The expression `a & b` evaluates to `TRUE` if and only if `a` evaluates to `TRUE` and `b` evaluates to `TRUE`.
- **OR** (`|`) takes two arguments, `a` and `b`. One is placed before the `|` and one after. The expression `a | b` evaluates to `TRUE` if `a` evaluates to `TRUE`, if `b` evaluates to `TRUE`, or if both `a` and `b` evaluate to `TRUE`.
- **NOT** (`!`) takes one argument, `a`, placed directly after the `!`. The expression `!a` evaluates to `TRUE` if and only if `a` evaluates to `FALSE`.

Boolean logic is extremely important for *conditionals*, which we will explore below.

One final aspect of *boolean logic* which we must consider is that of *boolean indexing*. Recall that we can extract a subset of a vector by putting a vector of integer indexes in brackets.

```
> myVec <- letters
> myVec
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
> myVec[1:5]            # First five letters
```

```
[1] "a" "b" "c" "d" "e"
```

You can also use a *logical vector* (a vector of only `TRUE` or `FALSE` members) to index a vector as well. The result of such an operation is a vector composed of all the members of the original vector which were indexed by `TRUE`. An example may help to clarify this

```
> myVec <- letters
> myVec
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

```
> idx <- 1:26
> bool_idx <- idx < 10              # logical vector
> bool_idx
```

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE
```

```
> myVec[bool_idx]                   # the first 10 letters of the alphabet
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i"
```

3.3.1 Conditionals

Conditionals are "tests" which return either `TRUE` or `FALSE` with respect to a particular variable. The most familiar conditionals are likely the *comparison operators*

comparison	*R* representation	context	notes
equality	`==`	`a == b`	True if `a` and `b` have the same value
inequality	`!=`	`a != b`	synonomous with `!(a == b)`
less than, greater than	`<`, `>`	`a < b`	true if `a` is less than `b`
less than or equal to	`<=`, `>=`	`a <= b`	synonomous with `(a < b) | (a == b)`
set-theoretic inclusion	`%in%`	`a %in% b`	true if vector `b` contains `a` as one of its members
vector union	`all()`	`all(a)`	true if every member of vector `a` is true
vector intersection	`any()`	`any(a)`	true if any member of vector `a` is true

The most basic use of conditionals involves the *R* `if` command, which has the following syntax:

```
> if (CONDITION) {
+   BLOCK_1
+ } else if (CONDITION_2) {
+   BLOCK_2
+ } else {
+   ELSE_BLOCK
+ }
```

The capitalized expressions represent code which may be changed:

expression	meaning
`CONDITION`	a condition which evaluates to `TRUE` or `FALSE`
`BLOCK_1`	a sequence of commands evaluated if `CONDITION` evaluates to `TRUE`
`CONDITION_2` (optional)	a second condition which is evaluated only if `CONDITION` evaluates to `FALSE`. Note that as many `else if()` clauses may be used as one chooses.
`BLOCK_2` (optional)	sequence of commands to be evaluated if `CONDITION_2` is evaluated and `TRUE`
`ELSE_BLOCK` (optional)	sequence of commands to be evaluated if none of the `CONDITION` blocks evaluate to be true.

An example may help to clarify this syntax.

```
> x1 <- 2
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
+ } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
> x1 <- 16
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
+ } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
> x1 <- 128
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
```

```
> } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
> x1 <- 1024
> if (x1 < 10) {
+   cat(x1, " has only 1 digit.")
+ } else if (x1 < 100) {
+   cat(x1, " has two digits.")
+ } else if (x1 < 1000) {
+   cat(x1, " has three digits.")
+ } else {
+   cat(x1, " has more than three digits.")
+ }
```

This example should produce the following output:

```
2  has only 1 digit.

16  has two digits.

128  has three digits.

1024  has more than three digits.
```

3.3.2 Flow control / loops

Flow control, the process by which commands are "selected" and evaluated is defined by two major constructs: *conditionals* (seen in the previous section) and *Loops*. A *loop* is a section of code which is repeatedly evaluated (often with variables assuming different values). In *R* , there are two main types of loops which we will consider.

While loops are used to evaluate a set of commands based on the result of a *conditional* test. The syntax for a `while` loop is as follows:

```
> while(CONDITION) {
+   BLOCK
+ }
```

Here, *`CONDITION`* is a conditional statement (see above), and *`BLOCK`* is a series of commands which are evaluated, in sequence, until *`CONDITION`* evaluates to `FALSE`. As an example, consider the fibonacci sequence $f_n = 0, 1, 1, 2, 3, 5, \ldots$, defined by:

$$f_n = \begin{cases} 0 & \text{if } n = 1, \\ 1, & \text{if } n = 2, \\ f_{n-1} + f_{n-2}, & \text{if } n > 2 \end{cases} \tag{1}$$

Suppose that we wanted to find the first number in the fibonacci sequence greater than 1000.

```
> x0 <- 0
> x1 <- 1
> while(x1 < 1000) {       # x1 < 1000 is the condition
+   newX1 <- x0 + x1       # -\
+   x0 <- x1               # --+ -- these three lines make up the BLOCK
+   x1 <- newX1            # -/
+ }
> x1
```

```
[1] 1597
```

The second of our looping constructs, the `for` loop, is used to repeat a sequence of commands a predetermined number of times.

```
> for (COUNTER in VECTOR) {
+   BLOCK
+ }
```

Here *COUNTER* is a variable which changes from one iteration of the loop to the next and *VECTOR* is a vector of values. *BLOCK* is evaluated once for each value in *VECTOR*. In each iteration, *COUNTER* is assigned a new value, taken from the members of *VECTOR*.

Continuing with the previous example, suppose now you wanted to know the first 20 values of the fibonacci sequence. Then:

```
> xVec <- c(0, 1)
> for (ii in 3:20){
+   xminus1 <- xVec[ii-1]
+   xminus2 <- xVec[ii-2]
+   newVal <- xminus1 + xminus2
+   xVec <- c(xVec, newVal)
+ }
> xVec
```

```
 [1]    0    1    1    2    3    5    8   13   21   34   55   89  144  233  377
[16]  610  987 1597 2584 4181
```

There are two special commands for dealing with loops

- `break` immediately terminates the loop and continues evaluation at the first command following the end brace of the loop.
- `next` ends the current loop iteration and starts the next one.

3.4 Example 4: Functions

One of the most fundamental of programming concepts is that of the *function*, a series of commands which take several values (variables) as input and return the result of computations on that input. It is generally helpful (and frequently necessary) to place a series of related commands in a function. *R* lets you define functions using the following syntax

```
> functionName <- function(argumentList) {
+   codeBlock
+   {return(returnValue)}
+ }
```

Breaking the components down, we have

1. *functionName* is the name you would like to assign the new function to.[14].
2. *argumentList* the inputs to the function
3. *codeBlock* the commands to be evaluated
4. *returnValue* (optional) the output value that is "returned" by the function

As an example, let us define a function which takes as input an integer, `k` and returns the first number in the fibonacci sequence equal to or greater than that number.

```
> fibLargerThan <- function(k) {
+   x0 <- 0
+   x1 <- 1
+   while(x1 < k) {
+     newX1 <- x0 + x1
+     x0 <- x1
+     x1 <- newX1
+   }
+   return(x1);
+ }
> fibLargerThan(100)
```

```
[1] 144
```

```
> fibLargerThan(1000)
```

```
[1] 1597
```

```
> fibLargerThan(10000)
```

```
[1] 10946
```

- Parts
  1. name
  2. args
  3. body
  4. return
- scope
- calling

[14] TODO: MORE ABOUT FUNCTION NAMES

3.5 Miscellany

4 Source files

As discussed in the overview, keeping *R* source code in text files makes maintaining and using your code much easier[15]. This section is intended to help you in the organization and construction of source code files.

4.1 Overview

Although there are few hard and fast rules defining what and how you may use source files (any commands that may be entered into the *R* interpreter may be put into a source file), the notion of *style* provides **strong** suggestions as to what should and should not be done. What follows below are general guidelines you *may* wish to consider[16]. More specific and detailed guidelines are available at `http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html` (Google's *R* style guide) and `http://www1.maths.lth.se/help/R/RCC/` (*R* Coding Conventions draft).

R source files are generally described as being *function* files or *script* files. We will explore each separately.

4.2 Style

Unlike the syntactic requirements and other "necessary" elements previously explored, code *style* is concerned not with whether the machine (the *R* interpreter) can read a source file but instead with whether a *person* can read (and understand) it. Following a consistent coding style ensures that both you and your peers can read and understand your code.

Our intent here is not to burden you with more "rules" or "requirements", but instead to provide you with helpful guidance which many have found to be helpful. Table 4 summarizes our suggestions on *R* source code style.

A more concrete example of these guidelines is provided in the following code listing. Where applicable comments referencing style guidelines are prefixed with `##STYLE`.

```
1 # simpleFactor.R
2 #
3 # Tools for finding the prime factorization of integers using the
4 # sieve of Eratosthenes.
5 #
6 ##STYLE - lines 1-4 serve as a 'commented' header
```

[15] A famous computer scientist once remarked that being a good programmer *required* one to be a bit lazy TODO: REFERENCE NEEDED HERE. While I doubt many professors would actively encourage laziness, there is no doubt that, when it comes to computer programming, the old adage of "work smarter, not harder" holds true. By building a collection of source files, you accumulate a body of code which you will (hopefully) be able to use at a later date.

[16] In the end, having a consistent style helps *you* more than anyone else. You will likely spend many more hours looking at your code than anyone else, so it pays to make your code easy to read.

Category	Description
variable names	Variable names should describe their function and follow a consistent capitalization scheme
indentation	*code blocks* (generally delimited by brackets) should be indented (recommended 2 spaces) relative to surrounding code
blank lines	use blank lines to separate independent "chunks" or concepts
spaces	blank spaces should surround symbols and parentheses
comments	comments should be used liberally to explain code and concepts
header	a "commented" header should be placed at the top of each source file describing its contents and other relevant info
line length	Individual lines should be no more than 80 characters long. Continuation lines (if necessary) should be indented relative to surrounding code
consistency	above all be consistent in your style choices (at least within a single file)

Table 4: Summary the basic *R* source code style recommendations.

```
7  #
8  # getPrimeFactors
9  # Prime factorization function.
10 #
11 # argument input_number - Integer to find the prime factorization of
12 # returns - A numeric vector listing the prime factors of input_number.  The
13 #           multiplicity of each number in the return vector represents the
14 #           multiplicity of that factor in input_number's prime factorization
15 #
16 ##STYLE - lines x - y describe the input and output of the getPrimeFactors
17 ##STYLE   function
18 getPrimeFactors <- function(input_number) {
19   ##STYLE - note that the name getPrimeFactors is explains the function well
20   if (input_number > 100000) {
21     stop(input_number, " is too large to efficiently factor\n");
22   }                              # The sieve of Eratosthenes isn't very fast
23
24   ##STYLE - Note how we increased indentation inside the function declaration
25   ##        and again inside the if-block
26   prime_factors <- c();
27
28   max_test_to <- floor(sqrt(input_number));
29   # The largest possible prime factor is the square root of input_number
30   possible_factors <- 2:max_test_to;
31
```

```
32    isFactor <- function(dividend, divisor) {
33      remainder <- dividend %% divisor;
34      # Remeber %% is the modular divisior operator
35      if (remainder == 0) {
36        return(TRUE);
37      } else {
38        return(FALSE);
39      }                                   # end if
40    }                                     # end function isFactor
41
42    ##STYLE - It is often useful to label closing braces
43
44    getPrimeMultiplicity <- function(dividend, divisor) {
45      multiplicity_count <- 0;
46      while(dividend %% divisor == 0) {
47        multiplicity_count <- multiplicity_count + 1;
48        dividend <- dividend / divisor;
49      }
50      return(multiplicity_count);
51    }
52
53    removeMultiples <- function(total_list, divisor) {
54      new_list <- total_list[total_list %% divisor != 0];
55      # logical indexing
56      return(new_list);
57    }                                     # end function removeMultiples
58
59    while (length(possible_factors) > 0) {
60      divisor <- possible_factors[1];
61      possible_factors <- possible_factors[-1];
62      # possible_factors[-1] refers to all elements of possible_factors except
63      #        the first
64      if (isFactor(input_number, divisor)) {
65        multiplicity_count <- getPrimeMultiplicity(input_number, divisor);
66        prime_factors <- c(prime_factors, rep(divisor,
67                                              multiplicity_count));
68
69        ##STYLE - the previous line was split to keep it from being too long.
70        ##        note the indentation so that divisor and multiplicity_count
71        ##        line up
72        possible_factors <- removeMultiples(possible_factors, divisor);
73      }                                   # end if
74    }                                     # end while
75
76
77    ##STYLE - Note naming consistency.  I use camelCase for function names and
78    ##        All-lowercase-with-underscores for variables.
79
80    if (length(prime_factors) == 0) {
81      prime_factors <- c(1, input_number);
```

```
82    } else {
83      prime_factors <- c(1, prime_factors);
84    }
85    return(prime_factors);
86  }
```

4.3 Functions

R function files are files that contain one (or more, if related) function definitions (see 3.4). and

4.4 Scripts

Part II

Exercises

1. (a) Starting with the fibonacci functions from the tutorial, write a new function which takes as input a number `k` and returns the largest fibonacci number less than or equal to `k`

   (b) Write a function which takes as input three numbers (say, `k`, `x`, and `y`) and returns either the second number (`x`) or the third number (`y`) depending on which is closer to `k`. (*Hint:* the functions `min()`, `max()`, and `abs()` may be helpful. See the online documentation for their usage.)

   (c) Write a function which takes as input a number `k` and returns the fibonacci number with is closest to `k`.

2. Write a function that takes as input a vector of integers and returns a vector of those integers, sorted from smallest to largest.[17]

3. Write a function that takes as input a polynomial[18] and returns the polynomial representing the first derivative of the input polynomial. You will be provided two functions, `parsePolynomial()` and `deparsePolynomial` to help you in this task. The function `parsePolynomial()` takes as input a polynomial in x and returns a vector of exponents and a vector of coefficients. The function `deparsePolynomial()` takes as input a vector of exponents and a vector of coefficients and returns a polynomial in x.

```
#!/usr/bin/env rr
# encoding: utf-8
# parsePolynomial.R
#
# parsePolynomial - a function for parsing a polynomial (in x) into a
```

[17] TODO: provide a hint
[18] TODO: parsing function

```
#             vector of coefficients and a vector of exponents.
# argument  - inputExpression - a polynomial in x, expressed as a string.
#             For example,   "x^3 + 2x^2 - x - 1"
# returns   - a list with two components, coefficients and exponents,
#             representing the coefficients and exponents of the polynomial,
#             respectively.
#
# Example:
#       > inputEx <- "x^3 + 2x^2 - x - 1"
#       > result <- parsePolynomial(inputEx)
#       > result[['exponents']]
#       [1] 3 2 1 0
#       > result[['coefficients']]
#       [1] 1 2 -1 -1
#
parsePolynomial <- function(inputExpression) {
  numberTerms <- nchar(gsub("[^x]", "", inputExpression));
  coefficients <- numeric(length=(numberTerms)+1);
  exponents <- numeric((length=numberTerms)+1);
  splitTerms <- strsplit(gsub('-', '+ -', inputExpression), '\\+')[[1]]
  for (ii in seq(along=splitTerms)) {
    term <- gsub(' ', '', splitTerms[ii]);
    if(gsub("[^x]", "", term) == '') {
      term <- paste(term, 'x^0', sep='');
    }
    if (gsub("([\\+|-]*)(.*)x.*", "\\2", term) == '') {
      term <- gsub('x', '1x', term);
    }
    if (gsub(".*x", "", term) == '') {
      term <- paste(term, '^1', sep='');
    }
    coef <- eval(as.integer(gsub('x.*', '', term)));
    coefficients[ii] <- coef;
    exponent <- eval(as.integer(gsub('.*\\^(.*)', '\\1', term)));
    exponents[ii] <- exponent;
  }
  o <- -(order(exponents)) + length(exponents) + 1;
  exponents <- exponents[o];
  coefficients <- coefficients[o];
  result <- list(exponents=exponents, coefficients=coefficients);
  return(result);
}
```