2. An Overview of R

2.1 The Uses of R

2.1.1 R may be used as a calculator.

R evaluates and prints out the result of any expression that one types in at the command line in the console window. Expressions are typed following the prompt (>) on the screen. The result, if any, appears on subsequent lines

```
> 2+2
[1] 4
> sqrt(10)
[1] 3.162278
> 2*3*4*5
[1] 120
> 1000*(1+0.075)^5 - 1000 # Interest on $1000, compounded annually
[1] 435.6293
>                         # at 7.5% p.a. for five years
> pi  # R knows about pi
[1] 3.141593
> 2*pi*6378 #Circumference of Earth at Equator, in km; radius is 6378 km
[1] 40074.16
> sin(c(30,60,90)*pi/180) # Convert angles to radians, then take sin()
[1] 0.5000000 0.8660254 1.0000000
```

2.1.2 R will provide numerical or graphical summaries of data

A special class of object, called a *data frame*, stores rectangular arrays in which the columns may be vectors of numbers or factors or text strings. Data frames are central to the way that all the more recent R routines process data. For now, think of data frames as matrices, where the rows are observations and the columns are variables.

As a first example, consider the data frame `hills` that accompanies these notes[7]. This has three columns (variables), with the names `distance`, `climb`, and `time`. Typing in `summary(hills)`gives summary information on these variables. There is one column for each variable, thus:

```
> load("hills.Rdata")   # Assumes hills.Rdata is in the working directory
> summary(hills)
     distance          climb           time
    Min.: 2.000      Min.: 300      Min.: 15.95
 1st Qu.: 4.500   1st Qu.: 725   1st Qu.: 28.00
  Median: 6.000    Median:1000    Median: 39.75
    Mean: 7.529      Mean:1815      Mean: 57.88
 3rd Qu.: 8.000   3rd Qu.:2200   3rd Qu.: 68.62
    Max.:28.000      Max.:7500      Max.:204.60
```

We may for example require information on ranges of variables. Thus the range of distances (first column) is from 2 miles to 28 miles, while the range of times (third column) is from 15.95 (minutes) to 204.6 minutes.

We will discuss graphical summaries in the next section.

[7] There is also a version in the Venables and Ripley MASS library.

2.1.3 R has extensive graphical abilities

The main R graphics function is `plot()`. In addition to `plot()` there are functions for adding points and lines to existing graphs, for placing text at specified positions, for specifying tick marks and tick labels, for labelling axes, and so on.

There are various other alternative helpful forms of graphical summary. A helpful graphical summary for the `hills` data frame is the scatterplot matrix, shown in Figure 6. For this, type:

```
> pairs(hills)
```

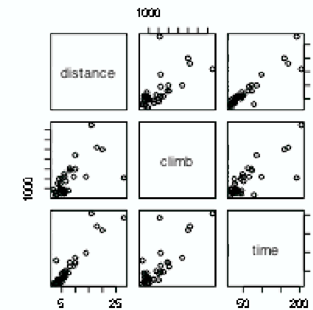

Figure 6: Scatterplot matrix for the Scottish hill race data

2.1.4 R will handle a variety of specific analyses

The examples that will be given are correlation and regression.

Correlation:

We calculate the correlation matrix for the `hills` data:

```
> options(digits=3)
> cor(hills)
         distance climb  time
distance    1.000 0.652 0.920
   climb    0.652 1.000 0.805
    time    0.920 0.805 1.000
```

Suppose we wish to calculate logarithms, and then calculate correlations. We can do all this in one step, thus:

```
> cor(log(hills))
         distance climb  time
distance     1.00 0.700 0.890
   climb     0.70 1.000 0.724
    time     0.89 0.724 1.000
```

Unfortunately R was not clever enough to relabel distance as log(distance), climb as log(climb), and time as log(time). Notice that the correlations between time and distance, and between time and climb, have reduced. Why has this happened?

Straight Line Regression:

Here is a straight line regression calculation. The data are stored in the data frame `elasticband` that accompanies these notes. The variable names are the names of columns in that data frame. The formula that is

supplied to the **lm()** command asks for the regression of distance travelled by the elastic band (**distance**) on the amount by which it is stretched (**stretch**).

```
> plot(distance ~ stretch,data=elasticband, pch=16)
> elastic.lm <- lm(distance~stretch,data=elasticband)
> lm(distance ~stretch,data=elasticband)

Call:
lm(formula = distance ~ stretch, data = elasticband)

Coefficients:
(Intercept)      stretch
    -63.571        4.554
```

More complete information is available by typing

```
> summary(lm(distance~stretch,data=elasticband))
```

Try it!

2.1.5 R is an Interactive Programming Language

We calculate the Fahrenheit temperatures that correspond to Celsius temperatures 25, 26, …, 30:

```
> celsius <- 25:30
> fahrenheit <- 9/5*celsius+32
> conversion <- data.frame(Celsius=celsius, Fahrenheit=fahrenheit)
> print(conversion)
  Celsius Fahrenheit
1      25       77.0
2      26       78.8
3      27       80.6
4      28       82.4
5      29       84.2
6      30       86.0
```

2.2 R Objects

All R entities, including functions and data structures, exist as objects. They can all be operated on as data. Type in **ls()** to see the names of all objects in your workspace. An alternative to **ls()** is **objects()**. In both cases there is provision to specify a particular pattern, e.g. starting with the letter `p'[8].

Typing the name of an object causes the printing of its contents. Try typing **q**, **mean**, etc.

In a long session, it makes sense to save the contents of the working directory from time to time. It is also possible to save individual objects, or collections of objects into a named image file. Some possibilities are:

```
save.image()                    # Save contents of workspace, into the file .RData
save.image(file="archive.RData")      # Save into the file archive.RData
save(celsius, fahrenheit, file="tempscales.RData")
```

Image files, from the working directory or (with the path specified) from another directory, can be attached, thus making objects in the file available on request. For example

```
attach("tempscales.RData")
ls(pos=2)   # Check the contents of the file that has been attached
```

The parameter **pos** gives the position on the search list. (The search list is discussed later in this chapter, in Section 2.9.)

[8] Type in **help(ls)** and **help(grep)** to get details. The pattern matching conventions are those used for **grep()**, which is modelled on the Unix grep command.

Important: On quitting, R offers the option of saving the workspace image, by default in the file **.RData** in the working directory. This allows the retention, for use in the next session in the same workspace, any objects that were created in the current session. Careful housekeeping may be needed to distinguish between objects that are to be kept and objects that will not be used again. Before typing **q()** to quit, use **rm()** to remove objects that are no longer required. Saving the workspace image will then save everything remains. The workspace image will be automatically loaded upon starting another session in that directory.

*[9]2.3 Looping

A simple example of a **for** loop is[10]

```
for (i in 1:10) print(i)
```

Here is another example of a **for** loop, to do in a complicated way what we did very simply in section 2.1.5:

```
> # Celsius to Fahrenheit
> for (celsius in 25:30)
+     print(c(celsius, 9/5*celsius + 32))
[1] 25 77
[1] 26.0 78.8
[1] 27.0 80.6
[1] 28.0 82.4
[1] 29.0 84.2
[1] 30 86
```

2.3.1 More on looping

Here is a long-winded way to sum the three numbers 31, 51 and 91:

```
> answer <- 0
> for (j in c(31,51,91)){answer <- j+answer}
> answer
[1] 173
```

The calculation iteratively builds up the object answer, using the successive values of **j** listed in the vector (31,51,91). i.e. Initially, **j**=31, and **answer** is assigned the value 31 + 0 = 31. Then **j**=51, and **answer** is assigned the value 51 + 31 = 82. Finally, **j**=91, and answer is assigned the value 91 + 81 = 173. Then the procedure ends, and the contents of **answer** can be examined by typing in answer and pressing the **Enter** key.

There is a more straightforward way to do this calculation:

```
> sum(c(31,51,91))
[1] 173
```

Skilled R users have limited recourse to loops. There are often, as in this and earlier examples, better alternatives.

2.4 Vectors

Examples of vectors are

```
c(2,3,5,2,7,1)
3:10   # The numbers 3, 4, .., 10
c(T,F,F,F,T,T,F)
c("Canberra","Sydney","Newcastle","Darwin")
```

[9] Asterisks (*) identify sections that are more technical and might be omitted at a first reading

[10] Other looping constructs are:

```
repeat <expression>    ## break must appear somewhere inside the loop
while (x>0) <expression>
```

Here <expression> is an R statement, or a sequence of statements that are enclosed within braces

Vectors may have mode logical, numeric or character[11]. The first two vectors above are numeric, the third is logical (i.e. a vector with elements of mode logical), and the fourth is a string vector (i.e. a vector with elements of mode character).

The missing value symbol, which is **NA**, can be included as an element of a vector.

2.4.1 Joining (concatenating) vectors

The **c** in **c(2, 3, 5, 7, 1)** above is an acronym for "concatenate", i.e. the meaning is: "Join these numbers together in to a vector. Existing vectors may be included among the elements that are to be concatenated. In the following we form vectors **x** and **y**, which we then concatenate to form a vector **z**:

```
> x <- c(2,3,5,2,7,1)
> x
[1] 2 3 5 2 7 1
> y <- c(10,15,12)
> y
[1] 10 15 12

> z <- c(x, y)
> z
[1]  2  3  5  2  7  1 10 15 12
```

The concatenate function **c()** may also be used to join lists.

2.4.2 Subsets of Vectors

There are two common ways to extract subsets of vectors[12].

1. Specify the numbers of the elements that are to be extracted, e.g.

```
> x <- c(3,11,8,15,12)  # Assign to x the values 3, 11, 8, 15, 12
> x[c(2,4)]   # Extract elements (rows) 2 and 4
[1] 11 15
```

One can use negative numbers to omit elements:

```
> x <- c(3,11,8,15,12)
> x[-c(2,3)]
[1]  3 15 12
```

2. Specify a vector of logical values. The elements that are extracted are those for which the logical value is **T**. Thus suppose we want to extract values of **x** that are greater than 10.

```
> x>10  # This generates a vector of logical (T or F)
[1] F T F T T
> x[x>10]
[1] 11 15 12
```

Arithmetic relations that may be used in the extraction of subsets of vectors are **<**, **<=**, **>**, >=, **==**, and **!=**. The first four compare magnitudes, **==** tests for equality, and **!=** tests for inequality.

[11] It will, later in these notes, be important to know the "class" of such objects. This determines how the method used by such generic functions as **print()**, **plot()** and **summary()**. Use the function **class()** to determine the class of an object.

[12] A third more subtle method is available when vectors have named elements. One can then use a vector of names to extract the elements, thus:

```
> c(Andreas=178, John=185, Jeff=183)[c("John","Jeff")]
 John Jeff
  185  183
```


2.4.3 The Use of NA in Vector Subscripts

Note that any arithmetic operation or relation that involves **NA** generates an **NA**. Set

```
y <- c(1, NA, 3, 0, NA)
```

Be warned that **y[y==NA] <- 0** leaves **y** unchanged. The reason is that all elements of **y==NA** evaluate to **NA**. This does not select an element of **y**, and there is no assignment.

To replace all **NA**s by 0, use

```
y[is.na(y)] <- 0
```

2.4.4 Factors

A factor is a special type of vector, stored internally as a numeric vector with values 1, 2, 3, *k*. The value *k* is the number of levels. An attributes table gives the 'level' for each integer value[13]. Factors provide a compact way to store character strings. They are crucial in the representation of categorical effects in model and graphics formulae. The class attribute of a factor has, not surprisingly, the value "factor".

Consider a survey that has data on 691 females and 692 males. If the first 691 are females and the next 692 males, we can create a vector of strings that that holds the values thus:

```
gender <- c(rep("female",691), rep("male",692))
```

(The usage is that **rep("female", 691)** creates 691 copies of the character string "female", and similarly for the creation of 692 copies of "male".)

We can change the vector to a factor, by entering:

```
gender <- factor(gender)
```

Internally the factor **gender** is stored as 691 1's, followed by 692 2's. It has stored with it the table:

1	female
2	male

Once stored as a factor, the space required for storage is reduced.

In most cases where the context seems to demand a character string, the 1 is translated into "female" and the 2 into "male". The values "female" and "male" are the *levels* of the factor. By default, the levels are in alphanumeric order, so that "female" precedes "male". Hence:

```
> levels(gender)  # Assumes gender is a factor, created as above
[1] "female" "male"
```

The order of the levels in a factor determines the order in which the levels appear in graphs that use this information, and in tables. To cause "male" to come before "female", use

```
gender <- relevel(gender, ref="male")
```

An alternative is

```
gender <- factor(gender, levels=c("male", "female"))
```

This last syntax is available both when the factor is first created, or later when one wishes to change the order of levels in an existing factor. Incorrect spelling of the level names will generate an error message. Try

```
gender <- factor(c(rep("female",691), rep("male",692)))
table(gender)
gender <- factor(gender, levels=c("male", "female"))
table(gender)
gender <- factor(gender, levels=c("Male", "female"))
                   # Erroneous - "male" rows now hold missing values
table(gender)
rm(gender)         # Remove gender
```

[13] The attributes() function makes it possible to inspect attributes. For example

```
attributes(factor(1:3))
```

The function **levels()** gives a better way to inspect factor levels.

2.5 Data Frames

Data frames are fundamental to the use of the R modelling and graphics functions. A data frame is a generalisation of a matrix, in which different columns may have different modes. All elements of any column must however have the same mode, i.e. all numeric or all factor, or all character.

Among the data sets that are supplied to accompany these notes is one called **Cars93.summary**, created from information in the **Cars93** data set in the Venables and Ripley *MASS* package. Here it is:

```
> Cars93.summary
        Min.passengers Max.passengers No.of.cars abbrev
Compact              4              6         16      C
Large                6              6         11      L
Midsize              4              6         22      M
Small                4              5         21     Sm
Sporty               2              4         14     Sp
Van                  7              8          9      V
```

The data frame has row labels (access with **row.names(Cars93.summary)**) Compact, Large, . . . The column names (access with **names(Cars93.summary)**) are **Min.passengers** (i.e. the minimum number of passengers for cars in this category), **Max.passengers, No.of.cars**., and **abbrev**. The first three columns have mode numeric, and the fourth has mode character. Columns can be vectors of any mode. The column **abbrev** could equally well be stored as a factor.

Any of the following[14] will pick out the fourth column of the data frame **Cars93.summary**, then storing it in the vector **type**.

```
type <- Cars93.summary$abbrev
type <- Cars93.summary[,4]
type <- Cars93.summary[,"abbrev"]
type <- Cars93.summary[[4]]       # Take the object that is stored
# in the fourth list element.
```

2.5.1 Data frames as lists

A data frame is a list[15] of column vectors, all of equal length. Just as with any other list, subscripting extracts a list. Thus **Cars93.summary[4]** is a data frame with a single column, which is the fourth column vector of **Cars93.summary**. As noted above, use **Cars93.summary[[4]]** or **Cars93.summary[,4]** to extract the column vector.

The use of matrix-like subscripting, e.g. **Cars93.summary[,4]** or **Cars93.summary[1, 4]**, takes advantage of the rectangular structure of data frames.

2.5.2 Inclusion of character string vectors in data frames

When data are input using **read.table()**, or when the **data.frame()** function is used to create data frames, vectors of character strings are by default turned into factors. The parameter setting **as.is=T, available both with read.table() and with data.frame(),** will if needed ensure that character strings are input without such conversion**.**

2.5.3 Built-in data sets

We will often use data sets that accompany one of the R packages, usually stored as data frames. One such data frame, in the *datasets* package, is **trees**, which gives girth, height and volume for 31 Black Cherry Trees.

```
> data(trees)      # Load data set (not needed for versions of R >= 2.0.0)
```

Here is summary information on this data frame

```
> summary(trees)
```

[14] Also legal is **Cars93.summary[2]**. This gives a data frame with the single column **Type**.

[15] In general forms of list, elements that are of arbitrary type. They may be any mixture of scalars, vectors, functions, etc.


```
     Girth           Height       Volume
 Min.   : 8.30   Min.   :63   Min.   :10.20
 1st Qu.:11.05   1st Qu.:72   1st Qu.:19.40
 Median :12.90   Median :76   Median :24.20
 Mean   :13.25   Mean   :76   Mean   :30.17
 3rd Qu.:15.25   3rd Qu.:80   3rd Qu.:37.30
 Max.   :20.60   Max.   :87   Max.   :77.00
```

Type **data()** to get a list of built-in data sets in the packages that have been loaded[16].

2.6 Common Useful Functions

```
print()     # Prints a single R object
cat()       # Prints multiple objects, one after the other
length()    # Number of elements in a vector or of a list
mean()
median()
range()
unique()    # Gives the vector of distinct values
diff()      # Replace a vector by the vector of first differences
              # N. B. diff(x) has one less element than x
sort()      # Sort elements into order, but omitting NAs
order()     # x[order(x)] orders elements of x, with NAs last
cumsum()
cumprod()
rev()       # reverse the order of vector elements
```

The functions **mean()**, **median()**, **range()**, and a number of other functions, take the argument **na.rm=T;** i.e. remove NAs, then proceed with the calculation.

By default, **sort()** omits any NAs. The function **order()** places NAs last. Hence:

```
> x <- c(1, 20,  2, NA, 22)
> order(x)
[1] 1 3 2 5 4
> x[order(x)]
[1]  1  2 20 22 NA
> sort(x)
[1]  1  2 20 22
```

2.6.1 Applying a function to all columns of a data frame

The function **sapply()** takes as arguments the the data frame, and the function that is to be applied. The following applies the function **is.factor()** to all columns of the supplied data frame **rainforest**[17].

```
> sapply(rainforest, is.factor)
    dbh    wood    bark    root  rootsk  branch species
  FALSE   FALSE   FALSE   FALSE   FALSE   FALSE    TRUE
> sapply(rainforest[,-7], range)   # The final column (7) is a factor
     dbh wood bark root rootsk branch
[1,]   4   NA   NA   NA     NA     NA
[2,]  56   NA   NA   NA     NA     NA
```

[16] The list include all packages that are in the current environment.

[17] Source: Ash, J. and Southern, W. 1982: Forest biomass at Butler's Creek, Edith & Joy London Foundation, New South Wales, Unpublished manuscript. See also Ash, J. and Helman, C. 1990: Floristics and vegetation biomass of a forest catchment, Kioloa, south coastal N.S.W. Cunninghamia, 2(2): 167-182.

The functions **mean()** and **range()**, and a number of other functions, take the parameter **na.rm**. For example

```
> range(rainforest$branch, na.rm=T)  # Omit NAs, then determine the range
[1]   4 120
```

One can specify **na.rm=T** as a third argument to the function **sapply**. This argument is then automatically passed to the function that is specified in the second argument position. For example:

```
> sapply(rainforest[,-7], range, na.rm=T)
     dbh wood bark root rootsk branch
[1,]   4    3    8    2    0.3      4
[2,]  56 1530  105  135   24.0    120
```

Chapter 8 has further details on the use of **sapply()**. There is an example that shows how to use it to count the number of missing values in each column of data.

2.7 Making Tables

table() makes a table of counts. Specify one vector of values (often a factor) for each table margin that is required. For example:

```
> library(lattice)  # The data frame barley accompanies lattice
> table(barley$year, barley$site)

       Grand Rapids Duluth University Farm Morris Crookston Waseca
  1932 10           10     10              10     10        10
  1931 10           10     10              10     10        10
```

WARNING: **NA**s are by default ignored. The action needed to get **NA**s tabulated under a separate **NA** category depends, annoyingly, on whether or not the vector is a factor. If the vector is not a factor, specify **exclude=NULL**. If the vector is a factor then it is necessary to generate a new factor that includes **"NA"** as a level. Specify **x <- factor(x,exclude=NULL)**

```
> x_c(1,5,NA,8)
> x <- factor(x)
> x
[1] 1  5  NA 8
Levels:  1 5 8
> factor(x,exclude=NULL)
[1] 1  5  NA 8
Levels:  1 5 8 NA
```

2.7.1 Numbers of NAs in subgroups of the data

The following gives information on the number of NAs in subgroups of the data:

```
> table(rainforest$species, !is.na(rainforest$branch))

                  FALSE TRUE
  Acacia mabellae     6   10
  C. fraseri          0   12
  Acmena smithii     15   11
  B. myrtifolia       1   10
```

Thus for Acacia mabellae there are 6 NAs for the variable branch (i.e. number of branches over 2cm in diameter), out of a total of 16 data values.

2.8 The Search List

R has a search list where it looks for objects. This can be changed in the course of a session. To get a full list of these directories, called *databases*, type:

```
> search()
[1] ".GlobalEnv"        "package:methods"   "package:stats"
```



```
[4] "package:graphics"  "package:grDevices" "package:utils"
[7] "package:datasets"  "Autoloads"         "package:base"
```

Notice that the loading of a new package extends the search list.

```
> library(MASS)
> search()
 [1] ".GlobalEnv"        "package:MASS"      "package:methods"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "Autoloads"
[10] "package:base"
```

Use of **attach()** likewise extends the search list. This function can be used to attach data frames or lists (use the name, without quotes) or image (**.RData**) files (the file name is placed in quotes).

The following demonstrates the attaching of the data frame **primates**:

```
> names(primates)
[1] "Bodywt"  "Brainwt"
> Bodywt
Error: Object "Bodywt" not found
> attach(primates)     # R will now know where to find Bodywt
> Bodywt
[1]  10.0 207.0  62.0   6.8  52.2
```

Once the data frame **primates** has been attached, its columns can be accessed by giving their names, without further reference to the name of the data frame. In technical terms, the data frame becomes a *database*, which is searched as required for objects that the user may specify.

2.9 Functions in R

We give two simple examples of R functions.

2.9.1 An Approximate Miles to Kilometers Conversion

```
miles.to.km <- function(miles)miles*8/5
```

The return value is the value of the final (and in this instance only) expression that appears in the function body[18]. Use the function thus

```
> miles.to.km(175)  # Approximate distance to Sydney, in miles
[1] 280
```

The function will do the conversion for several distances all at once. To convert a vector of the three distances 100, 200 and 300 miles to distances in kilometers, specify:

```
> miles.to.km(c(100,200,300))
[1] 160 320 480
```

2.9.2 A Plotting function

The data set **florida** has the votes in the 2000 election for the various US Presidential candidates, county by county in the state of Florida. The following plots the vote for Buchanan against the vote for Bush.

```
attach(florida)
plot(BUSH, BUCHANAN, xlab="Bush", ylab="Buchanan")
detach(florida)   # In S-PLUS, specify detach("florida")
```

Here is a function that makes it possible to plot the figures for any pair of candidates.

```
plot.florida <- function(xvar="BUSH", yvar="BUCHANAN"){
     x <- florida[,xvar]
     y <- florida[,yvar]
```

[18] Alternatively a return value may be given using an explicit **return()** statement. This is however an uncommon construction

```
      plot(x, y, xlab=xvar,ylab=yvar)
      mtext(side=3, line=1.75,
       "Votes in Florida, by county, in \nthe 2000 US Presidential election")
}
```

Note that the function body is enclosed in braces ({ }).

Figure 7 shows the graph produced by `plot.florida()`, i.e. parameter settings are left at their defaults.

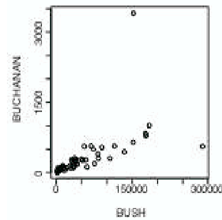

Figure 7: Election night count of votes received, by county, in the US 2000 Presidential election.

As well as `plot.florida()`, the function allows, e.g.

```
plot.florida(yvar="NADER")  # yvar="NADER" over-rides the default
plot.florida(xvar="GORE", yvar="NADER")
```

2.10 More Detailed Information

Chapters 7 and 8 have a more detailed coverage of the topics in this chapter. It may pay, at this point, to glance through chapters 7 and 8. Remember also to use R's help pages and functions.

Topics from chapter 7, additional to those covered above, that may be important for relatively elementary uses of R include:

The entry of patterned data (7.1.3)

The handling of missing values in subscripts when vectors are assigned (7.2)

Unexpected consequences (e.g. conversion of columns of numeric data into factors) from errors in data (7.4.1).

2.11 Exercises

1. For each of the following code sequences, predict the result. Then do the computation:

a)
```
answer <- 0
for (j in 3:5){ answer <- j+answer }
```

b)
```
answer<- 10
for (j in 3:5){ answer <- j+answer }
```

c)
```
answer <- 10
for (j in 3:5){ answer <- j*answer }
```

2. Look up the help for the function `prod()`, and use `prod()` to do the calculation in 1(c) above. Alternatively, how would you expect `prod()` to work? Try it!

3. Add up all the numbers from 1 to 100 in two different ways: using `for` and using `sum`. Now apply the function to the sequence 1:100. What is its action?

4. Multiply all the numbers from 1 to 50 in two different ways: using `for` and using `prod`.

5. The volume of a sphere of radius r is given by 4 r^3/3. For spheres having radii 3, 4, 5, …, 20 find the corresponding volumes and print the results out in a table. Use the technique of section 2.1.5 to construct a data frame with columns `radius` and `volume`.

6. Use `sapply()` to apply the function `is.factor` to each column of the supplied data frame `tinting`. For each of the columns that are identified as factors, determine the levels. Which columns are ordered factors? [Use `is.ordered()`].