Introduction to Programming in R by Timothy DelSole

This document is a tutorial to the software package R to accompany *Statistical Methods for Climate Scientists* by Timothy DelSole and Michael K. Tippett. It is written in an informal tone to engage the student in their first exposure to R.

To analyze climate data, it is essential to use a computer. Anyone of the following software packages could be used: Matlab, R, Python, SAS. This tutorial provides an introduction to a statistical package known as R. No prior experience with these software packages is assumed in this tutorial. I have chosen R because it is Open Source (i.e., free), well documented, supported by a large community, specifically designed for data analysis and graphical display, and is widely used by statisticians. Even though learning a new programming environment takes time, you will save more time as a result of using R than you will spend learning R itself.

The following introduction assumes that you are familiar enough with computers to login, change directories, and edit files. This introduction will familiarize you with the main commands in R that are most needed to perform the statistical calculations described in the first few chapters of the book.

1 Installing R

You can either download R yourself or use the R code installed at the COLA computers. I personally downloaded R onto my MAC laptop from http://cran.r-project.org/bin/macosx/ without any trouble at all. I am not able to provide support for installing R.

You are advised to create a separate directory for each statistical project on which you work. I suggest first creating the directory "CLIM762", which will contain all your work for this class, and then creating the subdirectory "R_intro," which will contain the work for this introductory session.

2 Running R

If you are on a Linux or Unix system, you start an R session by typing `R` (make sure you use capital `R`, not lower case `r`– commands in UNIX and R are case sensitive). On the Mac or PC, you click the R icon. In either case, you should see a greeting that looks something like the following:

```
R version 2.13.1 (2011-07-08)
Copyright (C) 2011 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: i386-apple-darwin9.8.0/i386 (32-bit)

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

  Natural language support but running in an English locale

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

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

[R.app GUI 1.41 (5874) i386-apple-darwin9.8.0]

>
```

The last line is the > symbol, which is the R prompt. You type commands at the R prompt interactively. This interactive window is called the *console*.

For codes longer than a dozen lines or so, you will want to save code in a text file. For future reference, we discuss how to do that here. Let the name of the file containing R code be `calculations.R`. Then, when you want to run this code, you type

```
source("calculations.R")
```

After typing this command, R will accept input from the named file. In particular, each line in the file `calculations.R` will be evaluated sequentially just as if they were typed in the interactive window. After all expressions in the file have been evaluated, R will return to interactive mode. If an error occurs, then R will terminate reading from the file. By default, none of the expressions inside the source file will appear on the console. This can be problematic when an error occurs, because the error message may give no clue as to which line caused the error. To force R to echo the expressions to the terminal before evaluating them, use the `echo=TRUE` argument:

```
source("calculations.R",echo=TRUE)
```

Another valuable command for debugging is `traceback()`, which will display "the stack," which often (but not always) indicates the line number of the last error

3 Numeric Vectors

The easiest way to teach a programming language is by example. So, let's get started. First, let us create a variable named x and set its value to 5:

```
1 > x = 5
2 > x
3 [1] 5
```

In the above example, we typed `x = 5` in line 1. This is an example of an *assignment* statement. Assignments can be executed using either = or <−, where the latter is merely a 'greater than' symbol < followed by a dash −. In certain situations, = does not work, so <− is preferred; nevertheless, we will use = in this introduction. The command `x = 5` performs two operations: first, it creates a variable x, and second, it sets the value of x to 5. In line 2, we typed x by itself. Line 3 shows R's response, which is "`[1] 5`", indicating that the first element of x is 5. It is worth emphasizing that the = function does not mean "equal to," but rather means "assign a value to." This distinction is especially important when the variable occurs on both sides of =. For instance, consider the following:

```
1 > x = 5
2 > x = x + 1
3 > x
4 [1] 6
```

If the function = were interpreted as "equal to," then line 2 would define an equation to be solved for x. Moreover, the solution to this equation is the null set– no value of x satisfies the equation $x = x + 1$! However, the proper interpretation of line 2 is: first evaluate the right hand side by incrementing the value of x by one, *then* assign the resulting value to x, thereby overwritting the original value of x.

Most numerical calculations are performed on groups of numbers. The basic way of handling groups of numbers in R is with *vectors*. A vector is a collection of elements each identified by an index. A function for creating vectors is the concatenation function `c(arg1, arg2, ..., argn)`. This function combines an arbitrary number of objects end to end into a single vector.

```
1 > x = c(1,100,-4.5,8)
2 > x
3 [1]   1.0 100.0  -4.5   8.0
```

Line 1 creates a four-dimensional vector whose elements are 1, 100, -4.5, and 8. Next, typing x by itself in line 2 tells R to write out all the elements of x.

Individual elements of a vector are accessed using brackets `[ ]`. For instance, the 3rd element of x is obtained as follows:

```
> x[3]
[1] -4.5
```

Multiple elements can be accessed by using vector arguments, as the following example illustrates.

```
> x = c(1,100,-4.5,8)
> x
[1]   1.0 100.0  -4.5   8.0
> x[c(1,4)]
[1] 1 8
> y = c(1,4)
> x[y]
[1] 1 8
```

More sophisticated ways of subsetting a vector will be discussed in sec. 11.

After a vector is created, any element can be accessed or written, including elements that have not been assigned. A vector has as many elements as the last element that has been assigned. Elements that have not been assigned are automatically assigned `NA`.

```
> rm(z) # remove the vector z
> z[10] # access 10th element of z (which is invalid since z does not exist)
Error: object 'z' not found
> z = 1 # create a 1D vector and assign the first element 1
> z
[1] 1
> z[10] # access the 10th element of z
[1] NA
> z[10] = 4 # assign the 10 element of z to 4
> z
 [1]  1 NA NA NA NA NA NA NA NA  4
```

In the above example, we have used # to insert comments. R will ignore all characters to the right of # on the same line.

4 Functions for Creating Numeric Vectors

An R function is of the form

```
fun.name(arg1,arg2,...)
```

where `fun.name` is the name of the function, and `arg1, arg2, ...` are arguments to the function. We have already met a function in the preceding section, namely the concatenation function `c()`. Two other frequently used functions are `seq` and `rep`. The function `seq`, for "sequence," generates regular sequences of numbers, as illustrated by the following examples:

```
> seq(from=1,to=10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(from=1,to=10,by=2)
[1] 1 3 5 7 9
> seq(from=0,length.out=7,by=0.5)
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
```

Regular sequences occur frequently in data analysis and graphical procedures, so this function will be seen often in future examples. The special case of step size 1 can be handled using the `:` syntax. The sequence also may be in decreasing order, include negative numbers, and be non-integers:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> 3:-7
 [1]  3  2  1  0 -1 -2 -3 -4 -5 -6 -7
> 2.4:9.4
[1] 2.4 3.4 4.4 5.4 6.4 7.4 8.4 9.4
```

The `rep` function, short for "repeat," replicates an object. When the object is a vector, the behavior of `rep` can be controlled with the arguments `each` and `times`, as illustrated in the following examples:

```
> x = c(2,4,6)
> rep(x,times=3)
[1] 2 4 6 2 4 6 2 4 6
> rep(x,each=3)
[1] 2 2 2 4 4 4 6 6 6
```

Of course, the above functions can be combined in an infinite variety of ways to create new vectors:

```
> x = seq(2,8,2)
> x
[1] 2 4 6 8
> y = rep(1:3,4)
> y
 [1] 1 2 3 1 2 3 1 2 3 1 2 3
> z = c(rep(x,2),y)
> z
 [1] 2 4 6 8 2 4 6 8 1 2 3 1 2 3 1 2 3 1 2 3
```

R has a variety of functions for getting properties of vectors, some of the most useful of which are summarized in table 1.

`length(x)`	number of elements in `x`
`min(x)`	smallest element in `x`
`max(x)`	largest element in `x`
`range(x)`	`c(min(x),max(x))`
`sum(x)`	sum of the elements of `x`
`prod(x)`	product of the elements of `x`
`mean(x)`	average of the elements of `x`
`median(x)`	median of the elements of `x`
`sort(x)`	sort the elements of `x` in ascending order

Table 1: Common R functions for extracting properties of a vector

5 Vector Arithmetic

The arithmetic operators for add, subtract, multiply, and divide are `+, -, *, /`, respectively. The power operator is `^`; e.g., `2 ^ 3` is `8`. The modulo operator `%%` gives the remainder when the first argument is divided by the second. Arithmetic operators applied to vectors are performed element-by-element.

```
> 2+3 # add 2 and 3
[1] 5
> 2*3 # multiply 2 and 3
[1] 6
> 10 %% 4 # divide 10 by 4 and give the remainder
[1] 2
> x = c(1,2,3)
> 1/x # divide 1 by each element of x
[1] 1.0000000 0.5000000 0.3333333
> y = c(4,5,6)
> x * y # multiply x and y elementwise
[1]  4 10 18
> x / y # divide x by y elementwise
[1] 0.25 0.40 0.50
> y ^ x # take each element of y to the corresponding power of x
[1]   4  25 216
```

The vectors occurring in the same expression do not need to be the same length. If the vectors have different sizes, *then shorter vectors are reused cyclically until they match the length of the longest vector.* This process is called *recycling*.

```
1 > x = c(1,2,3,4,5,6)
2 > y = c(1,3)
3 > x * y
4 [1]  1  6  3 12  5 18
5 > 2 * y
6 [1] 2 6
7 > z = x * y + 2 * y + 1
8 > z
9 [1]  4 13  6 19  8 25
```

In line 3, the term `x * y` is evaluated by first cyclically repeating the `y` vector three times, yielding the vector `(1,3,1,3,1,3)`, then multiplying this vector by `x` element by element, yielding line 4. Similarly, line 5 is evaluated by repeating 2 twice, yielding the vector `(2,2)`, then multiplying this vector by `y` element wise, yielding line 6.

If one of the smaller vectors is not an integral fraction of the longest vector, then R correctly evaluates the arithmetic expression, but also generates a warning message. This warning message is useful for debugging purposes since many important applications of recycling require integral recycling.

An example that illustrates many of the above features is the following. Consider a vector x_i for $i = 1, 2, \ldots, N$. Compute the unbiased variance estimate

$$\hat{\sigma}_X^2 = \frac{1}{N-1} \sum_{i=1}^{N} (x_i - \hat{\mu}_X)^2 \,, \tag{1}$$

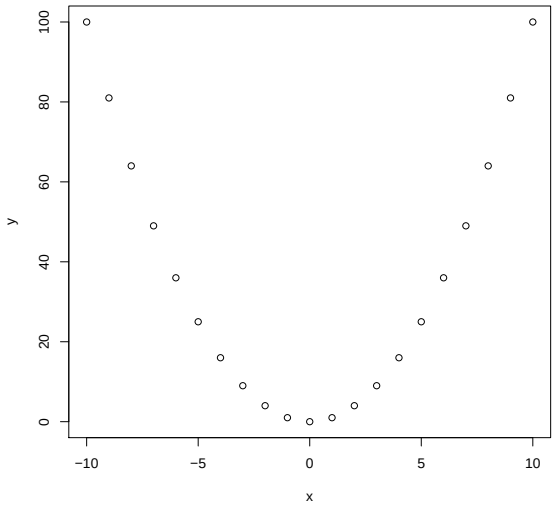
where $\hat{\mu}_X$ is the sample mean of x_i. A solution is shown in line 5 below:

```
1 > set.seed(1)
2 > x = rnorm(5)
3 > x
4 [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
5 > x.var  = sum((x-mean(x))^2)/(length(x)-1)
6 > x.var
7 [1] 0.9235968
8 > var(x)
9 [1] 0.9235968
```

It is important to fully understand how line 5 is evaluated. First, the function `mean(x)` is evaluated, which generates a single number, then this number is repeated five times so that it can be subtracted from the vector `x` element by element. The resulting difference is then squared element-wise and summed. This number is then divided by `length(x)-1`, which is 4 in this example. Line 9 verifies that the variance computed in line 5 matches the variance computed by the function `var(x)`.

6 Plotting X-Y Data

For 2-dimensional data, the main function for graphically illustrating data is `plot()`. This function is illustrated in fig. 1.




```
ntot = 10
x = (-ntot):ntot
y = x^2
plot(x,y)
```

Figure 1:

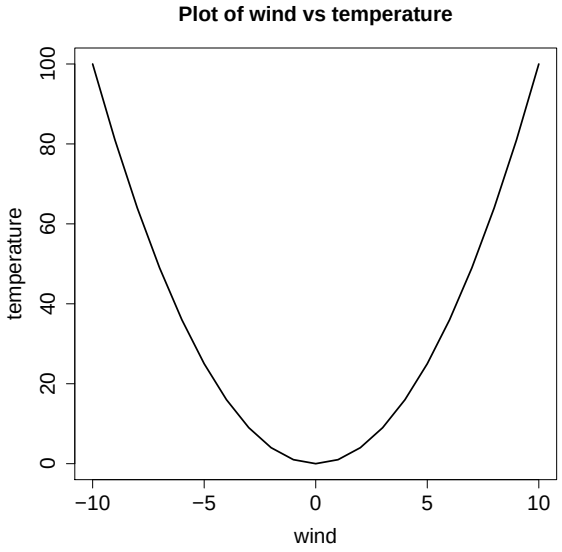
Invariably, you will want to change the look of the plot. Here, we would like to increase the font size. The size of the axis labels can be increased using `par`:

```
par(cex.lab=1.5)
```

This command magnifies the current setting of `cex.lab`, which controls the size of the x and y lables, by 1.5; i.e., the size is increased by 50%. A list of all graphics parameters can be obtained by typing `par()`. The purpose of each parameter is described in the manual page, which can be obtained by typing `?par`. If you know the parameter you want to change, then you can re-set its value as in the above example for `cex.lab`.

In addition to changing the size of the figure, we usually want to add labels, and modify the symbols or the lines used to plot the data. Such changes can be made by including appropriate arguments in `plot`. A list of valid arguments can be generated by typing `?plot.default`. An example of these changes is shown in fig. 2. Hopefully, the result is self-explanatory.

We often want to overlay multiple plots on the same frame. The easiest way is to use the `lines` and `points` commands, which add curves or points to a figure. Another way is to use the `plot` command again, except that you need to type `par(new=TRUE)` before the second plot, otherwise `plot` will delete the previous plot and create a new plot. Also! `plot()` creates a new set of axes




```
ntot = 10; x = (-ntot):ntot; y = x^2
par(cex.lab=1.5,cex.axis=1.5,cex.main=1.5)
plot(x,y,xlab="wind",ylab="temperature",type="l",
  col="black",lwd=2,main="Plot of wind vs temperature")
```

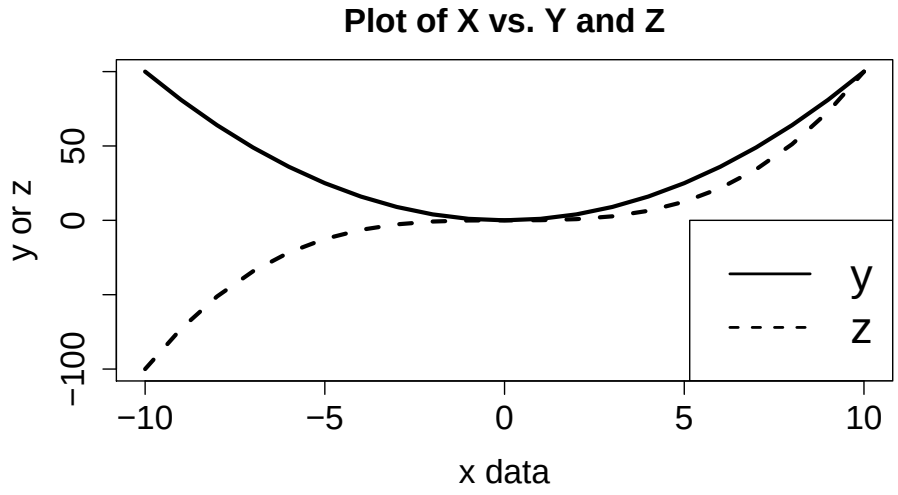
Figure 2:

with each call, whereas you usually want the axes to be the same. Therefore, the axes need to be determined before any plotting and held fixed for subsequent plots using the arguments `xlim` and `ylim`. The appropriate range for the axes can be found using the `range` command. Also, for each subsequent plot, it is unnecessary to re-plot the axes or axis labels. The axes can be suppressed by including `axes=FALSE` in the argument list to `plot()`, and axis labels can be suppressed using blank labels. A legend can be inserted using `legend`. All of these steps are illustrated in fig. 3.

In the end, you often need to generate a graphics file that can be inserted in a paper or report. This can be done by using the `pdf` or postscript commands before the plot commands, and then ending with `dev.off()`, as illustrated below:

```
pdf('figure.pdf',width=8,height=4)
.... ALL PLOT COMMANDS....
dev.off()
```

The above generates a PDF file called *figure.pdf*. To generate the postscript file *figure.eps*, type




```
ntot = 10; x = (-ntot):ntot; y = x^2; z = x^3/10
xrange = range(x)
yrange = range(y,z)
par(mfcol=c(1,1),mar=c(5,5,3,1))
par(cex.lab=1.5,cex.axis=1.5,cex.main=1.5)
plot(x,y,xlab="x data",ylab="y or z",type="l",
  lwd=3,xlim=xrange,ylim=yrange)
par(new=TRUE)
plot(x,z,xlab="",ylab="",type="l",axes=FALSE,
  lwd=3,xlim=xrange,ylim=yrange,lty="dashed")
title(main='Plot of X vs. Y and Z')
legend("bottomright",legend=c("y","z"),
  lwd=2,lty=c('solid','dashed'),cex=2)
```

Figure 3:

```
postscript('figure.eps',horizontal=FALSE,
  onefile=FALSE,height=4,width=8,pointsize=12)
.... ALL PLOT COMMANDS....
dev.off()
```

7 Functions Related to Probability Theory

R contains a variety of functions related to probability theory. For example, typing `rnorm(25)` in the R console tells R to generate 25 random numbers from a (standardized) normal distribution:

```
> rnorm(25)
 [1] -0.62645381  0.18364332 -0.83562861  1.59528080  0.32950777
 [6] -0.82046838  0.48742905  0.73832471  0.57578135 -0.30538839
[11]  1.51178117  0.38984324 -0.62124058 -2.21469989  1.12493092
[16] -0.04493361 -0.01619026  0.94383621  0.82122120  0.59390132
[21]  0.91897737  0.78213630  0.07456498 -1.98935170  0.61982575
```

Lines 2-6 show the response from R. The bracketed numbers `[1], [6], [11], [16], [21]` at the beginning of each line give the index of the first number on that line. For example, `1.51178117` is the 11th random number generated.

To fit the above printout into this document, a shortened line width is used. Also, the precise numbers generated by `rnorm(25)` differ for each execution, so the numbers you obtain will differ from those above. To achieve the same random numbers as above, type `set.seed(1)` just before `rnorm(25)`. This function sets the initial "seed" value for generating pseudorandom numbers, using a procedure whose details are not important at the moment. It is recommended that the seed value be set explicitly for all your programs. This coding practice ensures that the same random numbers will be generated for each execution, a property that facilitates debugging. You should then change the seed value and verify that your results are insensitive to the seed value. Also, to achieve the same line width as used in this introduction, type `options(width=70)`:

```
> options(width=70)
> set.seed(1)
> rnorm(25)
 [1] -0.62645381  0.18364332 -0.83562861  1.59528080  0.32950777
 [6] -0.82046838  0.48742905  0.73832471  0.57578135 -0.30538839
[11]  1.51178117  0.38984324 -0.62124058 -2.21469989  1.12493092
[16] -0.04493361 -0.01619026  0.94383621  0.82122120  0.59390132
[21]  0.91897737  0.78213630  0.07456498 -1.98935170  0.61982575
```

You should now see precisely the same output in your R console as shown above.

More details of a function can be obtained by typing the function name preceded by a question mark, as follows.

```
?rnorm
```

After typing this command, you will see documentation, which may not make much sense at this point. However, one of the lines is

```
rnorm(n, mean = 0, sd = 1)
```

This line states that the function `rnorm` has three arguments, namely `n, mean, sd`. Moreover, the argument `n` is required while the parameters `mean` and `sd` have default values equal to 0 and 1, respectively. Further reading of the documentation reveals that `rnorm(n)` generates `n` random

numbers from a normal distribution with zero mean and unit variance. To draw normal random numbers from a normal distribution with a different mean and standard deviation, say a mean of 2 and standard deviation of 3, we would type the following

```
> set.seed(1)
> rnorm(25,mean=2,sd=3)
 [1]  0.1206386  2.5509300 -0.5068858  6.7858424  2.9885233 -0.4614052
 [7]  3.4622872  4.2149741  3.7273441  1.0838348  6.5353435  3.1695297
[13]  0.1362783 -4.6440997  5.3747928  1.8651992  1.9514292  4.8315086
[19]  4.4636636  3.7817040  4.7569321  4.3464089  2.2236950 -3.9680551
[25]  3.8594772
```

It is not necessary to explicitly state the name of each argument. For instance, `rnorm(25,2,3)` is equivalent to `rnorm(25,mean=2,sd=3)`, because R assumes that the order of the un-named arguments matches the order specified in the manual pages. On the other hand, if the parameters are explicitly named, then they can occur in any arbitrary order. The first un-named argument is assumed to be the first un-specified argument:

```
> set.seed(1)
> rnorm(sd=3,n=25,mean=2)
 [1]  0.1206386  2.5509300 -0.5068858  6.7858424  2.9885233 -0.4614052
 [7]  3.4622872  4.2149741  3.7273441  1.0838348  6.5353435  3.1695297
[13]  0.1362783 -4.6440997  5.3747928  1.8651992  1.9514292  4.8315086
[19]  4.4636636  3.7817040  4.7569321  4.3464089  2.2236950 -3.9680551
[25]  3.8594772
```

R has functions not only for generating random numbers, but also for evaluating various probabilities, such as the cumulative distribution function $P(Z \leq z)$, the probability density function $p(z)$, and the quantile function. The following illustrates the use of these functions:

```
> pnorm(0) # probability that Z < 0
[1] 0.5
> dnorm(0) # probability density at z = 0
[1] 0.3989423
> qnorm(0.975) # the smallest z such that P(Z < z) > 0.975
[1] 1.959964
```

R also has similar functions for other distributions. The most useful for introductory statistics are summarized in table 2.

To evaluate the cumulative distribution, probability density, and quantile of the F distribution, say with parameters `df1=5`, `df2=5`, `ncp=0`, we would use the following commands:

R name	distribution	parameters	defaults
chisq	chi-squared	df, ncp	-,0
f	F	df1, df2, ncp	-, -, 0
norm	normal	mean, sd	0, 1
t	Student's t	df, ncp	-, 0
unif	uniform	min, max	0, 1

Table 2: R functions for common distributions.

```
> pf(0.5,5,10) # probability that F < 0.5, for F-dist with 5 and 10 dof
[1] 0.2299751
> df(0.5,5,10) # density of F at x = 0, for F-dist with 5 and 10 dof
[1] 0.687607
> qf(0.95,5,10) # smallest x such that P(F<x) > 0.95
[1] 3.325835
```

R has a number of functions for numerical computation, including the familiar functions `log, exp, sin, cos, tan, sqrt, abs`. Many other functions can be found by typing `?Math`. All standard functions in R will accept vectors as arguments and evaluate the function element wise.

```
> x = c(1,2,3,4,5)
> sqrt(x)
[1] 1.000000 1.414214 1.732051 2.000000 2.236068
> log(x)
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
> log(x)/sqrt(x)
[1] 0.0000000 0.4901291 0.6342841 0.6931472 0.7197625
```

8 Logical Vectors

In addition to numerical vectors, R uses logical vectors. A logical vector is a vector that takes on the values `TRUE` or `FALSE` (or `NA` when missing data exists, as explained later). The abbreviations `T` and `F` can be used for assignments, although this is generally discouraged. Logical vectors can be created using `c` and `rep`.

```
> x = c(T,F,T)
> x
[1]  TRUE FALSE  TRUE
> rep(x,2)
[1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE
> rep(x,each=2)
[1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE
```

Logical vectors also can be created using the *comparison operators* summarized in table 3. Note that equality is tested using the double equal sign ==, which should be sharply distinguished from the

single equal sign =, which is used for assignments. If the comparison operator is applied to vectors, the comparison operator is applied element-by-element.

$<$	"less than"
$>$	'greater than"
$<=$	'less than or equal to"
$>=$	'greater than or equal to"
$==$	'equal to"
$!=$	'not equal to"

Table 3: Comparison operators in R

```
1 > 2 > 4
2 [1] FALSE
3 > 2 < 4
4 [1] TRUE
5 > x = c(1,2,3)
6 > y = c(-1,3,-5)
7 > x < y
8 [1] FALSE  TRUE FALSE
```

Vectors can be assigned the value of comparison operators.

```
1 > x = c(1,2,3)
2 > y = c(0,2,4)
3 > z = x == y
4 > z
5 [1] FALSE  TRUE FALSE
6 > w = x <= y
7 > w
8 [1] FALSE  TRUE  TRUE
```

Line 3 may confuse you if you haven't caught the difference between the assignment = and comparison ==. To be clear, the comparison operator == is evaluated first, so `x == y` yields the value `FALSE TRUE FALSE`, which is then assigned to `z`.

Logical expressions can be combined using the logical operators `&`, `|`, `!`, which correspond to the logical *and, or,* and *not*.

```
> x = c(1,2,3)
> xge2 = x >= 2
> xge2
[1] FALSE  TRUE  TRUE
> !xge2
[1]  TRUE FALSE FALSE
> y = c(-1,3,-5)
> yg1 = y > 1
> yg1
[1] FALSE  TRUE FALSE
> xge2 & yg1
[1] FALSE  TRUE FALSE
```

Two useful functions for summarizing logical vectors are `any` and `all`, which take logical vectors and return a single `TRUE` or `FALSE`, depending on whether any or all elements are `TRUE`.

```
> x = c( -1, 8, 5, -3)
> any(x > 0) # is any element of x is positive?
[1] TRUE
> all(x > 0) # are all elements of x are positive?
[1] FALSE
```

When arithmetic operators are applied to logical variables, `TRUE` and `FALSE` are coerced into the integers 1 and 0, respectively, then the arithmetic operator is applied. This feature is especially useful for determining the fraction of cases that satisfy a condition.

```
> set.seed(1)
> x = rnorm(10)
> x
 [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
 [7]  0.4874291  0.7383247  0.5757814 -0.3053884
> x < 0
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
> mean(x<0) # determine the fraction of cases in which x < 0
[1] 0.4
```

9 Character Vectors

Character variables are contained in single quotes or double quotes. Double quotes is preferred to avoid confusion with other symbols in certain fonts. Character vectors may be concatenated using `c()`.

```
> x = c("mike","tim","tony")
> x
[1] "mike" "tim"  "tony"
> x[2]
[1] "tim"
```

A more sophisticated concatenation function is `paste()`, which coerces its arguments to character strings and then concatenates them, separated by the string `sep`, whose default value is `" "`. If the arguments are vectors, then `paste` concatenates the vectors term-by-term. If the vectors have different sizes, then shorter vectors are reused cyclically until all elements of the longest vector have been concantenated.

```
> x = c("mike","tim","tony")
> paste(x,3)
[1] "mike 3" "tim 3"  "tony 3"
> paste(x,1:3)
[1] "mike 1" "tim 2"  "tony 3"
> paste(x,1:3,sep=" is number ")
[1] "mike is number 1" "tim is number 2"  "tony is number 3"
```

To concatenate vectors into a single string, use the `collapse` argument:

```
> x = c("mike","tim","tony")
> paste(x,1:3,collapse=" ")
[1] "mike 1 tim 2 tony 3"
> paste(x,1:3,sep=" is number ",collapse = " and ")
[1] "mike is number 1 and tim is number 2 and tony is number 3"
```

Numbers and logicals can be coerced into character strings using `as.character`. The function `is.character` is used to determine whether a variable is a character. A number of functions for parsing strings also exist, including `substr, nchar, strsplit, sub, grep`.

10 Modes

Each object in R has a mode. The most common modes are `"null", "logical", "numeric", "complex", "character", "list"`. Many operations that are applied to groups of variables require the variables to have the same mode, otherwise the variables are *coerced* into the required format. For instance, all items in a vector must have the same mode. If the concatenation function is applied to objects with differing modes, such as a mix of numeric and character strings, then the objects are coerced into the same mode according to certain rules (as spelled out in the documentation page). The mode of `x` can be ascertained using `mode(x)`. To test whether a variable has a certain mode, use functions like `is.null, is.logical, is.character`. To coerce an object into a certain mode, use functions like `as.null, as.logical, as.character`.

11 Vector Indexing

A very powerful feature of R is that particular elements of a vector can be extracted or assigned using the format `x[index]`, where `index` is a vector. This process is called *indexing*. The value of `index` can take on different forms, each of which has important applications.

1. `index` can be a vector of positive integers, in which case `x[index]` will access the elements of `x` corresponding to the indices given by `index`.

```
1  > x = c(18,-22,34,11,-25,33)
2  > index = 1 # extract the 1st element of x
3  > x[index]
4  [1] 18
5  > index = 1:3 # extract the first three elements of x
6  > x[index]
7  [1]  18 -22  34
8  > index = c(1,5,5,3) # extract the 1st, 5th, 5th, 3rd elements of x
9  > x[index]
10 [1]  18 -25 -25  34
11 > y = c(1,3,2) # any vector can be used (e.g., 'y' instead of 'index')
12 > x[y]
13 [1]  18  34 -22
```

   Note that an index can be repeated more than once and can appear out of order, as in line 8. Indexing also can be used to re-assign the value of `x`, as shown below.

```
1  > x = c(18,-22,34,11,-25,33)
2  > x[1] = -13 # assign the 1st element of x to -3
3  > x
4  [1] -13 -22  34  11 -25  33
5  > index = 3:6
6  > x[index] = c(0,1) # assign elements 3-6 of x to the vector (0,1,0,1)
7  > x
8  [1] -13 -22   0   1   0   1
```

   Line 2 reassigns the first element of `x` to -3. Line 6 re-assigns lines 3-6 to the vector `(0,1,0,1)` (where the vector `c(0,1)` is recycled).

2. `index` can be a vector of logical values, in which case `x[index]` will access the elements for which the corresponding value of `index` is `TRUE`. This form of indexing is very useful for extracting values of a vector that satisfy certain conditions.

```
> rain   = c(21,14,32,26)
> index  = c(FALSE,TRUE,FALSE,TRUE)
> rain[index]
[1] 14 26
> temp   = c(13,22,15,24)
> index = rain < 23
> index
[1]  TRUE  TRUE FALSE FALSE
> rain[index]
[1] 21 14
> index = rain < 23 & temp >= 14
> index
[1] FALSE  TRUE FALSE FALSE
> rain[index]
[1] 14
```

3. `index` can be a vector of negative integers, in which case `x[index]` will produce all elements of `x` *except* the corresponding elements of `-index`. This feature is especially useful for leave-K-out cross validation procedures.

```
> x = c(18,-22,34,11,-25,33)
> index = -3 # extract all BUT the 3rd element
> x[index]
[1]  18 -22  11 -25  33
> index = -c(3,5) # extract all BUT the 3rd and 5th element
> index
[1] -3 -5
> x[index]
[1]  18 -22  11  33
```

4. `index` can be character strings. This feature will be discussed in sec. ?.

12 Missing Values

In some cases, certain elements of a vector are not known, perhaps because the corresponding observations are "missing" or "not available." The special value `NA` can be assigned to such elements. Any operation involving `NA` is set to `NA`.

```
> x = c(2, -4, NA, 3)
> x
[1]  2 -4 NA  3
> y = rep(1,4)
> y
[1] 1 1 1 1
> x + y
[1]  3 -3 NA  4
```

Different functions treat NA in different ways, often depending on certain arguments. This fact is illustrated in the following example.

```
1 > x = c(2, -4, NA, 3)
2 > sum(x)
3 [1] NA
4 > sum(x,na.rm=TRUE)
5 [1] 1
6 > mean(x,na.rm=TRUE)
7 [1] 0.3333333
```

Line 1 creates a four-element vector, with one element assigned to NA. Line 3 shows that the sum of the elements of x is assigned NA, because one of the elements is NA. Lines 5 and 7 give the sum and mean of the elements of x, after all NA's have been removed.

The elements of vector that are NA can be determined using the is.na function. In particular, is.na(x) generates a logical vector such that TRUE indicates that the corresponding element in x is NA. Note that the logical expression x == NA is NA for any x, reflecting the fact that testing equality with respect to an unknown value is not defined.

It is good programming practice to initialize all vectors with NA; e.g., using x = rep(NA,10). For instance, if all elements of the initialized vector are expected to be assigned a numerical value, then finding a vector containing NA at the end of a code indicates a programming bug.

Other forms of "missing" values are produced by arithmetic operations. For instance, 1/0 is assigned Inf, while 0/0 is assigned NaN. The functions is.infinite and is.nan are used to determine whether an element is Inf and NaN, respectively.

Another value worth mentioning is NULL, which is essentially an "empty" object. Concatenating a set of vectors with the NULL value results in just the vectors without the NULL value; in contrast, concatenating a set of vectors with NA will include NA values.

```
1 > c(1,2,3,NULL)
2 [1] 1 2 3
3 > c(1,NULL,2,NA)
4 [1]  1  2 NA
```

The command is.null(x) is TRUE if x is assigned to NULL. The NULL value has several uses which will be explained as needed.

13 Loops

Many calculations need to be performed repeatedly with different calculations depending on intermediate results. Such calculations often require *control-flow* constructs. We have avoided discussing control-flow for a very good reason: R is efficient at evaluating *vectorized* expressions, so it is important to perform iterative calculations in vectorized form if at all possible. If this is not possible, then the most common control-flow construct is the for loop construction, which has the form

```
for ( i in vector ) expr
```

where `i` is a variable that is assigned to each element of `vector` successively, and `expr` is an expression that is evaluated once for each assignment. Usually, `expr` depends on `i`. As a simple example, let us add the vectors `x` and `y` together to form `z`. The vectorized approach is to type `z = x + y`, while an equivalent approach using the `for`-loop is

```
for ( i in 1:ntot) z[i] = x[i] + y[i]
```

In this example, the statement `for ( i in 1:ntot)` tells R to execute the expression following it for `i=1`, then `i=2`, etc. until `i = ntot`. Thus, R first evaluates the expression `z[1] = x[1] + y[1]`, followed by `z[2] = x[2] + y[2]`, and so on until `z[ntot] = x[ntot] + y[ntot]`.

Although the above loop is equivalent to the vectorized expression `z = x + y`, the two approaches have very different execution times, which can be shown using the command `system.time`.

```
> ntot = 1000000
> x = rnorm(ntot)
> y = rnorm(ntot)
> z = numeric(length=ntot)
> system.time( z <- x + y)
   user  system elapsed
  0.012   0.002   0.014
> system.time(for ( i in 1:ntot) z[i] = x[i] + y[i])
   user  system elapsed
  3.006   0.010   3.000
```

The above shows that the vectorized calculation takes only 0.014 seconds, while the `for` loop takes 3 second– over 200 times longer. This simple example should make clear that vectorized calculations are preferred over `for`-loops, when possible.

A subtle point in the above example should be emphasized: the vector `z` was created before the `for`-loop was executed. The reason for creating the vector `z` is that the brackets `[ ]` cannot extract an element of a vector unless the vector itself exists, as the following example illustrates:

```
> rm(x,y,z)
> ntot = 10
> x = rnorm(ntot)
> y = rnorm(ntot)
> for ( i in 1:ntot) z[i] = x[i] + y[i]
Error in z[i] = x[i] + y[i] : object 'z' not found
```

As the error message indicates, the `for`-loop failed because `z` did not exist: the `rm` command in line 1 "removed" the vector `z` from memory. To avoid this error, the variable `z` needs to exist before the

loop. One way to do this is to use the command `z = numeric(length=ntot)`, which creates a numeric vector of length `ntot` with all elements equal to zero. However, a good habit is to initialize vectors with `NA`:

```
> z = numeric(length=10)
> z
 [1] 0 0 0 0 0 0 0 0 0 0
> z = as.numeric(rep(NA,10))
> z
 [1] NA NA NA NA NA NA NA NA NA NA
```

Commands may be grouped with braces and separated by either semicolons or new lines. As a simple example, we use a `for` loop to calculate the correlation coefficient between two independent random variables 1000 times

```
> set.seed(1)
> ntot = 1000; ndim = 10
> xy.cor = as.numeric(rep(NA,ntot))
> for ( i in 1:ntot) {
+         x = rnorm(ndim); y = rnorm(ndim)
+         xy.cor[i] = cor(x,y)
+ }
> mean(xy.cor)
[1] 0.01644176
> var(xy.cor)
[1] 0.1105325
```

Note the prompt + inside the `for`-loop, which indicates that the command is not syntactically complete. In general, if a command is not complete at the end of a line, then R gives the + prompt on each subsequent line until the command is syntactically complete.

Other looping constructs include `repeat` and `while`.

The object used for looping need not be numeric. For instance, the following example loops over character strings:

```
> iall = c("red","blue","green")
> for ( i in iall ) print(i)
[1] "red"
[1] "blue"
[1] "green"
```

Loops over character objects are especially useful when indexing objects by character name, or when only text or graphic output is desired with each iteration.

The command `seq(along.with = x)` generates the sequence `1,2, ... length(x)`, which is handy for generating integer indices for looping.

In some cases, we want to build a vector with each iteration, for instance using the concatenation function. The tricky part is starting the process, since the very first time the loop is executed the vector needs to exist before it can be concatenated. This is where the `NULL` value becomes useful:

```
1 > rm(x)
2 > for ( i in 1:10) x = c(x,i)
3 Error: object 'x' not found
4
5 > x = NULL
6 > for ( i in 1:10) x = c(x,i)
7 > x
8  [1]  1  2  3  4  5  6  7  8  9 10
```

Line 3 shows that the loop failed because the concatenation function attempted to concatenate `x` and `i` when `x` did not exist (it was "removed" from memory by line 1). Lines 5-8 show that if `x` is initialized with `NULL`, then concatenating it with `i` leaves just `i`.

14 Conditional Execution

R has an `if` construct of the form

```
1 if ( condition ) expr_1
```

which executes the expression `expr_1` only if `condition` is `TRUE`. The `if-else` construct can be used to execute one expression when the condition is true and another statement when the condition is false:

```
1 if ( condition ) expr_1 else expr_2
```

The expressions `expr_1` and `expr_2` could contain further `if-else` constructs.

A vectorized version of `if-else` is `ifelse`, which takes the form

```
1 ifelse ( condition, x, y)
```

and returns a vector of the length of the longest argument, with elements `x[i]` if `condition[i]` is true, otherwise `y[i]`.

15 Lists

In certain complex calculations, it is useful to return a wide variety of variables. For instance, in fitting a linear model, we may want the value of the fitted parameters, their uncertainties, and perhaps

text that describes the data. Similarly, a data set may contain numeric, logical, and character values that need to be grouped together. The function `list` can be used to group objects of different modes together. Each object in the list is called a *component*.

As an example, suppose we have five temperature measurements in units of Kelvin, and for each measurement there is a logical variable indicating cloudiness. This data set could be grouped into a list as follows.

```
1  > temp   = c(275, 279, 285, 300, 294)
2  > cloudy = c(T,F,F,T,F)
3  > tunits = "Kelvin"
4  > stat.data = list(temp,cloudy,tunits)
5  > stat.data
6  [[1]]
7  [1] 275 279 285 300 294
8
9  [[2]]
10 [1]  TRUE FALSE FALSE  TRUE FALSE
11
12 [[3]]
13 [1] "Kelvin"
```

Line 4 creates an object called `stat.data` that groups together the data defined in lines 1-3. Line 5 asks R to write out the contents of the list. The resulting output shows that the list has three components, indicated by the double bracket `[[...]]`. Below each component, R writes out the contents of the respective component. Components of a list are automatically numbered and can be accessed using the double brackets `[[...]]`. If the component is a vector, individual elements of the vector can be accessed using subscripting.

```
1  > stat.data[[1]]
2  [1] 275 279 285 300 294
3  > stat.data[[1]][3]
4  [1] 285
5  > stat.data[[2]][c(2,3)]
6  [1] FALSE FALSE
```

Individual components of a list also can be named and accessed using these names in a variety of ways, as illustrated below

```
1  > stat.data = list(temperature=temp,cloudiness=cloudy,units=tunits)
2  > stat.data$temperature
3  [1] 275 279 285 300 294
4  > stat.data[[1]]
5  [1] 275 279 285 300 294
6  > stat.data[["temperature"]]
7  [1] 275 279 285 300 294
8  > stat.data[1]
9  $temperature
10 [1] 275 279 285 300 294
```

The use of names can be very helpful when the list contains many components– you might forget which number corresponds to a particular component. Note the difference in outputs between `stat.data[[1]]` and `stat.data[1]` after lines 4 and 8: the operator `[[...]]` extracts the first object in the list and produces a result with the same mode as the object, whereas the operator `[...]` is a general subscripting operator and therefore returns a sublist of the original list.

```
1  > mode(stat.data[[1]])
2  [1] "numeric"
3  > mode(stat.data[1])
4  [1] "list"
5  > stat.data[c(1,3)]
6  $temperature
7  [1] 275 279 285 300 294
8
9  $units
10 [1] "Kelvin"
11
12 > stat.data[1]$temperature
13 [1] 275 279 285 300 294
```

16 Data Frames

A data frame is a special case of a list in which each component has the same number of elements. For most purposes, a data frame can be interpreted as a matrix with a fixed number of rows and columns, but differing from a matrix in that the columns may have different modes. Data frames can be constructed using the function `data.frame`:

```
1  > temp   = c(275, 279, 285, 300, 294)
2  > cloudy = c(T,F,F,T,F)
3  > tunits = "Kelvin"
4  > stat.data = data.frame(temp,cloudy,tunits)
5  > stat.data
6    temp cloudy tunits
7  1  275   TRUE Kelvin
8  2  279  FALSE Kelvin
9  3  285  FALSE Kelvin
10 4  300   TRUE Kelvin
11 5  294  FALSE Kelvin
```

Note that the component `tunits` was recycled to complete the matrix. Also, the name of each column, listed in line 6, is taken from the name of the variable. These names can be set by the user, similar to the way the names are set in lists. The components of the data frame can be accessed in a variety of ways, as the following example illustrates:

```
1  > stat.data = data.frame(temperature=temp,cloudy,tunits)
2  > stat.data
3    temperature cloudy tunits
4  1         275   TRUE Kelvin
5  2         279  FALSE Kelvin
6  3         285  FALSE Kelvin
7  4         300   TRUE Kelvin
8  5         294  FALSE Kelvin
9  > stat.data[1]
10   temperature
11 1         275
12 2         279
13 3         285
14 4         300
15 5         294
16 > stat.data["temperature"]
17   temperature
18 1         275
19 2         279
20 3         285
21 4         300
22 5         294
23 > stat.data[["temperature"]]
24 [1] 275 279 285 300 294
25 > stat.data$temperature
26 [1] 275 279 285 300 294
```

Many data sets are stored in files with a fixed number of rows and columns. A convenient way to read these files and load the data into a data frame is to use the command `read.table()`.

Data frames can be augmented using the commands `cbind` and `rbind`, which stand for 'column-bind' and 'row-bind'.

```
> stat.data = data.frame(temperature=temp,cloudy,tunits)
> precip    = c(1,0.5,0.3,2.1,1.4)
> stat.data = cbind(stat.data,precip)
> stat.data
  temperature cloudy tunits precip
1         275   TRUE Kelvin    1.0
2         279  FALSE Kelvin    0.5
3         285  FALSE Kelvin    0.3
4         300   TRUE Kelvin    2.1
5         294  FALSE Kelvin    1.4
> stat.data = rbind(stat.data,c(288,TRUE,'Kelvin',0.8))
> stat.data
  temperature cloudy tunits precip
1         275   TRUE Kelvin      1
2         279  FALSE Kelvin    0.5
3         285  FALSE Kelvin    0.3
4         300   TRUE Kelvin    2.1
5         294  FALSE Kelvin    1.4
6         288   TRUE Kelvin    0.8
```

Data frames also can be truncated using the usual subscripting with negative integers:

```
> stat.data[-3]
  temperature cloudy precip
1         275   TRUE      1
2         279  FALSE    0.5
3         285  FALSE    0.3
4         300   TRUE    2.1
5         294  FALSE    1.4
6         288   TRUE    0.8
```

Elements of a data frame can be extracted in a wide variety of ways, as illustrated below.

```
> stat.data[2,]
  temperature cloudy tunits precip
2         279  FALSE Kelvin    0.5
> stat.data[2,][3]
  tunits
2 Kelvin
> stat.data[2,][[3]]
[1] Kelvin
Levels: Kelvin
> stat.data[2,3]
[1] Kelvin
Levels: Kelvin
> stat.data[,2]
[1] "TRUE"  "FALSE" "FALSE" "TRUE"  "FALSE" "TRUE"
> stat.data[[2]][3]
[1] "FALSE"
```

Note that lines 9 and 12 introduce another form of output, namely `Levels`. Characters in a data frame are interpreted as "levels" in Analysis of Variance and other statistical procedures. To suppress this action, use `stringsAsFactors=FALSE`:

```
> stat.data = data.frame(temperature=temp,cloudy,tunits)
> stat.data[2,3]
[1] Kelvin
Levels: Kelvin
> stat.data = data.frame(temperature=temp,cloudy,tunits,stringsAsFactors=FALSE)
> stat.data[2,3]
[1] "Kelvin"
```

17 Functions

In addition to supplying standard functions like `c(), mean(), sin()`, R allows the user to create functions. User-created functions provide a powerful way of consolidating calculations, especially if the function is simple to use. A function is defined by an assignment statement of the form

```
function_name = function(arg_1, arg_2, ... ) expression
```

After this assignment, the function may be called using the command `function_name(arg_1, arg_2, ...)`, where `arg_1, arg_2, ...` are expressions giving the value of the arguments. The value of the function, which may be any mode, is the result of the last expression in the group evaluated. To illustrate, we take the example in sec. 13 for computing realizations of the correlation coefficient for random data, and write it as a function:

```
> cor.random = function(ntot,ndim) {
+         set.seed(1)
+         xy.cor = as.numeric(rep(NA,ntot))
+         for ( i in 1:ntot) {
+                 x = rnorm(ndim); y = rnorm(ndim)
+                 xy.cor[i] = cor(x,y)
+         }
+         xy.cor
+ }
>
> ntot = 1000; ndim = 10
> z = cor.random(ntot,ndim)
> mean(z)
[1] 0.01644176
> var(z)
[1] 0.1105325
```

It is possible to write functions that have *optional arguments*. Optional arguments are indicated by `= expr` in the function definition, where `expr` is an expression. For instance,

```
1 f = function(x,y=2) {...}
```

defines a function with arguments `x` and `y`, where `y` will be assigned the value 2 if the user does not specify `y`. Thus, `f(1,3)` calls the function `f` with the arguments `x=1, y =3`, while `f(1)` calls the function with the arguments `x=1, y=2`. As a further illustration, consider the function `cor.random` defined above. Suppose we want to change the seed value from 1 to 2 in a particular call, but for general calls, we would prefer not to specify this argument, in which case the seed value should automatically be assigned to 1. A function that does this is the following:

```
1  > cor.random = function(ntot,ndim,iseed=1) {
2  +         set.seed(iseed)
3  +         xy.cor = as.numeric(rep(NA,ntot))
4  +         for ( i in 1:ntot) {
5  +                 x = rnorm(ndim); y = rnorm(ndim)
6  +                 xy.cor[i] = cor(x,y)
7  +         }
8  +         xy.cor
9  + }
10 > mean(cor.random(ntot,ndim))
11 [1] 0.01644176
12 > mean(cor.random(ntot,ndim,1))
13 [1] 0.01644176
14 > mean(cor.random(ntot,ndim,2))
15 [1] 0.01491169
```

In lines 10 and 12, the function is called with the argument `iseed=1`. In line 14, the function is called with the argument `iseed=2`, which is why it gives a slightly different result.

The next point is very important: functions can access objects that were defined at the time the function comes into existence, even if these objects do not appear as arguments:

```
1  > d = 10
2  > e = 20
3  > dumf = function(f) print(f+e)
4  > dumf(d)
5  [1] 30
6  > e = 30
7  > dumf(d)
8  [1] 40
9  > rm(e)
10 > dumf(d)
11 Error in print(f + e) : object 'e' not found
```

Line 3 creates a function named `dumf` that prints the sum `f+e`, where `f` is an argument and `e` is a variable that existed at the time `dumf` was created. Line 4 calls `dumf` with the argument `d`, which has been assigned the value 10. Since the value of `e` when the function was called is 20, the result is $10 + 20 = 30$. If the value of `e` is changed after the function is defined, a subsequent function call will use the latest value of the object, as illustrated in lines 6-8. If `e` is removed, then a subsequent function call produces an error.

For most statistical applications, we strongly advise you to *avoid* using objects defined outside the function definition. Instead, this feature should be reserved for advanced applications. Extensive use of this feature requires understanding the concepts of *scope* and *environment*, which are beyond the scope of this introduction.

18 A First Example Analyzing Data

We now show an example in the context of a real data set. Specifically, we use an index called the Pacific Decadal Oscillation (PDO) index, which is a particular linear combination of sea surface temperatures in the Pacific Ocean poleward of 20N.[1]. The January-March mean value of this index is plotted in fig. 4. A glance at the figure suggests a difference in variability across 1976. The goal of this example is to answer the question "has the variability of the PDO changed in recent decades?"

To address the above question, first download the file `PDO.latest.txt` from the website[1]. Because the file format is not standard (e.g., blank columns, metadata at the top and bottom, asterisks next to some data values, and a blank line between header and data), R needs a little help reading the file. To read this data file, put it in a directory, say `/data/indices/`, then type in the R console:

```
1 iyst.read    = 1900
2 iynd.read    = 2016
3 nyrs.read    = iynd.read - iyst.read + 1
4 dir.indices  = '/data/indices/'
5 fdata        = paste(dir.indices,'PDO.latest.txt',sep='')
6 header.pdo   = scan(fdata,what='character()',skip=32,nlines=1,quiet=TRUE)
7 index.pdo    = read.table(fdata,col.names=header.pdo,skip=34,nrows=nyrs.read)
8 if (index.pdo[1,1] != iyst.read) stop('first year is not 1900, as expected')
9 index.pdo[,1] = iyst.read:iynd.read
```

Assuming no error messages appear, you can then look at the data by typing `index.pdo`. The output should look something like this:

[1] for more details and data, see http://jisao.washington.edu/pdo/

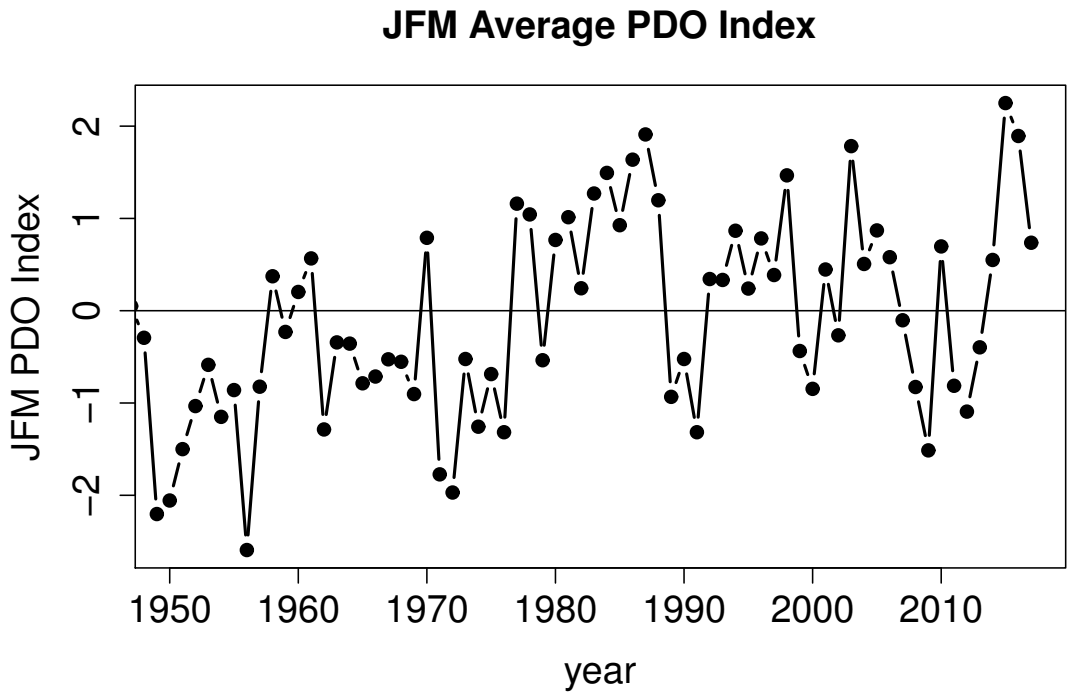


Figure 4: The January-March mean Pacific Decadal Oscillation (PDO) index.

```
> index.pdo
     YEAR   JAN   FEB   MAR   APR   MAY   JUN   JUL   AUG   SEP  ...
1    1900  0.04  1.32  0.49  0.35  0.77  0.65  0.95  0.14 -0.24  ...
2    1901  0.79 -0.12  0.35  0.61 -0.42 -0.05 -0.60 -1.20 -0.33  ...
3    1902  0.82  1.58  0.48  1.37  1.09  0.52  1.58  1.57  0.44  ...
4    1903  0.86 -0.24 -0.22 -0.50  0.43  0.23  0.40  1.01 -0.24  ...
...
116 2015  2.45  2.30  2.00  1.44  1.20  1.54  1.84  1.56  1.94  ...
117 2016  1.53  1.75  2.40  2.62  2.35  2.03  1.25  0.52  0.45  ...
```

We will consider only the January-March mean PDO index after 1950. The following commands will compute such an index. Be sure you understand each line:

```
## DEFINE METADATA
season  = 'JFM'
iyst    = 1950
if ( season == 'JFM') npic.season = 1:3

## COMPUTE SEASONAL MEANS
pdo   = rowMeans(index.pdo[,npic.season+1])

## SPECIFY YEAR
year  = index.pdo[,1]

## USE ONLY YEARS GREATER THAN OR EQUAL TO 'IYST'
npic = year >= iyst
pdo  = pdo[npic]
year = year[npic]
nyrs = length(year)
```

After the above commands have been executed, then R will have assigned the variables `pdo, year, nyrs`. Using the command `cbind` (for 'column bind'), the variables `year` and `pdo` should look like this:

```
> cbind(year,pdo)
      year        pdo
 [1,] 1950 -2.0566667
 [2,] 1951 -1.5000000
 [3,] 1952 -1.0333333
 [4,] 1953 -0.5866667
...
[66,] 2015  2.2500000
[67,] 2016  1.8933333
```

To create a figure, type:

```
par(mar=c(5,5,4,3),cex.lab=1.5,cex.axis=1.5,cex.main=1.5)
plot(year,pdo,type='b',pch=19,lwd=2,xlab="year",ylab="JFM PDO Index")
title(main=paste(season,'Average PDO Index'))
abline(h=0)
```

This figure should be consistent with fig. 4.

No we investigate whether the variability of the PDO has changed in recent decades. Variability can be measured in many ways, but for Gaussian variables the most natural measure is *variance*. Thus, we re-frame the question as: test the hypothesis that the PDO index during 1950-1977 and 1978-2016 were drawn from Gaussian distributions with the same variance. We will write a function that accepts data sets as vector arguments, performs the equality of variance test described in chapter 2, and returns a list variable with relevant information for the test. We will call the function `var.equal.test`. Do not name it `var.test`, as this is a pre-defined function in R. As explained in chapter 2, we need

to compute the sample variances of the two data sets, take their ratio such that the larger variance is over the smaller variance, and compare with the appropriate critical values of an F distribution. These steps are completed by the following function:

```
var.equal.test = function(data1,data2,alpha=0.05) {
### THIS FUNCTION TESTS EQUALITY OF VARIANCE OF TWO
### IID NORMALLY DISTRIBUTED RANDOM VARIABLES
###
# INPUT:
#   DATA1: [N1]-DIMENSIONAL VECTOR OF DATA
#   DATA2: [N2]-DIMENSIONAL VECTOR OF DATA
#   ALPHA: SIGNIFICANCE LEVEL OF THE TEST (DEFAULT = 5%)
# OUTPUT LIST:
#   F.MAX: RATIO OF VARIANCE, CONSTRUCTED TO BE GREATER THAN 1
#   F.CRIT: THE UPPER CRITICAL THRESHOLD OF SIGNIFICANCE
#   PVAL: P-VALUE OF THE F.MAX RATIO
#   VAR1: UNBIASED ESTIMATE OF THE VARIANCE OF DATA1
#   VAR2: UNBIASED ESTIMATE OF THE VARIANCE OF DATA2
#   RATIO: VAR1/VAR2

n1    = length(data1); n2    = length(data2)
mean1 = sum(data1)/n1; mean2 = sum(data2)/n2
var1  = sum((data1-mean1)^2)/(n1-1)
var2  = sum((data2-mean2)^2)/(n2-1)

if ( var1 > var2) {
  f.max  = var1/var2
  f.crit = qf(alpha/2,n1-1,n2-1,lower.tail=FALSE)
  pval   = 2*pf(f.max,n1-1,n2-1,lower.tail=FALSE)
} else {
  f.max  = var2/var1
  f.crit = qf(alpha/2,n2-1,n1-1,lower.tail=FALSE)
  pval   = 2*pf(f.max,n2-1,n1-1,lower.tail=FALSE)
}

ratio       = f.max

list(f.max=f.max,f.crit=f.crit,pval=pval,var1=var1,var2=var2,ratio=ratio)
}
```

Applying this function to the data will look something like this:

```
### SPLIT DATA INTO TWO PARTS
nbreak = which( year == 1977)
pdo1 = pdo[1:nbreak]
pdo2 = pdo[(nbreak+1):nyrs]

## TEST EQUALITY OF VARIANCE
source('var.equal.test.R')
print(var.equal.test(pdo1,pdo2))
$f.max
[1] 1.276986

$f.crit
[1] 2.079678

$pval
[1] 0.511863

$var1
[1] 0.7420675

$var2
[1] 0.9476096

$ratio
[1] 1.276986
```

These results show that the ratio of the largest variance over the smallest variance is 1.28, which is less than the 5% critical value of 2.1. Consistent with this, the p-value is greater than 5%. Therefore, the ratio of variance is not sufficiently different from one to reject the null hypothesis of equal variance. Thus, the answer to the question "has the variability of PDO changed in recent decades?" is "the observed change is not strong enough to reject the hypothesis of no change in variance."

As mentioned above, R has a built-in function called `var.test` that performs the equality-of-variance test. This is illustrated in the following:

```
> var.test(pdo2,pdo1)

        F test to compare two variances

data:  pdo2 and pdo1
F = 1.277, num df = 38, denom df = 27, p-value = 0.5119
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.6140306 2.5438093
sample estimates:
ratio of variances
          1.276986
```

The above results are consistent with the results from our function.

19 Self-Test

Try to perform the following common applications before looking at the answer:

- A vector `x` contains "missing values" that are identified with the value -99.99. Set these missing values to `NA`.

```
> x = c(1,3,-99.99,8,4,3,-99.99)
> x[ x== -99.99] = NA
> x
[1]  1  3 NA  8  4  3 NA
```

- Given the vector `x = c(2,4,6,8,10)`, insert the number '5' between 4 and 6.

```
> x = c(2,4,6,8,10)
> x = c(x[1:2],5,x[3:5])
> x
[1]  2  4  5  6  8 10
```

20 Quick Examples of More Complicated Tasks

- Find the maximum value along each row of a matrix. There are two relevant functions: `apply` or `pmax`. Also, the `do.call` function needs to be used in combination with `pmax`. In general, the `pmax` function is faster, often by a lot.

```
> ntrials = 100000
> mset    = 100
> z       = rnorm(ntrials*mset)
> dim(z)  = c(ntrials,mset)
> system.time(do.call(pmax,data.frame(z)))
   user  system elapsed
  0.488   0.247   0.728
> system.time(apply(z,1,max))
   user  system elapsed
  1.016   0.036   1.045
```