An Introduction To R

Gordon J Ross

Contents

We are going to learn the basics of the R programming language, which is currently the most common language used by statisticians to analyse data, and is increasingly being used in industry (along with other high level languages such as Python and MATLAB). This lecture will cover the basics of installing the R software, setting up the environment, and manipulating data.

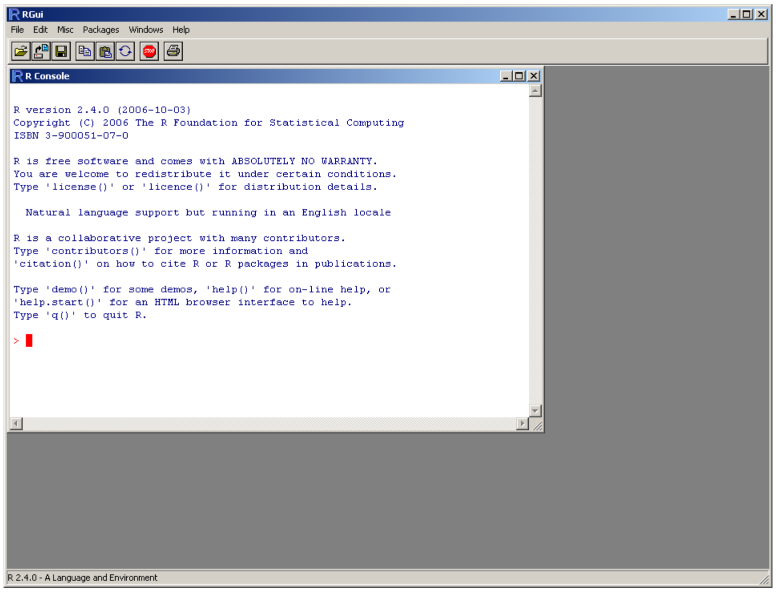
The material in this lecture is meant to be introductory. For more detail on particular aspects of R, a decent resource is: http://cran.r-project.org/doc/manuals/R-intro.pdf

1 Installing R

Windows/Mac: Go to http://cran.r-project.org/, select ”Download R for Windows/OS X” on the top right of the page, download the file, and double click it. The installation process should then begin. It is advisable not to change the installation directory unless you know what you are doing, so just use the location which the installer suggests.

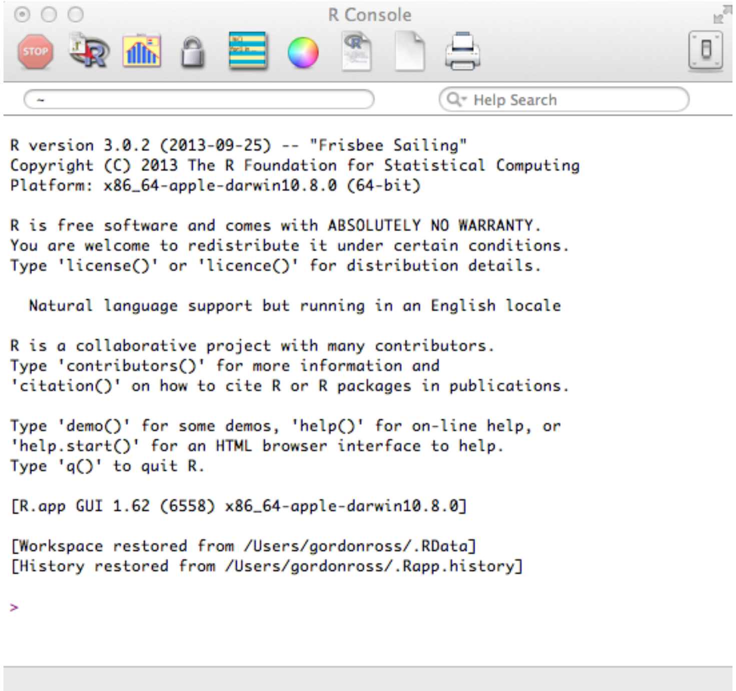
Linux: Installation varies based on your distro. For Ubuntu, see http://cran.r-project.org/bin/linux/ubuntu/. For other distros, try http://cran.r-project.org/bin/linux/, or consult Google.

After installation, you can launch R in Windows by selecting it from the Start menu. In OS X, you can find it listed as ”R” in your Applications folder. When you start up R, you should see something like this in Windows:


RGui
File Edit Misc Packages Windows Help
R Console
R 2.4.0 - A Language and Environment

```
R version 2.4.0 (2006-10-03)
Copyright (C) 2006 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

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.

>
```

or something like this in OS X



```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-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 1.62 (6558) x86_64-apple-darwin10.8.0]

[Workspace restored from /Users/gordonross/.RData]
[History restored from /Users/gordonross/.Rapp.history]

>
```

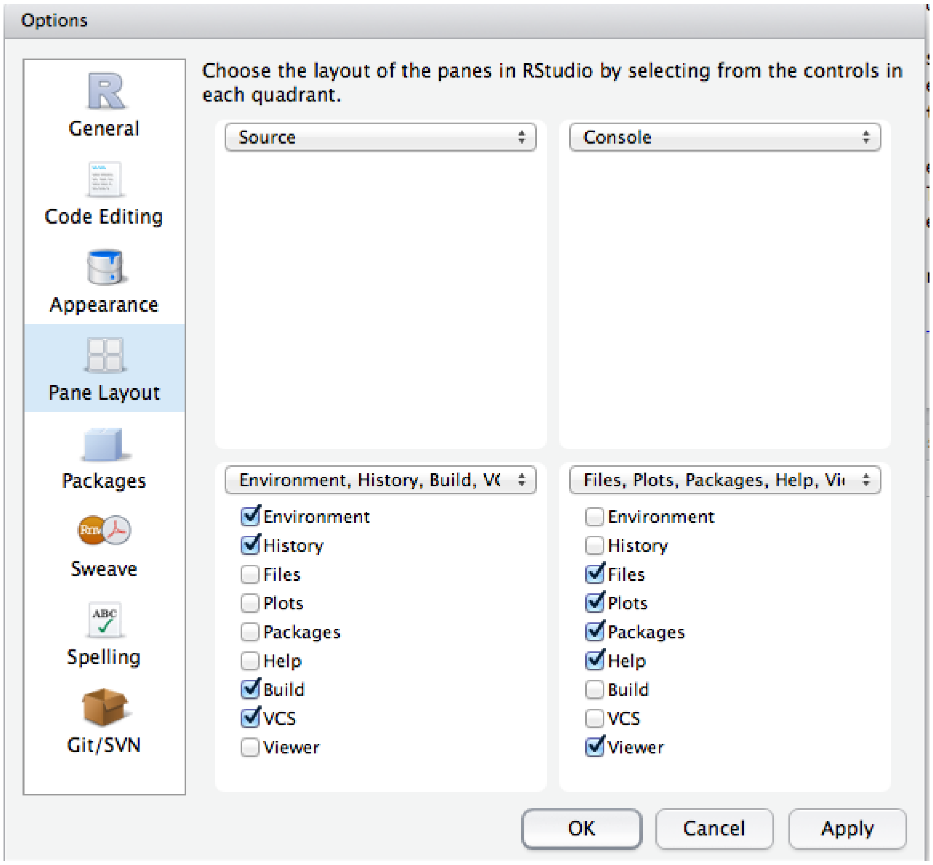
This is the standard GUI (Graphical User Interface) which comes with R, and allows you to interact with the language. The alternative way to start an R session is to launch it in a terminal window by typing ”R”. I do not advise using R in terminal mode, but note that in linux this is the default way of interacting with the R environment, since no GUI is provided in the standard installation.

1.1 Installing RStudio (optional but recommended)

The above R environment is sufficient for doing everything that you will need to do in this class. However many people (including myself) find the standard R interface a bit ugly to use, and prefer to use the RStudio program to interact with R instead. This is essentially just an an alternative graphical interface – it does not change any of the language, it only provides a nicer way to input commands and edit documents. As such, its use is entirely optional, but is recommended (it is especially recommended in linux, since as mentioned above there is no default graphical interface provided with the R installation. However if you are an emacs/vim user then you may prefer to use these instead).

To install RStudio in Windows, Mac OSX, and some linux distros (including Ubuntu), go to http://www.rstudio.com/products/rstudio/download/ and download and run the relevant file for your operating system. After installation, start RStudio, which will again be found in either your start menu in Windows, or Applications folder in OS X.

When you start RStudio, you will see multiple windows. The default window layout is a bit strange, so I advise changing the layout before you start. To do this, go to the Tools menu at the top, and select ”Global Options”. Now, select ”Pane Layout” and change the panels so they look like the below picture, with Source in the top left, and Console in the top right.



Choose the layout of the panes in RStudio by selecting from the controls in each quadrant.

Source

Console

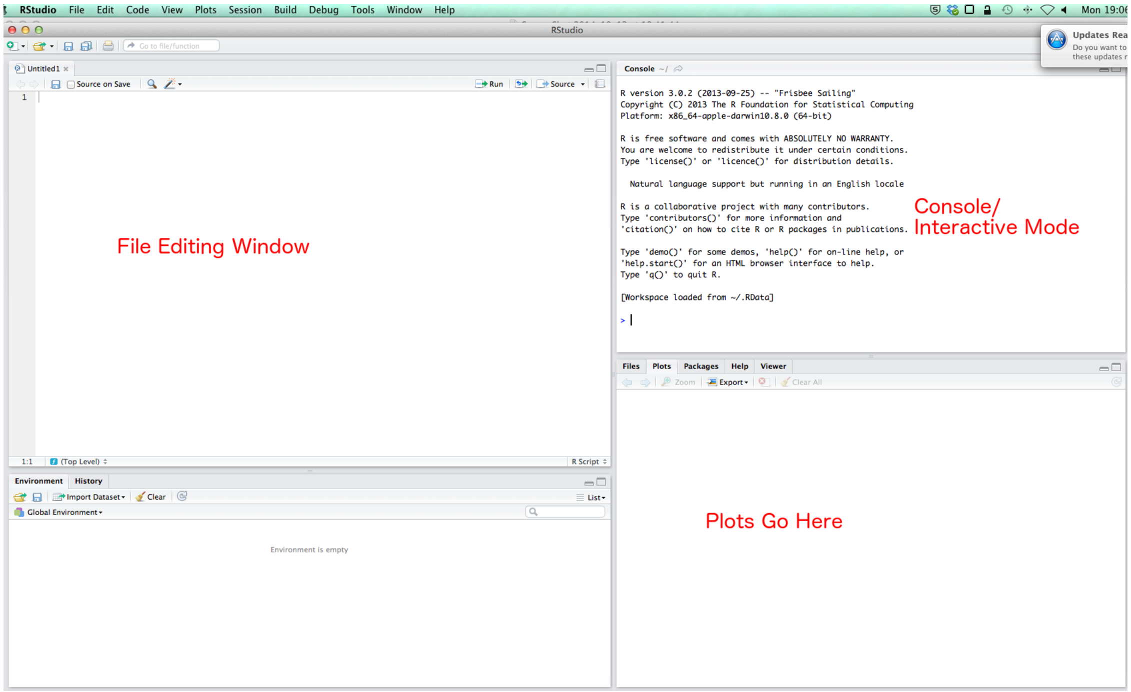
Environment, History, Build, VC

☑ Environment
☑ History
☐ Files
☐ Plots
☐ Packages
☐ Help
☑ Build
☑ VCS
☐ Viewer

Files, Plots, Packages, Help, Vi

☐ Environment
☐ History
☑ Files
☑ Plots
☑ Packages
☑ Help
☐ Build
☐ VCS
☑ Viewer

After this, select ”New R Script” in the ’File’ menu to create a new code file. Your screen should look something like this, assuming you made the window layout change:



```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin10.8.0 (64-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.

[Workspace loaded from ~/.RData]

> |
```

2 Getting Started

There are two main ways to interact with R. The first is to type commands directly into the interactive session, which is on the right of the RStudio GUI (see above picture), or in the main window if you are running standard R without RStudio. In this case, R processes the command immediately, and replies. This is ideal if you only intend to run a very small number of commands. The second method is to type commands into a text file which can be saved to disk, and then run this file (or parts of the file) in R. This latter method is usually superior, since your commands will be saved for the future. We will discuss the interactive method first.

Lets start by typing commands directly into the interactive session.

```
> 2 + 2
[1] 4
```

The text which starts with the > symbol denotes the command that I typed in to R. The line which starts with the [1] symbol denotes R's reply. Note that the # character represents a comment and R ignores everything on the line after it. So I could alternatively type:

```
> 5 * 3 #R will ignore this
```

```
[1] 15
```

Note that * denotes multiplication in R, and in almost every other programming language.

As in most programming languages, we typically write code by assigning values to variables. This can be done as follows:

```
> x <- 5 #create a variable called 'x', initialised to the value 5
> y <- 3 #create a variable called 'y', initialised to the value 3
> x + y
[1] 8
```

Variables can have any essentially any name as long as it starts with a letter, and doesn't contain any non-alphanumeric symbols.

```
> john <- 20
> sarah333 <- 2
> peter <- john * sarah333
> peter #display the value of the 'peter' variable
[1] 40
```

Note: A useful trick in interactive mode is to press the up arrow on your keyboard – this repeats the previously entered command.

2.1 Vectors

The 'x', 'y', 'john' etc variables above had single numbers assigned to them. However, variables can also be assigned vectors (lists) of numbers. We create a vector using the 'c' command

```
> x <- c(1,2,3,4) #create a vector containing the numbers from 1 to 4
> x #display the value of x
[1] 1 2 3 4
```

We can also use the colon operator which creates a vector containing all the integers between two specified numbers:

```
> x <- 5:10
> x
[1]  5  6  7  8  9 10
```

Most arithmetic functions can operate directly on vectors:

```
> x <- 5:10
> x * 2 #multiply all elements of x by 2
[1] 10 12 14 16 18 20

> x + 1 #add 1 to all elements of x
[1] 6 7 8 9 10 11
```

We can also combine different vectors. R does this in an element-wise way, which is easier to show by example:

```
> x <- c(2,4,8)
> y <- c(1,5,10)
> x + y
[1] 3 9 18
> x*y
[1] 2  20 80
```

i.e. x+y adds the first element of x to the first element of y, and assigns this to the first element of the answer. Then the second element of x is added to the second element of y, and becomes the second element of the answer, and so on. Multiplication works in the same way, as does subtraction and division (check this – the symbol for division is x / y).

Finally we can select specified elements from a vector by using the [] notation. If we write x[2:3] for example, this returns the second and third elements of the x vector:

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

2.2 Non-interactive (Text-File) Mode

The alternative to interactive mode is to write R commands in a text document. This is often preferable since it allows the commands to be saved for future use and reference. The easiest way to do this is to create a new text file from within the the R (or RStudio) GUI by going into the File menu and selecting "New". This creates a new window which you can type commands into, which is shown in the left pane of RStudio, or as a separate window in vanilla R. Go into the File menu and select "Save" (or press Control-S on your keyboard) to find a location to save the file to.

In this mode, pressing return after typing a line no longer causes the command to be run immediately. To run commands that you have typed, instead select the commands you want to run using your mouse/trackpad, right click, and select 'Run'. If you are using RStudio then a convenient keyboard shortcut is to select the commands, then hold the Control key, and press Enter/Return.

When you have finished with a textile, you can save it somewhere on your hard drive and then load it back at a future date (using the Load command in the File menu).

3 Basic Data Analysis

3.1 Categorical Data

Having covered the basics of installing and using R, we now discuss how to use it to perform basic data analysis. We first consider some categorical data. Suppose that 20 people are randomly selected and asked which football team they support. We would like to produce some plots of the resulting data.

The first thing to do is to load the data into R. This is usually done by saving it as a text-file and reading this file into R. We will cover this in a future lecture but its quite fiddly at first, so for now lets just assume that we type the data into R directly (obviously this will not be practical when working with large amounts of data). We create a variable called 'surveydata' which is assigned a vector of character strings containing the data – note that we could call the variable anything, but using a descriptive name like 'surveydata' helps you remember what the variable actually is.

```
> surveydata <- c("Man Utd", "Liverpool", "Arsenal", "Everton",
  "Man Utd", "Chelsea", "Everton", "Arsenal", "Arsenal", "Man Utd",
  "Man Utd", "Man Utd", "Arsenal", "Liverpool", "Man City", "Liverpool",
  "Man City", "Spurs", "Liverpool", "Man City" )
```

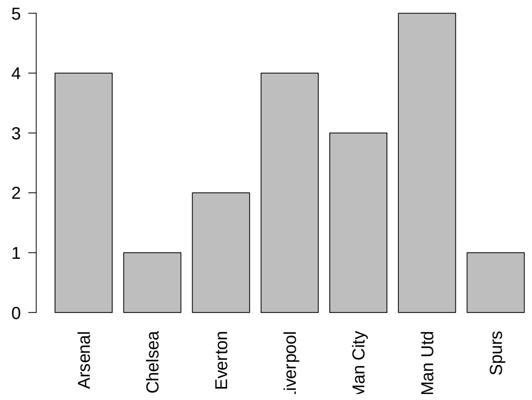
The easiest way to summarise categorical data is to use the table() function. This provides a list of all the different values present in the data, along with the number of people who have each value:

```
> table(surveydata)
surveydata
  Arsenal   Chelsea   Everton Liverpool  Man City   Man Utd     Spurs
        4         1         2         4         3         5         1
```

So we can see that 4 people replied 'Arsenal', 1 replied 'Chelsea', and so on. To plot the data, we can use the barplot() function:

```
> barplot(table(surveydata), las=2)
```

This produces the following plot:

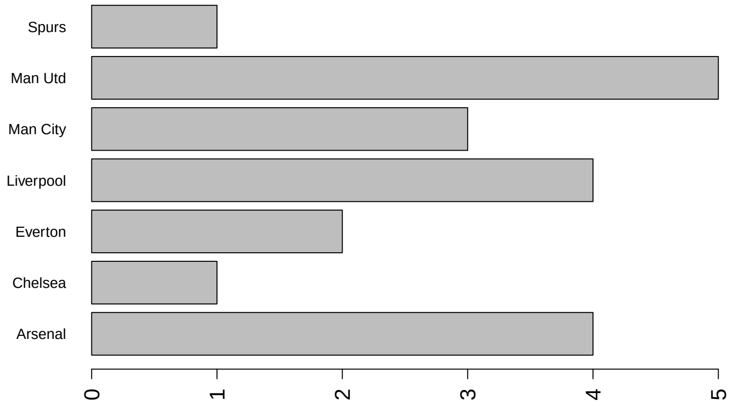


You might wonder what the 'las=2' part means. This is an optional argument to the barplot function which tells it to display the names of each category with the text flowing upwards, rather than left-to-right. Try running the barplot command without this argument to see the difference.

In general, most R functions can take optional arguments which modifies their default behaviour. For example, suppose we want to draw the same barplot, but horizontally rather than vertically. This can be done by supplying the 'horiz=TRUE' argument to the barplot function:

```
> barplot(table(surveydata), las=2, horiz=TRUE)
```

which produces the following:



Note that 'cex.names' is another optional argument which controls the size of the text used on the bar labels, The default size is 1, so setting it to 0.7 makes the text smaller

How do we know which optional arguments a function takes? In R, this is contained in the function documentation. To access this, simply type the name of the function with a question mark before, into the interactive session:

```
> ?barplot #view the documentation for the barplot function
> ?table
```

This is such an important point that I'm going to repeat it:

> To get a detailed description of a particular R function, along with a list of its optional arguments and what they do, type the name of the function with a question mark before it, into the interactive session

3.2 Ordinal Data

We now discuss ordinal data, which (recall from Lecture 1) is numerical data that has a discrete (restricted) set of values rather than being continuous – for example, test scores which can only take integer values like 50 or 51, without fractions like 50.5.

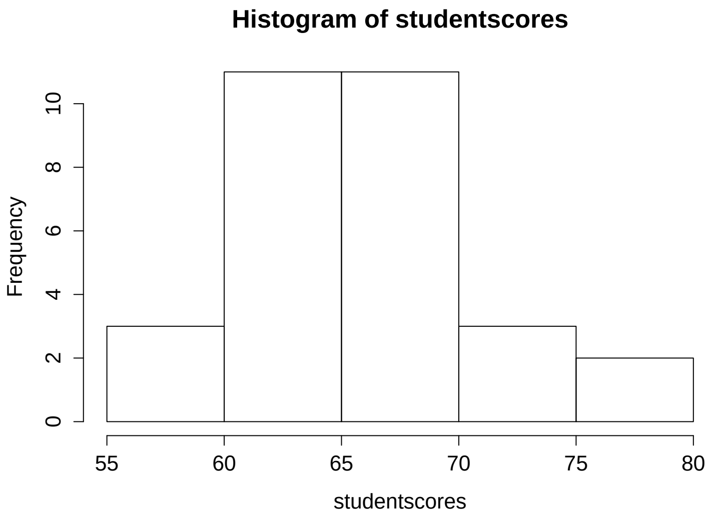
Suppose that we have the exam scores obtained by 30 randomly selected students that we wish to analyse. As before, we will (for now) enter the data into R directly:

```
> studentscores <- c(64, 64, 68, 68, 62, 61, 67, 69, 64, 69, 67, 62, 67,
  59, 72, 75, 63, 60, 68, 64, 77, 65, 68, 65, 61, 66, 56, 72, 66, 76)
```

We can plot this data as a histogram using the hist() function:

```
> hist(studentscores)
```

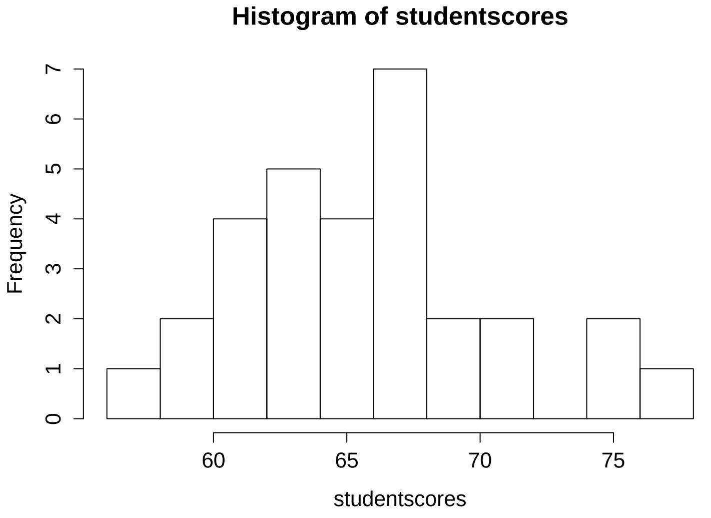
This produces the following plot:



Note that R automatically chooses the number of bins to use. This choice is often sensible, but sometimes we may wish to set the number of bins manually instead. This can be done using the 'breaks' optional argument, which allows us to supply the number of bins. For example, to display the data using 10 bins we use:

```
> hist(studentscores, breaks=10)
```

which produces:



As well as plotting the data, we can also compute summary statistics of interest.

For example, to get the average (mean/expected) score we can use:

```
> mean(studentscores)
[1] 66.16667
```

Remember that the mean is defined as

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

where 'n' is the length of the vector, which is 30 in this case since there are 30 students. We could hence compute the mean manually instead, using the 'sum' function, which returns the sum of all the elements in a vector:

```
> sum(studentscores)/30
[1] 66.16667
```

Having to manually enter the length ('n') here is inelegant, so we can instead use the length() function which returns the length of a vector:

```
> sum(studentscores)/length(studentscores)
[1] 66.16667
```

I will now list some other commonly used summary statistics for numerical data. Most of these were discussed in Lecture 1

```
> var(studentscores) #computes the variance of the scores
[1] 24.28161

> sd(studentscores) #computes the standard deviation of the scores.
[1] 4.927637

> max(studentscores) #returns the maximum score
[1] 77

> min(studentscores) #returns the minimum score
[1] 56

> median(studentscores)
[1] 66
```

The function min() gives us the smallest score. Suppose we wanted to find the second smallest score instead. To do this easily, we can use the R function sort() which returns the scores in sorted order, starting with the smallest

```
> sort(studentscores)
 [1] 56 59 60 61 61 62 62 63 64 64 64 64 65 65 66 66 67 67 67 68 68
     68 68 69 69 72 72 75 76 77
```

If we wanted to sort the scores starting with the largest, we can use the decreasing=TRUE optional argument to the sort function:

```
> sort(studentscores, decreasing=TRUE)
 [1] 77 76 75 72 72 69 69 68 68 68 68 67 67 67 66 66 65 65 64 64 64
 64 63 62 62 61 61 60 59 56
```

4 Continuous Data and Probability DIstribtuions

4.1 Exponential Distribution

When building statistical models, we will use various probability distributions such as the Exponential distribution and the Normal (Gaussian) distribution. Each distribution has a set of R functions which handle simulating from, and evaluating this distribution. The usual convention in R that applies to most probability distributions is that the function for simulating from the distribution with the letter 'r', followed by the abbreviated name of the distribution. The density function similarly has the first letter 'd', and the probability distribution has the first letter 'p'.

So for the Exponential distribution the functions are:

- **rexp(n,lambda)** – simulates n observations from the Exponential(λ) distribution
- **dexp(x,lambda)** – evaluates the Exponential(λ) distribution at point 'x'
- **pexp(x,lambda)** – evaluates $p(X < x)$ where X is a random variable with an Exponential(λ) distribution.

Recall that the Exponential distribution has the functional form:

$$f_X(x) = \lambda e^{-\lambda x}$$

The dexp() function evaluates this for a given value x. This is literally the same as substituting 'x' into the above formula. For example, lets say $\lambda = 0.1$. Verify the following two expressions are the same:

```
lambda <- 0.1
x <- 1
dexp(x,lambda)
lambda * exp(-lambda*x)
```

(note as a mathematical aside that 'exp(-lambda*x)' is how to implement $e^{-\lambda x}$ in R. Specifically, both exp(z) and e^z denote the exponential function)

The 'rexp' function lets us simulate data from an Exponential distribution. For example, lets say we believe that the times between earthquakes in some region follow an Exponential(0.1) distribution. We can simulate sample values from this distribution (i.e. generate random numbers which obey this probability law) using the rexp() function:

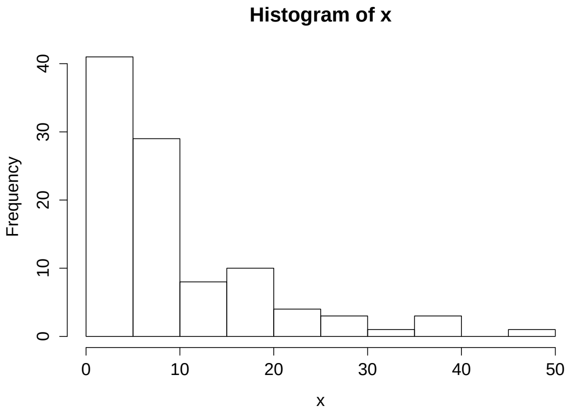
```
lambda <- 0.1
x <- rexp(100, lambda) #draw 100 observations from the Exponential(0.1) distribution
```

We can view the first 10 observations that were generated using the array slice notation [] introduced above:

```
> x[1:10]
 [1]  1.1699895 15.8308296  1.4250965 13.1608859  5.1762164 17.6814138  6.6719528
 [8]  2.4727945  0.6367363 35.3287867
```

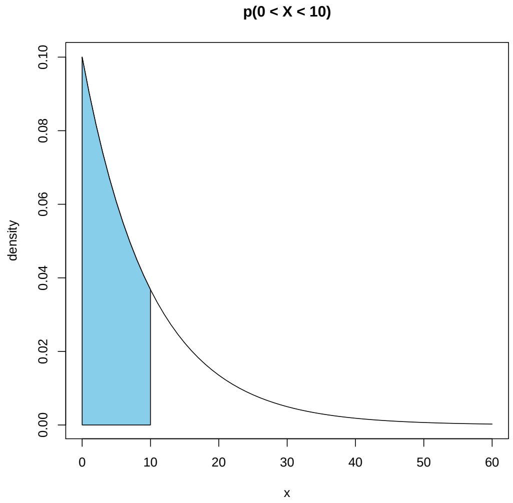
Your numbers will differ from these because they are random samples from the distribution! We can also plot the simulated data as a histogram:

```
 > hist(x)
```



Although your numbers will be slightly different, the general shape should be the same. Note that it is close to the exponential decay curve we saw in the lecture. It is more jagged than the smooth curve, because this is a fairly small sample. If you repeat the above simulation but generate 10000 data points rather than 100, you will see a much smoother histogram.

Many of the questions we wish to answer about random processes involve calculating the area under the curve of the probability distribution. For example, suppose that we believe that the time-between-earthquakes on a particular fault follows an Exponential(0.1) distribution. If an earthquake occurs today and we wish to ask "What is the probability of the next earthquake occurring within 10 years?", then this is equivalent to asking for the probability of a random variable with an Exponential(0.1) distribution having a value between 0 and 10. Recall that this is given by the area under the Exponential(0.1) distribution curve:



This area can be found using the pexp function.

> The function pexp(x, lambda) returns the probability of a random variable with the Exponential(λ) distribution having a value less or equal to x

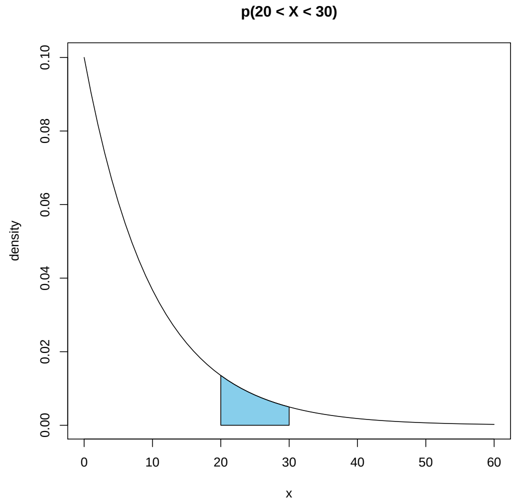
So:

```
> pexp(10,0.1)
[1] 0.6321206
```

The probability of the next earthquake happening within 10 years is hence 0.63, or 63%. Of course, this is the same answer we get if we integrate the Exponential distribution to find the area mathematically:

$$\int_0^{10} 0.1e^{-0.1x}dx = 0.63$$

Now suppose we instead wish to find the probability of the next earthquake occurring in between 20 and 30 years time. This can be written $p(20 < X \leq 30)$ and is equal to the following area:



This can again be found using the pexp() function. Note that $p(20 < X \leq 30)$ is equal to $p(X \leq 30) - p(X \leq 20)$ (this might not be obvious – if it isn't, then keep thinking about it and perhaps try to see it geometrically from the above picture. The shaded area is equal to the area that is less than 30, minus the area that is less than 20). We can again find this using R:

```
> pexp(30,0.1) - pexp(20,0.1)
[1] 0.08554821
```

So the chance of the next earthquake happening in between 20 and 30 years is around 8.6%. This is again the same number that we computed in the lecture notes by hand, using the integral:

$$\int_{20}^{30} 0.1e^{-0.1x} dx = 0.0855$$

In summary:

To find the probability that the random variable X takes on values in the interval $[a, b]$, i.e. that $p(a < X \leq b)$, use

```
> pexp(b,lambda) - pexp(a,lambda)
```

4.2 Normal Distribution

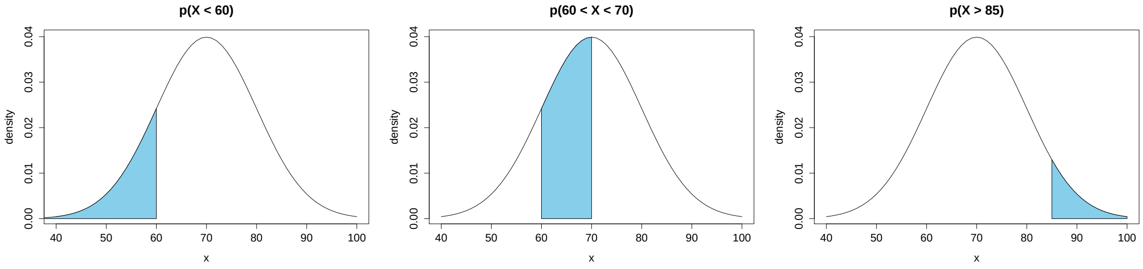
We now discuss the Normal distribution. This is very similar to the Exponential distribution, and the relevant functions are now 'rnorm', 'dnorm', and 'pnorm' (note that the naming convention based on the first letter is exactly the same as for the Exponential distribtuion! This is the standard naming convention used in R).

- **rnorm(n,mu,sigma)** – simulates n observations from the N(μ, σ) distribution
- **dnorm(x,mu,sigma)** – evaluates the N(μ, σ) distribution at point 'x'
- **pnorm(x,mu,sigma)** – evaluates $p(X \leq x)$ where X is a random variable with a N(μ, σ) distribution.

Most of the analysis of data which has a Normal distribution parallels the above Exponential case. For example, suppose we wish to study the weight (in kilograms) of the population of male UCL students. Lets first suppose that we have theoretical grounds to believe that the true population distribution of UCL male weights is Normal, with a mean of 70 and standard deviation of 10. Based on this, we would like to ask:

1. What is the probability that a randomly selected male will weigh less than 60 kilograms?
2. What is the probability that a randomly selected male will weigh between 60 and 70 kilograms?
3. What is the probability that a randomly selected male will weigh more than 85 kilograms?

These questions are again answered by considering the areas under the (known) probably distribution, which is $N(70, 10)$. The relevant areas are:



We can compute this in R in a similar way as we done with the Exponential distribution. For Question 1), we have:

```
> pnorm(60,70,10)
[1] 0.1586553
```

So the probability of a randomly selected male weighing less than 60 kilograms is about 0.16 (16%). Similarly for Question 2), the probability of a randomly selected male weighing between 60 and 70 kilograms is:

```
> pnorm(70,70,10) - pnorm(60,70,10)
[1] 0.3413447
```

where we have used the result shown in the box towards the end of the previous section. Finally, for Question 3), recall from lectures that for a random variable X with any probability distribution, and for any value a, we have $p(X > a) = 1 - p(X \leq a)$, and so:

```
> 1 - pnorm(85,70,10)
[1] 0.0668072
```

The probability of a randomly selected male weighing more than 85 kilograms is 0.067

As in the Exponential case, in practice we will usually not know the true values of μ and σ that govern the process which we are studying. In other words, we may be reluctant to simply accept that $\mu = 70$ and $\sigma = 10$. Suppose instead that we went out and actually weighed 20 male UCL students. We observe the following data:

```
> weights <- c(63.73546, 71.83643, 61.64371, 85.95281, 73.29508, 61.79532,
    74.87429, 77.38325, 75.75781, 66.94612, 85.11781, 73.89843, 63.78759,
    47.85300, 81.24931, 69.55066, 69.83810, 79.43836, 78.21221, 75.93901)
```

The observed mean and standard deviation are:

```
> observedMean <- mean(weights)
> observedMean
[1] 71.90524
> observedSd <- sd(weights)
> observedSd
[1] 9.132537
```

If we wish to make predictions based on the observed mean and standard deviation, rather than simply assuming they are equal to 70 and 10 respectively, then we can simply substitute in these observed values in the pnorm() function. Note that this is implicitlyy based on the same moment-matching technique that we discussed in Lecture 2 for the Exponential distribution – μ in the Normal distribution formula is the population mean, so matching it to the empirically observed mean gives us the estimate:

$$\hat{\mu} = \bar{X}$$

Similarly σ in the Normal distribution formula is the population standard deviation, so matching it to the observed standard deviation gives the observed standard deviation as its estimator (we will discuss this in more detail in a future lecture). The estimated answers to the 3 questions above can hence now be calculated as:

```
> pnorm(60, observedMean, observedSd)
[1] 0.09618382

> pnorm(70, observedMean, observedSd) - pnorm(60, observedMean, observedSd)
[1] 0.3211883

> 1 - pnorm(85, observedMean, observedSd)
[1] 0.0758064
```

which are close to, but not exactly equal to, the values previously found under the assumed $N(70, 10)$ distribution

5 More Data Analysis

We now discuss how to use R to carry out some slightly more realistic data analysis. Note: some of this material depends on concepts which will not be covered until later in the course.

First, we need to load in some data. Previously we entered data into R by using the c() command, however typing in the numbers manually is obviously not practical when working with large amounts of data. The more usual method is to load in the data from a text file. For this class, we will study the series of Italian earthquake data discussed in the lecture. Download the "Italy.csv" file from the following URL:

`https://www.dropbox.com/s/yg8aowakndzvmev/Italy.csv?dl=1`

and save this somewhere, e.g.

`C:\Desktop`

on windows, or

```
 ~/Desktop/
```

on Mac OS X. This file can then be read into R using:

```
earthquakedata <- read.csv("~/Desktop/Italy.csv", sep=',') #OS X
earthquakedata <- read.csv("C:\\Desktop\\Italy.csv", sep=',') #windows
```

(note the double backslash in the Windows version, R needs this for file paths on windows, a single slash will not work for technical reasons. It is not needed in OSX/Linux).

if you view this object, it should like like:

```
> earthquakedata

 [1,]  249.000        5.5
 [2,]  560.000        5.2
 [3,] 1198.000        4.7
 [4,] 1257.000        4.5
 [5,] 1279.000        5.3
 ...
```

for 425 rows. Each row represents one earthquake. The first column is the time of the earthquake, measured by the number of days after Jan 1st 1970 when the earthquake happened. The second column is the magnitude of the earthquake. So for example, the second earthquake occurred 560 days after Jan 1st 1970, and was magnitude 5.2. We are interested in the time between successive earthquakes, so we create a variable containing this information:

```
times <- diff(earthquakedata[,1])
```

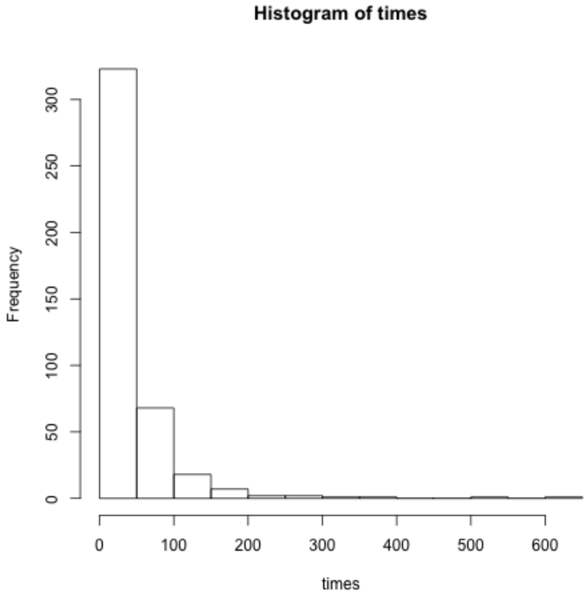
The earthquakes[,1] command tells R to select the first column of the earthquakes matrix, and the diff() command tells R to take the differences, giving us the time between the earthquakes (see the ?diff manual page if this is confusing). We end up with:

```
> times
[1] 311.00000 638.00000  59.00000  22.00000  36.00000 134.04167  35.00000
  62.95833 18.00000 ...
```

So there was 311 days between the first and second earthquake, 638 days between the second and third, and so on. We can plot this data as a histogram using the hist() function:

```
> hist(times)
```

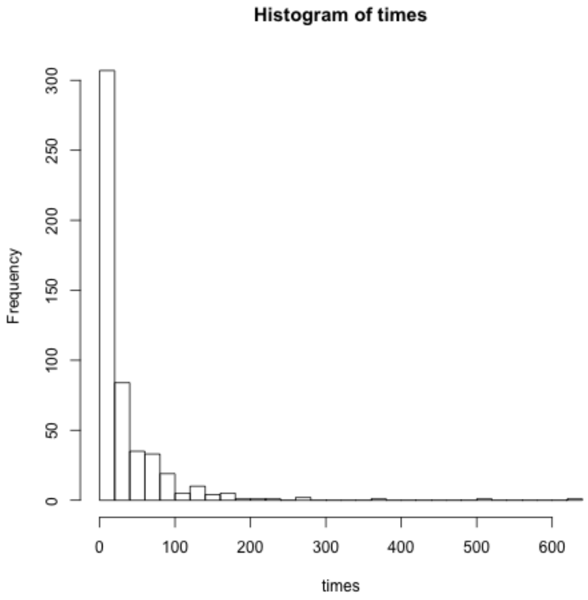
This produces the following plot:



Note that R automatically chooses the number of bins to use. This choice is often sensible, but sometimes we may wish to set the number of bins manually instead. This can be done using the 'breaks' optional argument, which allows us to supply the number of bins. For example, to display the data using 30 bins we use:

```
> hist(times, breaks=30)
```

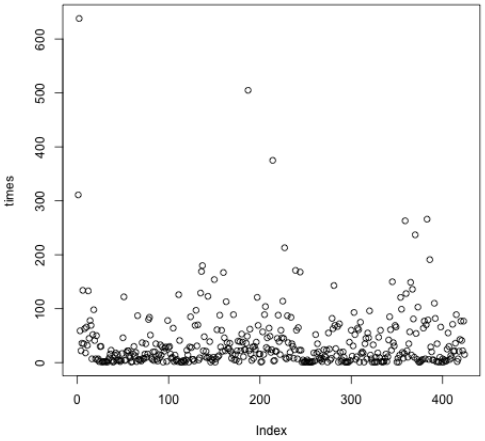
which produces:



We can also plot the raw series of times, which is often useful:

```
> plot(times)
```

which produces:



where we can see the clusters which we noticed in the lectures.

As well as plotting the data, we can also compute summary statistics of interest. For example, to get the average (mean/expected) score we can use:

```
> mean(times)
[1] 37.57783
```

other summary statistics are:

```
> var(times) #variance
[1] 3594.733

> sd(times) #standard deviation
[1] 59.95609

> max(times) #maximum value
[1] 638

> min(times) #minimum value
[1] 1
```

If we wish to fit an Exponential distribution to this data, we need to first estimate λ. This is done in the same way as before:

```
> lambdaest <- 1/mean(times)
> lambdaest
[1] 0.02661144
```

So we estimate as $\hat{\lambda} = 0.027$. We can use this to perform a Kolmogorov–Smirnov test, to assess whether the Exponential distribution is a good model for this data. This is done using the 'ks.test' function, which takes several arguments. The first is the empirical data to be tested. The second is the name of the theoretical distribution which we would like to test if the data came from, which is 'pexp' in our case since we are testing if the times are Exponentially distributed. The final arguments are the estimated parameters of the distribution, which is our lambast:

```
> ks.test(times,'pexp',lambdaest)

One-sample Kolmogorov-Smirnov test

data:  times
D = 0.1547, p-value = 3.109e-09
alternative hypothesis: two-sided
```

The important part here is the p-value, which is extremely low, signalling that our Exponential assumption should be rejected (note: there are also some error messages about having ties in the data, but these can be ignored).

Finally, we will show how to calculate the log likelihood of this data. Recall from the lectures this is defined as:

$$LL(X_1, \dots, X_n) = \sum_{i=1}^{n} \log(f(X_i))$$

which for the Exponential distribution is:

$$LL(X_1, \dots, X_n) = \sum_{i=1}^{n} \log(\lambda e^{-\lambda X_i})$$

we can compute this using the 'dexp' function:

```
loglik <- sum(dexp(times, lambdaest,log=TRUE))
```

This needs some unpacking. The 'dexp' function computes $\lambda e^{-\lambda X_i}$ for each element of the vector 'times'. The 'log=TRUE' argument tells R to take the logarithm of this, after computing it. The 'sum' function then adds all these together. We can see how the sum() function works in a much simpler case:

```
> sum(c(2,3,4)) #evaluate 2+3+4
[1] 9
```

Note as a technical point that we could have calculated the log likelihood by instead taking the raw values, multiplying them together, and taking the log of the total, as in the following:

```
lik <- prod(dexp(times, lambdaest, log=FALSE))
loglik <- log(lik)
```

Although in theory this should give the same answer as above, in practice this is very bad programming style because multiplying together lots of very small numbers will often result in numerical underflow, which causes errors. Without getting into too much needless technical detail, you should almost always calculate log likelihoods by using the log=TRUE argument, and summing them, rather than taking the product of the raw values and taking the log at the end.