R in a hourR

David Hood, IT Training, University of Otago

2/24/2017

Quantitative Data

Before using any professional level analysis tool, your life is easiest if the data you is using is well organised. For data in a spreadsheet, this means:

- A rectangular block of data. It may contain spaces for missing entries, but no fully blank rows or columns.
- Columns are variables. These are the things that are potential important in your analysis. Each column contains one type of data (numbers, or text, or dates)
- Rows are observations. Individual entries you are keeping track of
- One row of headings at the top of the sheet, and each column has a heading.

By saving a well organised data set as a csv file, it is easy to move into other analysis programs (including R). However, any formatting that you have applied to the spreadsheet (number of decimal places, date formatting) will be what is in the csv. Any dates are best formatted as 4 digit years and numeric months. Remove any dollar signs, commas, and spaces within numeric values.

Project folder

It can be a good idea to keep together the raw data, the records of an analysis, and the results, all in a single folder. To do this, you can change the working directory to the folder you want to use. In RStudio use the Session menu - Set Working Directory - Choose Directory command to find you folder containing your data. If just using R use the Misc - Change Working Directory command.

Replication

You can type commands directly into R, but a better plan is to write the commands out in a text file and run the text file as a group of commands. This lets you (or anyone else) instantly repeat your analysis. In the basic version of R, you can run the R commands from a script file in your project folder using the source("yourFileName") command, and write new scripts with a File - New Document command. However, it is much easier to work in RStudio with File - New File - R Script, as you can work with a script in the script window, running it all using the source button (or tick source on save to run it whenever you save the script), and also you can select individual lines (or just be clicked in a single line) and use the Run button at the top of the script.

By having a script saved in the same folder as your data, you can shift to another machine and repeat all your commands by opening the script and rerunning the commands.

Reading in a csv

There are different variations for data in different sources, so for reading in a csv file from the working directory, the most basic command is

```
read.csv("yourfilename.csv")
```

You feed the name of the file into the function (because it is text, it is in quotes), it looks in the working directory, and reads in the file. But in this most basic form it then dumps the contents of the csv all over the console as output because we did not save it anywhere. So instead we use

```
my_data <- read.csv("yourfilename.csv")
```

Where my_data is the name (letters, numbers, full stops, and underscores only) you want to refer to the data by, so if you are using this as an example you should be substituting your own name from this point on. Think of it as being the label on the box the data is in. The csv data comes out of the function and is put in the box by the arrow symbol (less than + minus sign)

str()

Now that your data is stored you can investigate it with the str() command, for structure.

```
str(my_data)
```

Looking at the results of str() you may notice changes to variable (column) names, like spaces replaced by . symbols. This is to make the names safe to use in R. When referring to them in R, you will need to use this form of the name.

You will get a report of the contents of your data in console. A $ will be at the start of each row, then the name of the column/variable, and then the type of the data. In particular, if you have columns containing text or dates they will have been read in as a data type of factor, which is an organised categorical variable. If you are going to need to make changes to a text column, such as converting a text representation of a date to a true date, it is better to read the data in as disorganized text to be changed, which is a type of data called a character. To change the way a function, like read.csv(), behaves, you need to change the settings. To learn about the settings you consult the help for that function.

consulting help

In the console in R, type a question mark and the name of the function

```
?read.csv
```

In this case it opens the help for read.table and related functions, of which read.csv is one. Among the potential settings we can add are things like skip, if we wanted to start reading further down the file, and stringsAsFactors to control how text is read in. If stringsAsFactors = FALSE, text is read in as text and is easy to modify. If stringsAsFactors = TRUE, text is read in as Factors (organised text), which is easy to analyse and graph but hard to change. The default is TRUE. We can return to our read.csv line in the script and change it from the default by rerunning the line:

```
my_data <- read.csv("yourfilename.csv", stringsAsFactors = TRUE)
```

Then repeating the str() command to check the updated data.

tidying up the data

If the date columns are not a date type, see the Calculations section If there were empty boxes (no value) that should be 0, see the Calculations section If there were numeric values (like -99) which should be empty (no value), see the Calculations section If there were extra symbols (like $ or ,) which should be removed, see the Calculations section

Using data

To use variables in a data frame (the rectangular block), you need to be able to specifically refer to them. Having checked the names with str, you can refer to individual variables in the form

```
DataFrameName$VariableName
```

If you had a data_frame named my_data with the variables (columns) labelled numeric1 (a column of numbers), numeric2 (a column of numbers), and categorical1 (a column of text category labels) you might make graphs by referring to the variable in the data as:

A comparison of two numerics

```
plot(my_data$numeric1, my_data$numeric2)
```

A histogram distribution of a single numeric variable

```
hist(my_data$numeric1)
```

A boxplot distribution of a numeric variable on the basis of a categorical variable

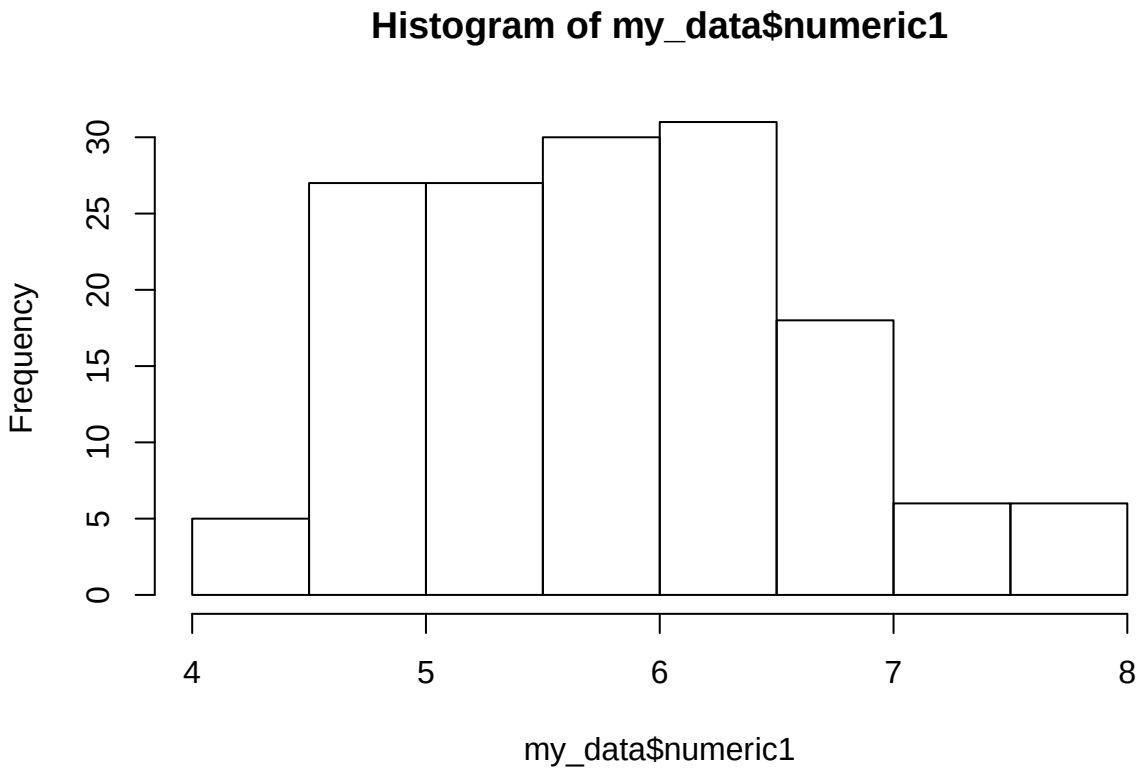
```
boxplot(my_data$numeric1 ~ my_data$categorical1)
```

The ~ (tilde) symbol in R means dependent on.

Graphs can be saved as image files, normally a png image.

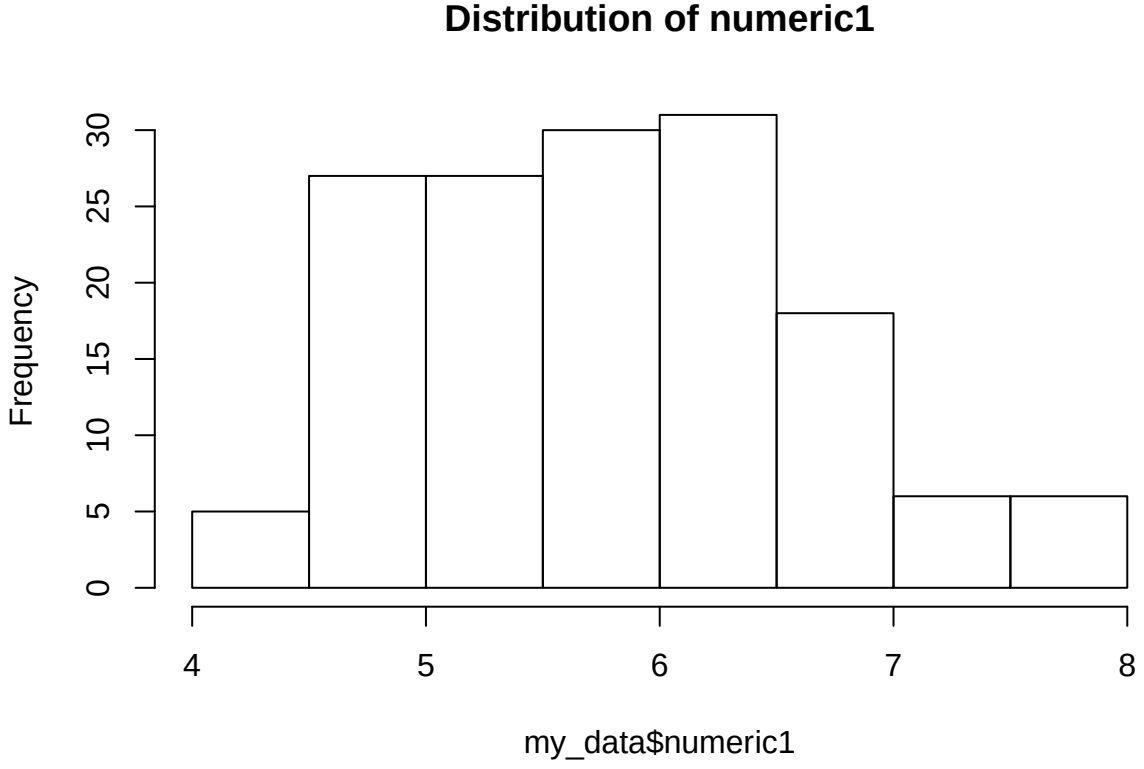
For making fancier graphs it is normally a matter of changing settings in the graph making command, or using additional settings to add more features to the graph. In both cases, the help offers some settings, but in the case of making graphs it often refers you to the help for par() as well. As an example of working through the settings

```
hist(my_data$numeric1)
```



I want to customise the title, the help for hist() (checked with the ?hist command in console) suggests a setting called main to the hist command in my script

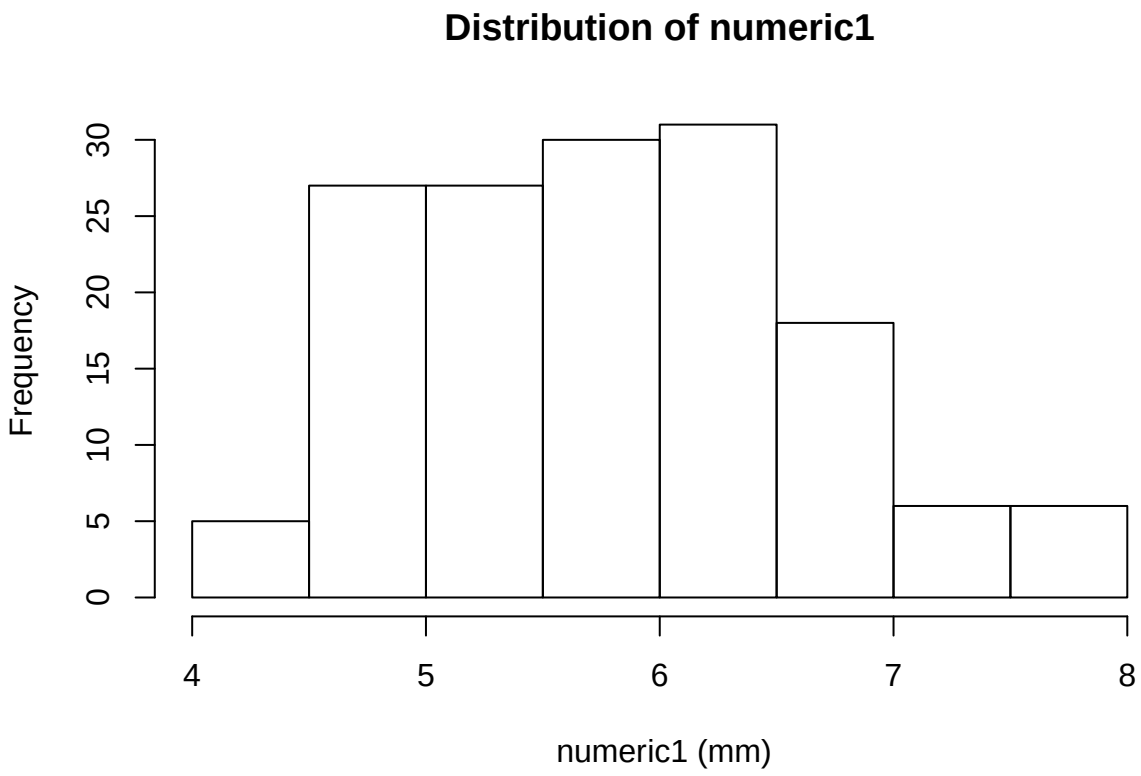
```
hist(my_data$numeric1, main="Distribution of numeric1")
```



I want to customise the x axis, the help for hist() suggests a setting called xlab. As commands start to get

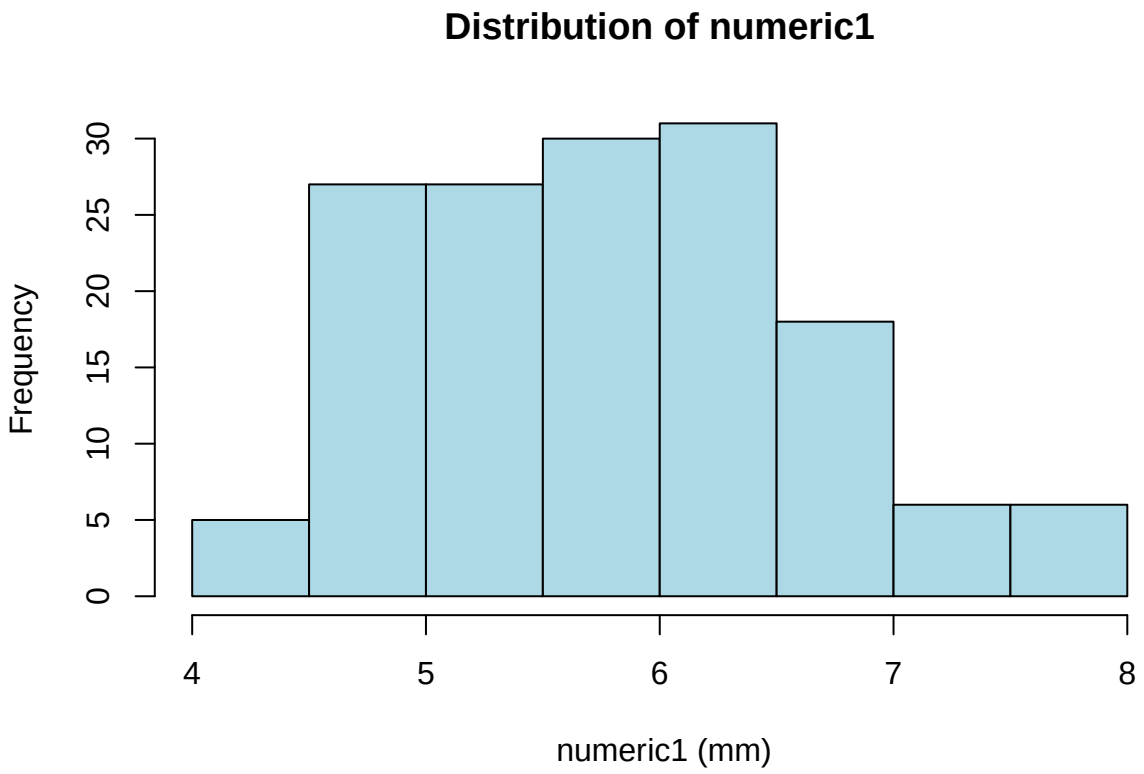
long, you can separate them over several lines providing it is clear to R it is "too be continued". For example if a line finishes with a comma R will assume more settings are coming. If you are running commands in console and provide an unfinished command, R will give a + prompt to indicate it is waiting for you to finish the command.

```
hist(my_data$numeric1, main="Distribution of numeric1",
     xlab=" numeric1 (mm)")
```



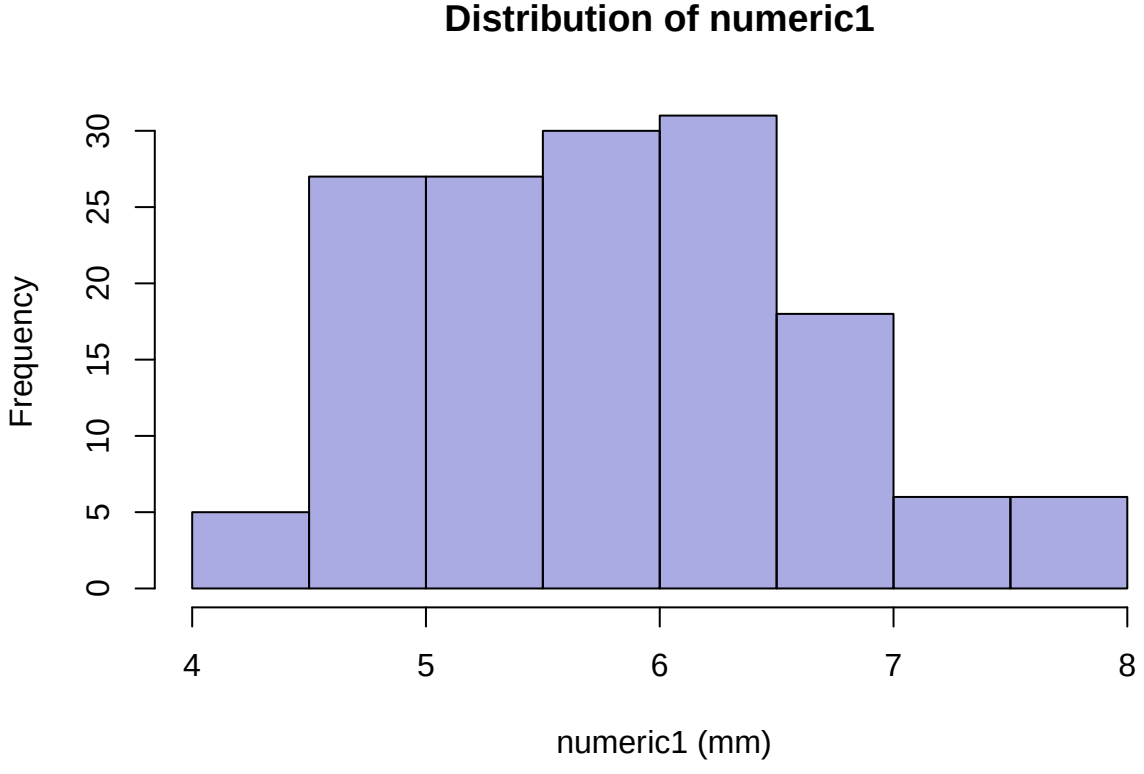
I want to customise colouring, the help for hist() suggests a setting called col, which if I do a web search colours in R can be specified by words

```
hist(my_data$numeric1, main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="lightblue")
```



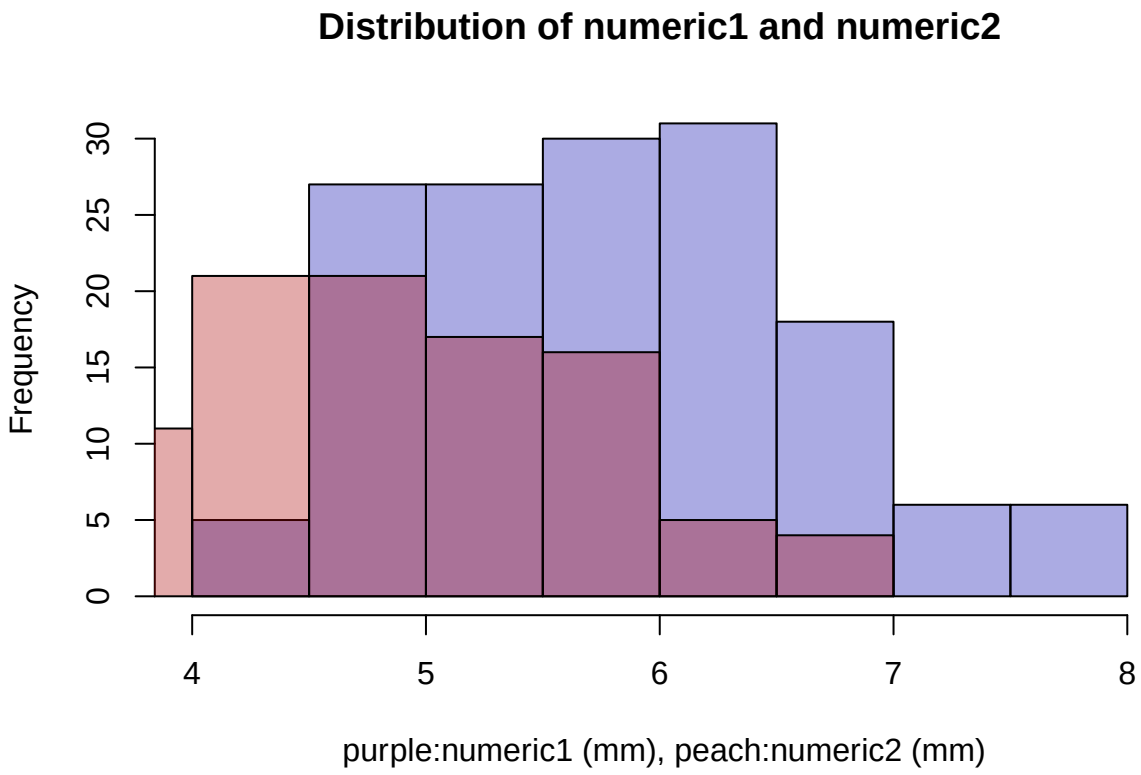
For very fancy, see-through colouring, you have to provide the amount of red, green, blue, and solidness to use as hexidecimal (0 to 9 then A to F) numbers in the form #RRGGBBSS

```
hist(my_data$numeric1, main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="#0000AA55")
```



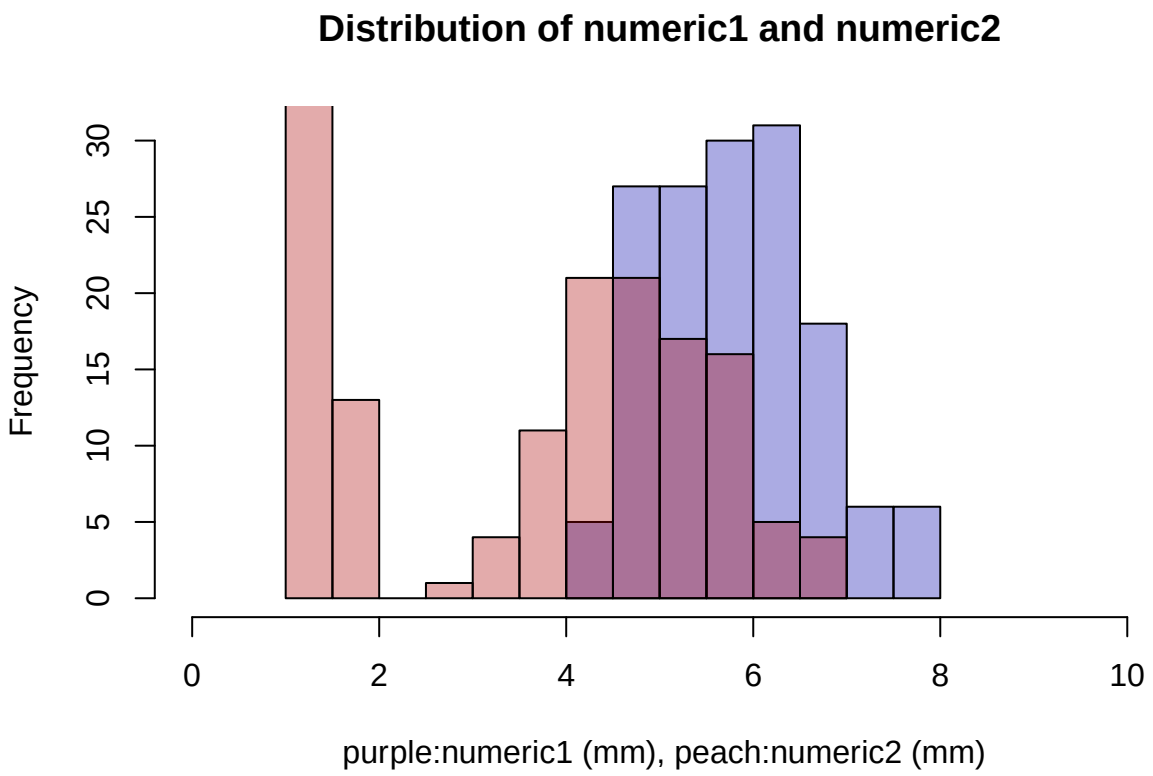
I want to overlay two histograms and have noticed in the help there is an add setting I could use on the second histogram

```
hist(my_data$numeric1, main="Distribution of numeric1 and numeric2",
     xlab=" purple:numeric1 (mm), peach:numeric2 (mm)", col="#0000AA55")
hist(my_data$numeric2, col="#AA000055", add=TRUE)
```



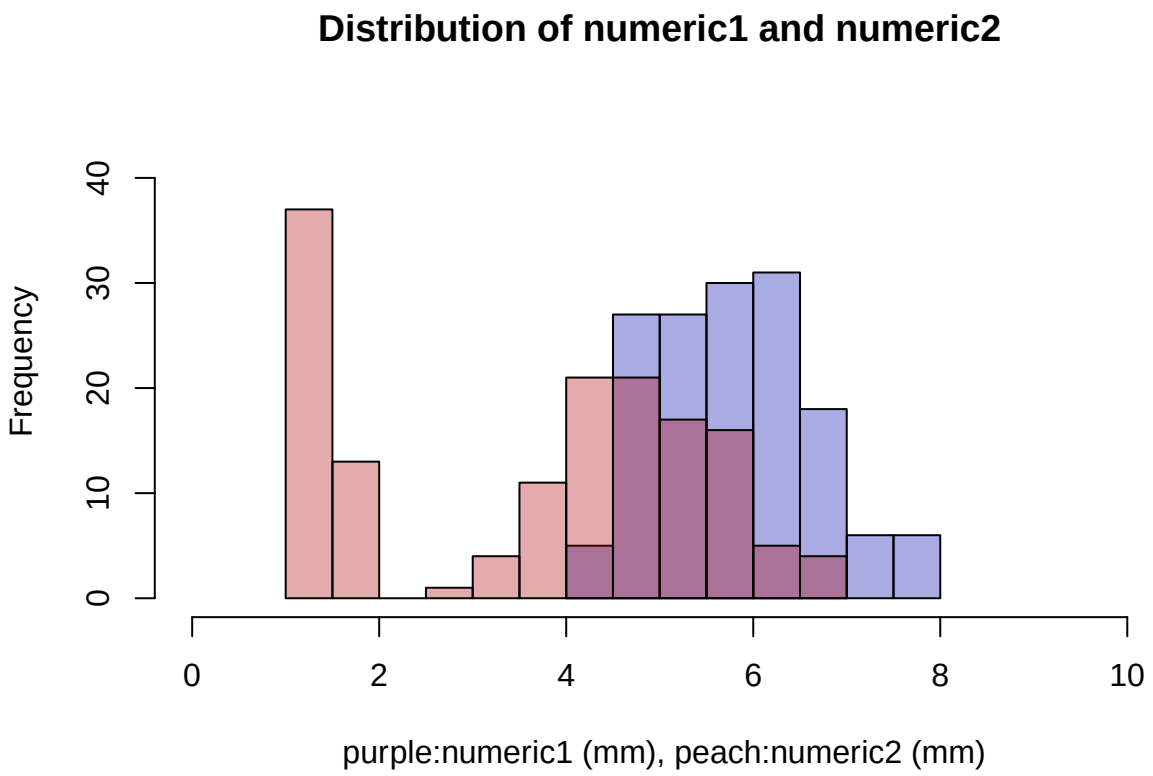
But the second histogram doesn't use the same range as the first, and the help suggests axis size is controlled with a xlim setting, this needs 2 entries (for the minimum and maximum) and I can provide that with the c() function for combining things. This is an example of a function inside a function

```
hist(my_data$numeric1, main="Distribution of numeric1 and numeric2",
     xlab=" purple:numeric1 (mm), peach:numeric2 (mm)", col="#0000AA55",
     xlim=c(0,10))
hist(my_data$numeric2, col="#AA000055", add=TRUE)
```



The height is wrong, but the help suggests ylim is the answer

```
hist(my_data$numeric1, main="Distribution of numeric1 and numeric2",
     xlab=" purple:numeric1 (mm), peach:numeric2 (mm)", col="#0000AA55",
     xlim=c(0,10), ylim=c(0,45))
hist(my_data$numeric2, col="#AA000055", add=TRUE)
```

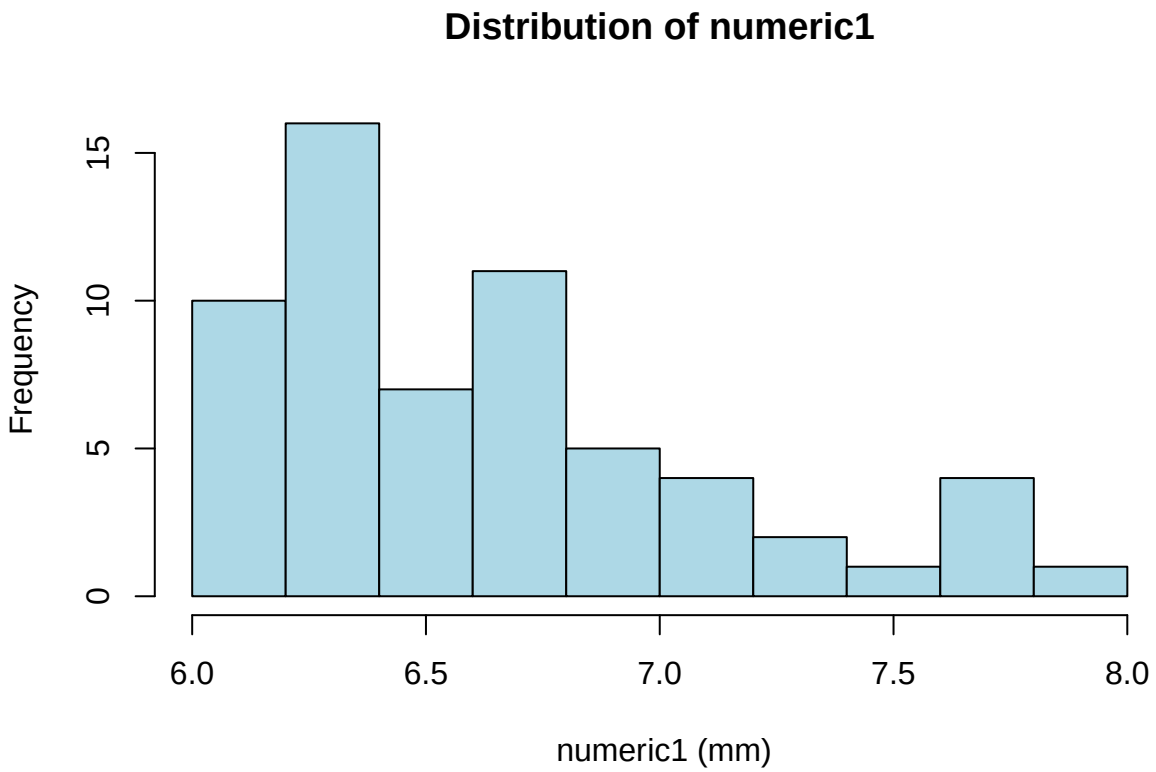


I might then decide to add a legend to the graph in the topright with a legend() function. Try working out the settings from the help.

Subsetting your data

Subsetting is when you only want to use some of your data. There are two main forms of subsetting a data set- temporarily subset the data for a task, and permanently creating a new copy of the subsetted data. If you are using a dataframe (normally when preparing data for a statistical test), you can put the criteria for making the subset of the rows you want in square brackets [] after the details of what you want, followed by a comma, followed by the details of the columns you want. If you were subsetting a single column (an array) you do not need the comma and the second entry. You can make what is going on in the data clearer by separating it into a two step process: specifying the rule for the data, then applying that rule. Continuing with the histogram example, an example of a logical rule

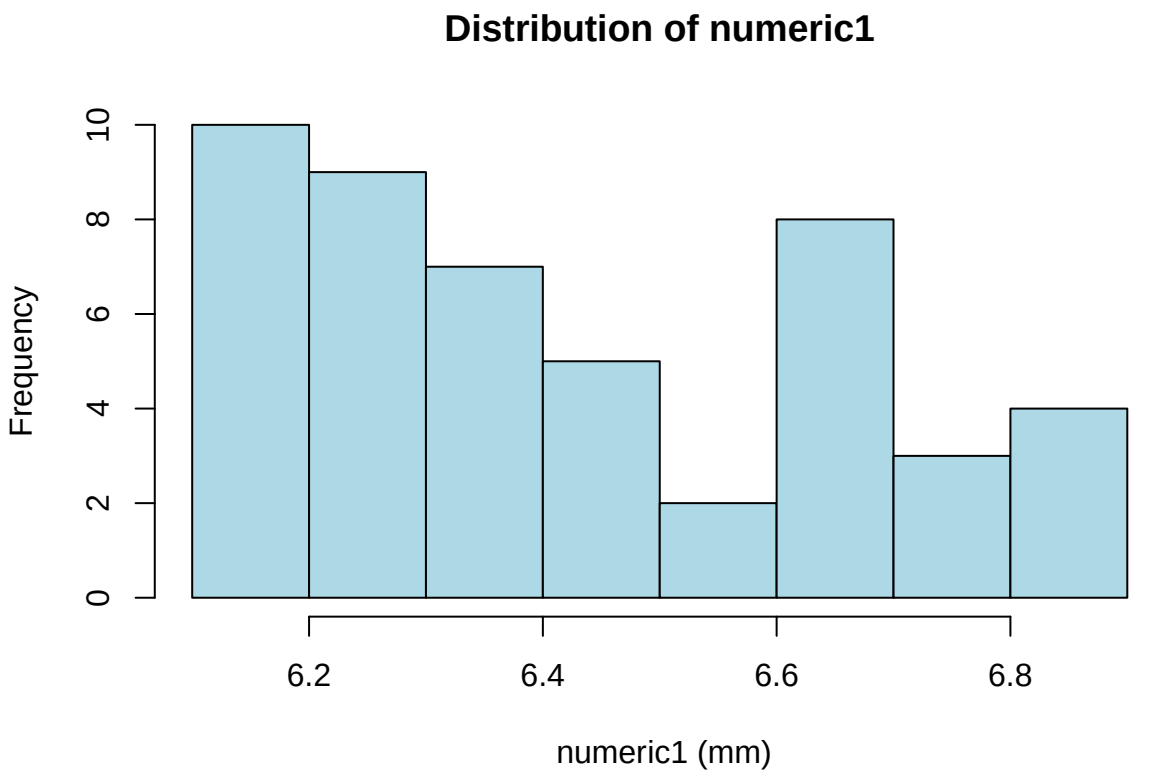
```
rowcriteria <- my_data$numeric1 > 6
hist(my_data$numeric1[rowcriteria], main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="lightblue")
```



More

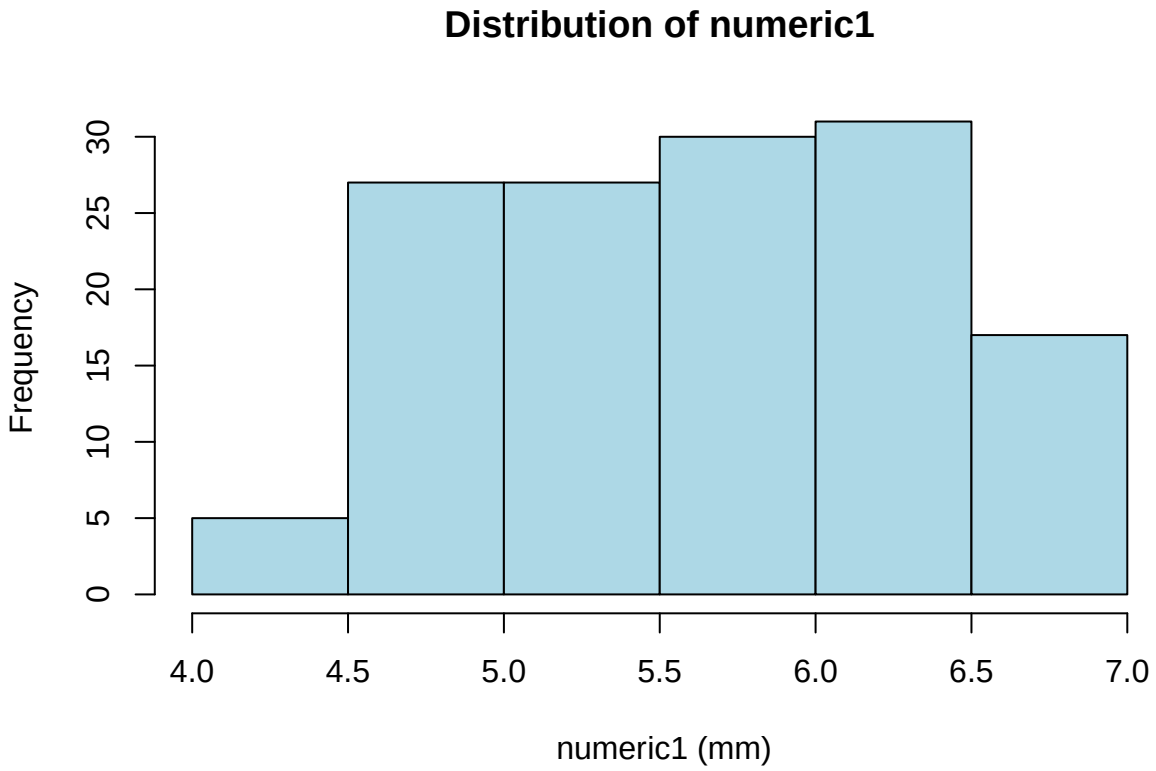
complex rules can be done with the and symbol & to narrow responses

```
rowcriteria <- my_data$numeric1 > 6 & my_data$numeric1 < 7
hist(my_data$numeric1[rowcriteria], main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="lightblue")
```



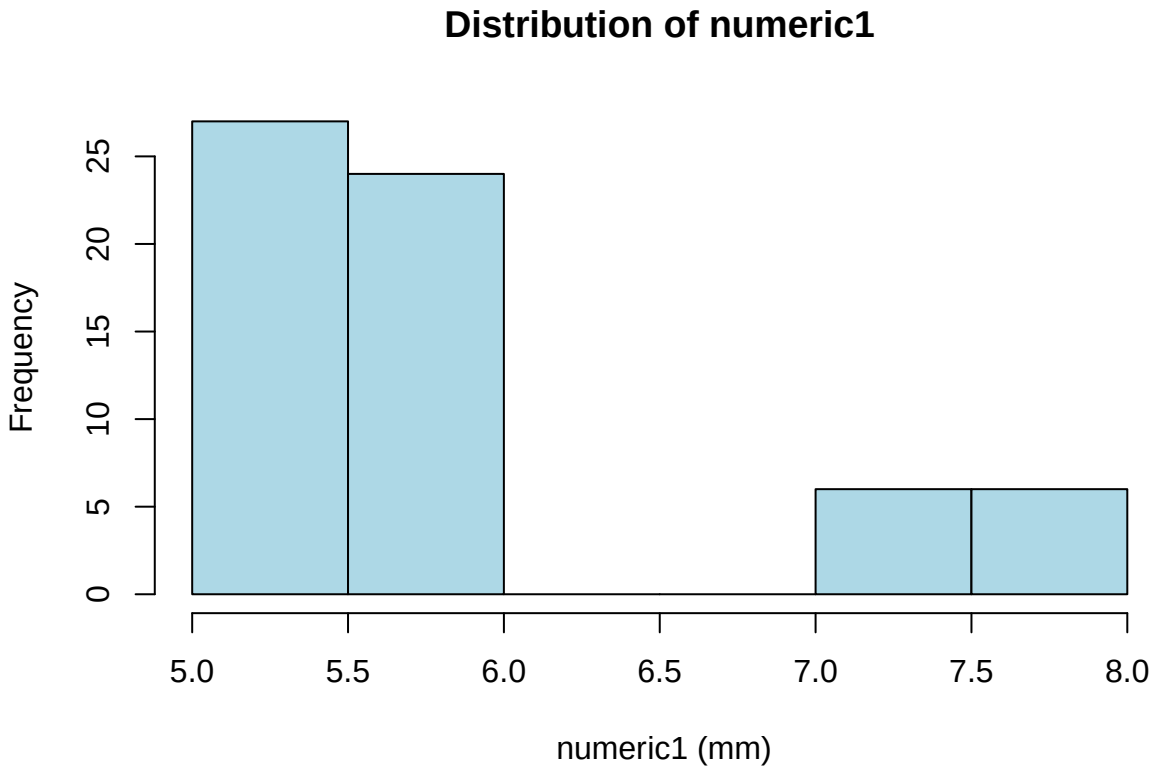
The or symbol | can be used to find more than one group

```
rowcriteria <- my_data$numeric1 < 6 | my_data$numeric1 < 7
hist(my_data$numeric1[rowcriteria], main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="lightblue")
```



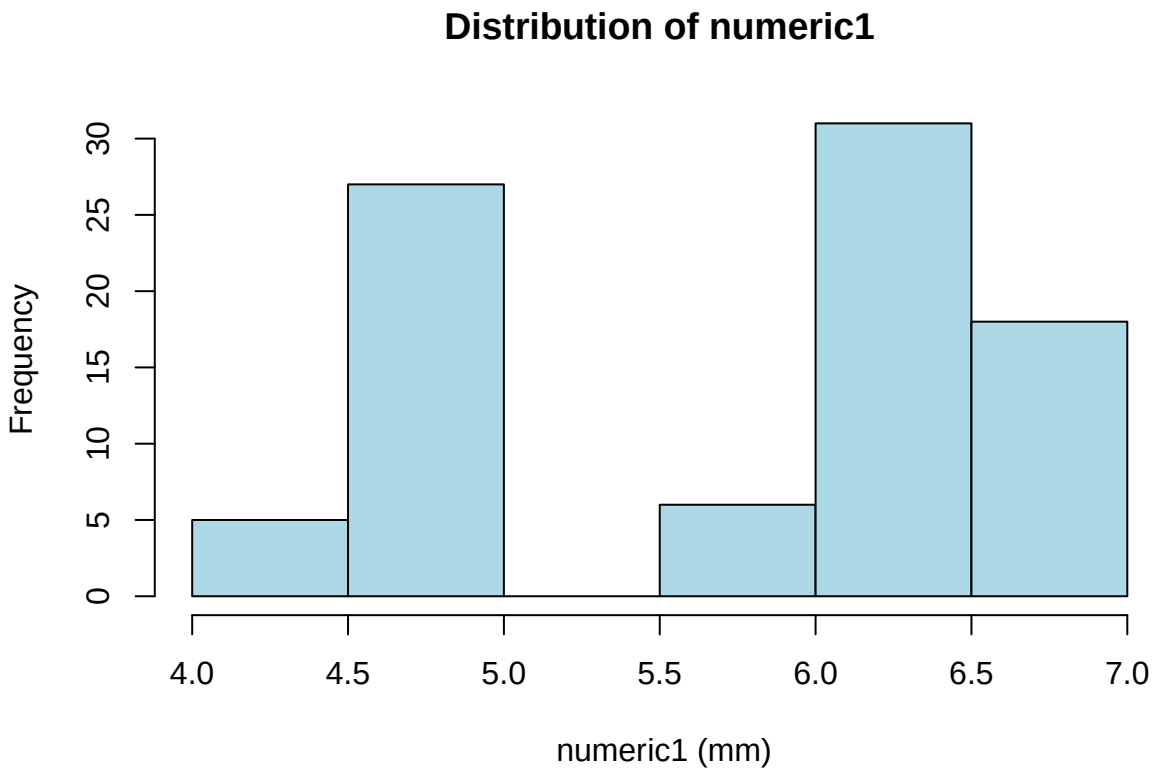
If writing complex rules, it is a good idea to add parentheses to control the order of operation. You can also extend over multiple lines if it is clear to R it is "too be continued"

```
rowcriteria <- (my_data$numeric1 > 5 & my_data$numeric1 < 6) |
  (my_data$numeric1 > 7 & my_data$numeric1 < 8)
hist(my_data$numeric1[rowcriteria], main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="lightblue")
```



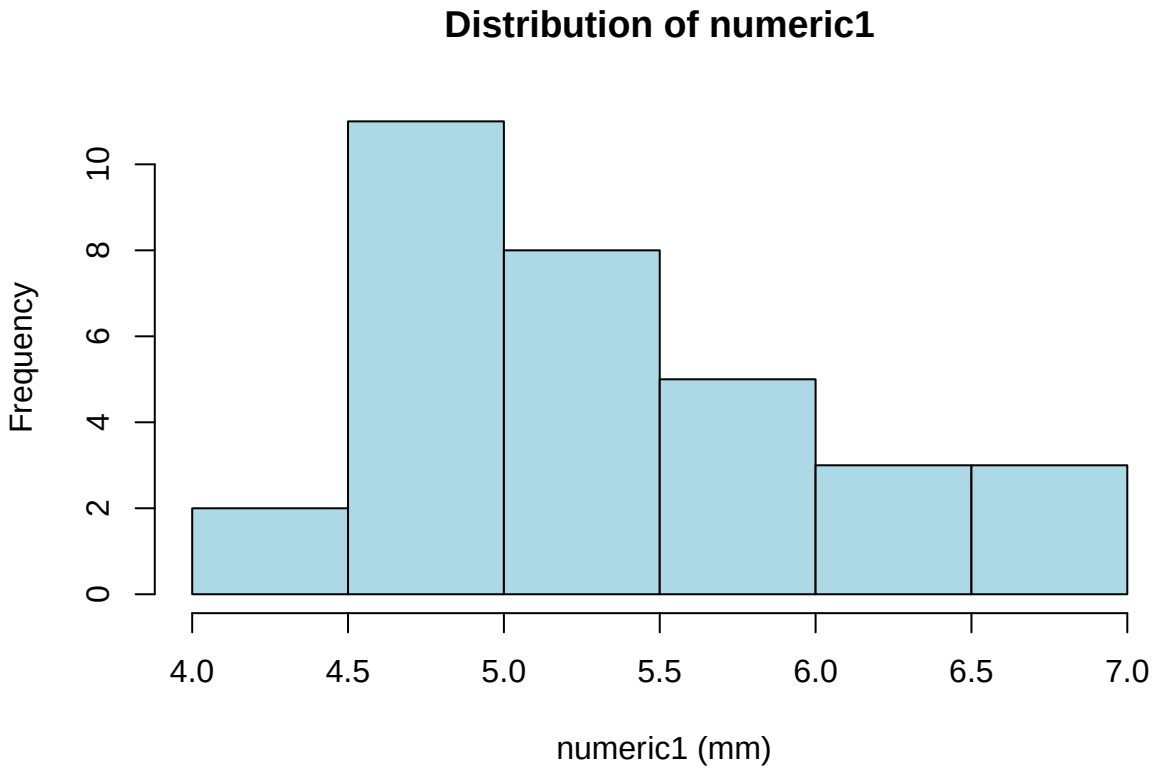
Applying a NOT symbol ! to a group swaps the set to "Not these ones"

```
rowcriteria <- !((my_data$numeric1 > 5 & my_data$numeric1 < 6) |
                   (my_data$numeric1 > 7 & my_data$numeric1 < 8))
hist(my_data$numeric1[rowcriteria], main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="lightblue")
```



Though it is less common, you can also specify rows numerically, either in ranges, or a list of numbers, or both

```
rowcriteria <- c(1:20,37,46,51:59,84)
hist(my_data$numeric1[rowcriteria], main="Distribution of numeric1",
     xlab=" numeric1 (mm)", col="lightblue")
```



If you are copying a few columns in a data frame for a specific test, you need to specify both the rows and the columns. It is often easiest to specify the columns by name

```
colcriteria <- c("numeric1", "numeric2")
rowcriteria <- c(1:20,37,46,51:59,84)
data_for_t_test <- my_data[rowcriteria, columncriteria]
```

For preparing data for statistical tests, blank entries (not applicable, marked N/A) often need special handling. Normal rules do not apply to unknown values, as you don't know if they meet the rule or not. As many statistical tests object to blank values, you need to identify them with a special is.na() function, and not use those entries with a !

```
colcriteria <- c("numeric1", "numeric2")
rowcriteria <- !is.na(numeric1) & !is.na(numeric2)
data_for_t_test <- my_data[rowcriteria, columncriteria]
```

Calculations

To create a new entry in the data frame that is a result of a calculation based on other entries, assign it to a new variable use the form DataFrameName$NewVariableName <- calculation. Where the calculation uses a mix of variable names, parentheses to control order, operators, and functions. In R, like other professional level analysis tools, the results of a calculation are conducted on the assumption "for each entry in a column do this".

Making a calculation on two columns and storing it in a third- the ratio of numeric1 to numeric2 for each observation

```
my_data$numeric1_per_numeric2 <- my_data$numeric1 / my_data$numeric2
```

Making a calculation using a function- how far is each numeric1 from the median for numeric1

```
my_data$numeric1_from_median <- my_data$numeric1 -
    median(my_data$numeric1, na.rm=TRUE)
```

If you are wondering what na.rm does, consult the help for median ?median

Converting dates in text format to true dates using a function

To convert textual dates with year, month, and day information to a true date, you can use the as.Date function, but are going to have to tell R how the dates are set out. Assuming you are dealing with 4 digit numeric years, numeric months, and numeric days, you need to provide a template to how the dates are set out. If it is in the common New Zealand format of dd/mm/yyyy the conversion is

```
my_data$event_date <- as.Date(my_data$old_date, format="%d/%m/%Y")
```

Converting numbers in text format to true numbers using a function

If your data had numeric entries with stray bits of text in the column, you violated the well orgnanised data rule about only one kind of data in a column and need to fix it

```
my_data$numeric_character4 <- as.numeric(my_data$character4)
```

This will cause a warning message, because it will have to convert the entries with text to blanks, as they are not number entries.

Converting dates in factor format to true dates using a function

If textual dates are factors (checked with the str() command) you need to convert the data to characters then convert the characters to a date

```
my_data$event_date <- as.Date(as.character(my_data$old_date), format="%d/%m/%Y")
```

This is an example of a function inside another function, getting the data ready for the next step.

Converting numbers in factor format to true numbers using a function

Again, you need to convert to characters, then convert the character to numbers

```
my_data$numeric_character4 <- as.numeric(as.character(my_data$character4))
```

Using subsetting to only change some values

You can put subsetting restrictions on the result when doing a calculation, so that the calculation only applies to some of the entries

Empty spaces to zero values

If you realised that you had left entries blank in you data, and they should be 0, you could only change the blank entries in a column by using subsetting

```
my_data$fixed_numeric1 <- my_data$numeric1
my_data$fixed_numeric1[is.na(my_data$fixed_numeric1)] <- 0
```

Numerics to N/A

If you realised that you had left entries blank in you data, and they should be 0, you could only change the blank entries in a column by using subsetting

```
my_data$fixed_numeric1 <- my_data$numeric1
my_data$fixed_numeric1[my_data$fixed_numeric1 == -99] <- NA
```

In subsetting, two equals signs are used for checking if something is equal to another thing. A single equals sign has already been used for providing settings for functions.

Added symbols in numerics

If you did not remove dollar signs and commas from the orginal data before bring in to R, you need to remove them before converting a character column to numeric

```
my_data$character4 <- gsub("$", "", my_data$character4, fixed=TRUE)
my_data$numeric_character4 <- as.numeric(as.character(my_data$character4))
```

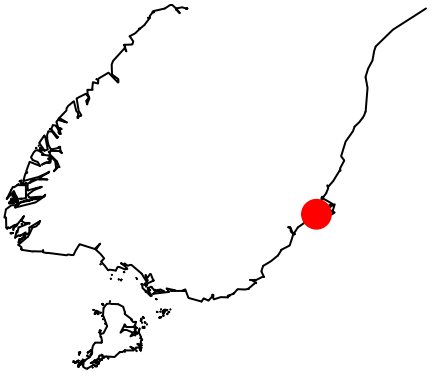
Using added libraries

Sometimes special functions and data is available in added libraries (called packages). To take advantage of this there is a two stage process. There is a one-off step of adding the library to the computer you are using. You also need to load the library in each R session you want to use the R commands and data from that library. For example there is a high quality New Zealand map in the mapdata package, which can be drawn using commands from the maps package. So to make a high quality map involves a one-off install of both packages

```
install.packages("maps", dependencies=TRUE)
install.packages("mapdata", dependencies=TRUE)
```

Then, providing I load the libraries each session I want to use them in, I can make a map

```
library(maps)
library(mapdata)
map("nzHires", xlim=c(160,174), ylim=c(-48,-44))
points(170.5028, -45.8788, pch=19, col="red", cex=2)
```



Installing at home

R is available from the http://www.r-project.org/ website, with appropriate versions for most major computer systems. Once the core program is installed on your computer system (this may need administrator access) you can add other additional packages from inside the program.

R Studio is a very popular overlay for working with R and is strongly recommended, but you need to have R already installed. R Studio is not available from the main R site, instead it is at http://www.rstudio.com