Computing Workshop

Notebook:	Computing		
Created:	16/10/2020 14:32	**Updated:**	18/10/2020 10:06
Author:	Frances Britton		

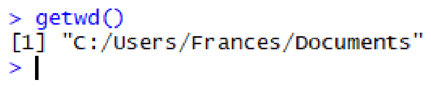
All code is written in black, explanations of each line are written in green.

Working Directory (where things are saved)

> getwd() This shows you where work is saved. It displays the result below.

[1] "Z:/My Documents/R"

Mine looked like this:



```
> getwd()
[1] "C:/Users/Frances/Documents"
> |
```

> setwd("Z:/My Documents/R") Sets the WD to what you want

> list.files() This returns a list of files in the WD

> q() Quit R - doesn't save workspace!

Mini Calculations

Type basic functions into the console and you will get an answer

```
> 2*3
   [1] 6
```

```
> pi
   [1] 3.141593
```
It actually has more decimals of pi, it just doesn't show it.

R also knows trig (radians)

If you want a single string, like a word or sentence you need to put quote marks e.g
```
> "Mithrandir"
   [1] "Mithrandir"
```

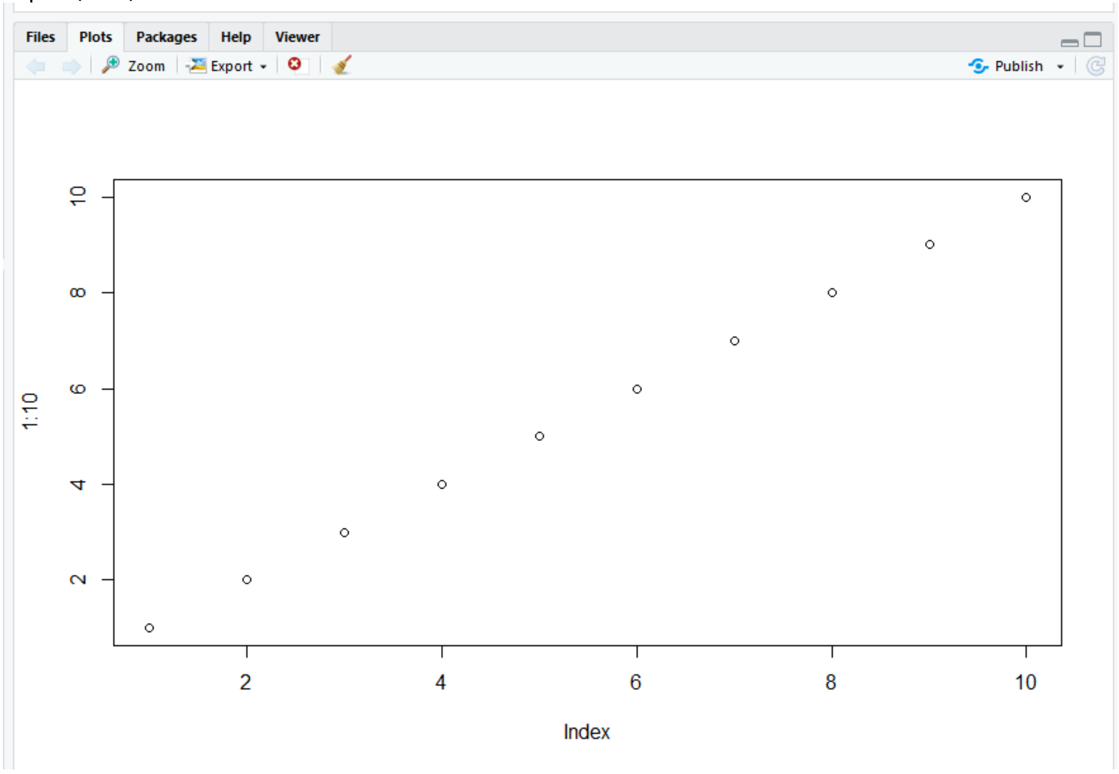
The [1] is to show you that the result is a 1x1 array, because R works in arrays/matrices. Single answers like 6 or Mithrandir are 1x1.

1:10 makes a set of 10 values (1, 2, 3,...,10), this is called a 1D array or a vector.
Mine looked like this

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
>
```

Functions

Functions are usually in the form function(...), for example continuing from the previous demo
> plot(1:10)



Defining Variables
> x <- 3*pi   This creates the variable x, which is 3 x pi (9.424...). When we use x, we are using this number instead.

```
> x <- 3*pi
> x
[1] 9.424778
> print(x)
[1] 9.424778
> |
```

We can also use the print function to get x as well. You can also use the = sign instead of <- , but it might be easier to understand because it specifically means assign in this case. = can be used for more stuff.

You can store a set of numbers like a vector.

```
> x = 1:10
> x
 [1]  1  2  3  4  5  6  7  8  9 10
```

We can make a 2nd variable using the 1st. Like y= f(x) for the values of x.
> y = x+2
> y
  [1] 3 4 5 6 7 8 9 10 11 12
> y[3] returns the 3rd value. R starts from 1, not 0 like Python.
  [1] 5

It's important to note that y has computed x + 2, then stored the values only. If we change x now, y won't change with it. You need to keep updating!

> y = seq(1, 2, length=5) creates a <u>sequence</u> 5 values long of evenly spaced values between 1 and 2.

```
   [1] 1.00 1.25 1.50 1.75 2.00
```

> x +y

```
   [1]  2.00  3.25  4.50  5.75  7.00  7.00  8.25  9.50 10.75 12.00
```

Here y is only 5 values long, but x is 10, so what R did is repeat the y values again.

A number stored on its own is just a scalar. If it's not an integer (a decimal), it is a floating point number. A set of numbers is a vector.

> x <- c(1.2, 1.3, 1.4, 1.5) c combines several numbers into one object, so scalars into a vector.

This is better for if you have some uneven weird numbers.

> c(1,2,3) + c(1,2,3)

[1] 2 4 6 This works, because each vector has 3 values. All match up well!

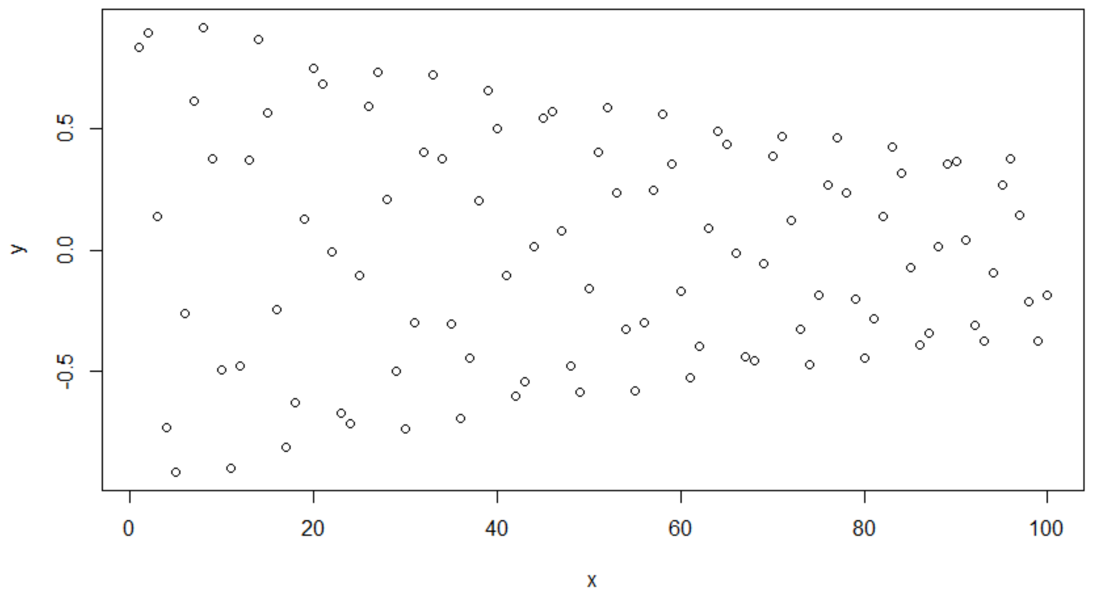
Writing Maths

> x = 1:100 x is now every integer between 1 and 100

> y <- sin(x)*exp(-x/100) This is

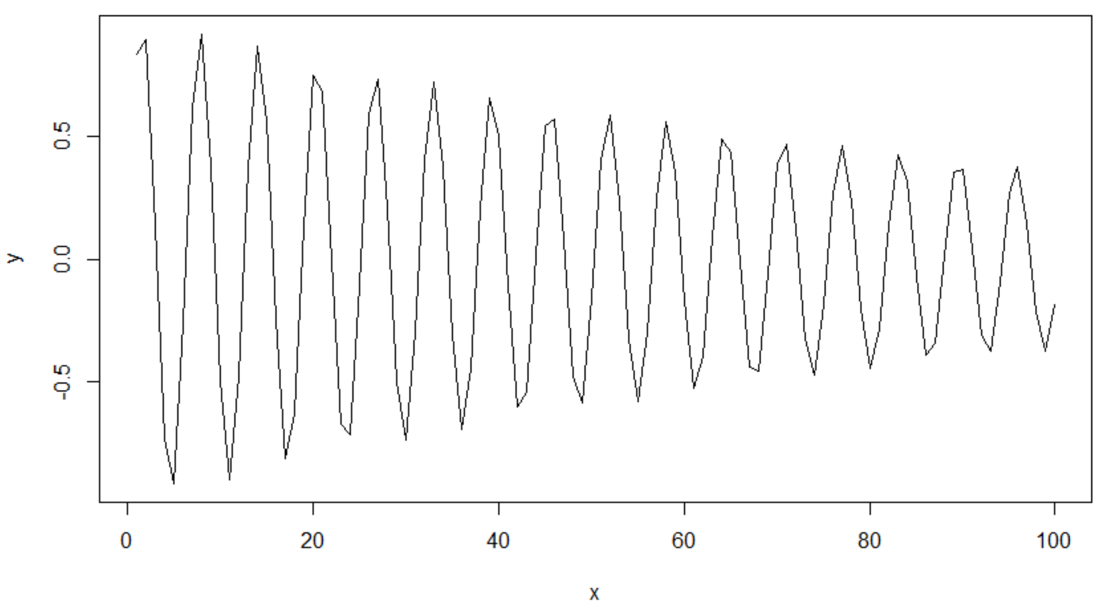
$$y = \sin(x)e^{-x/100}$$

> plot(x, y) plots variable against variable on a graph. Looks very 1995 html style.

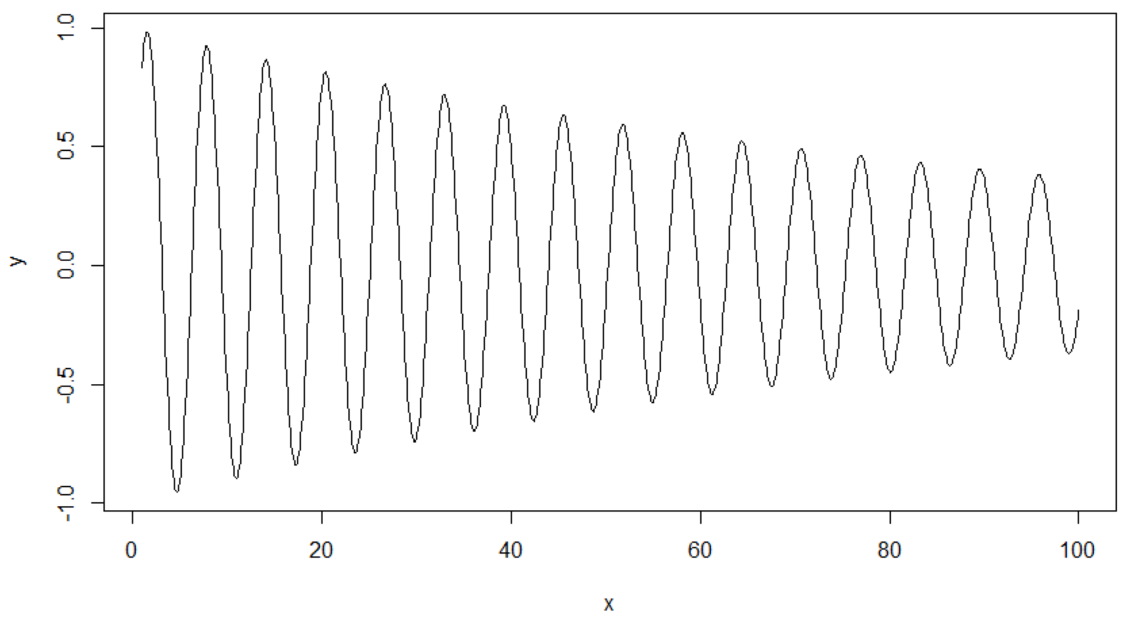


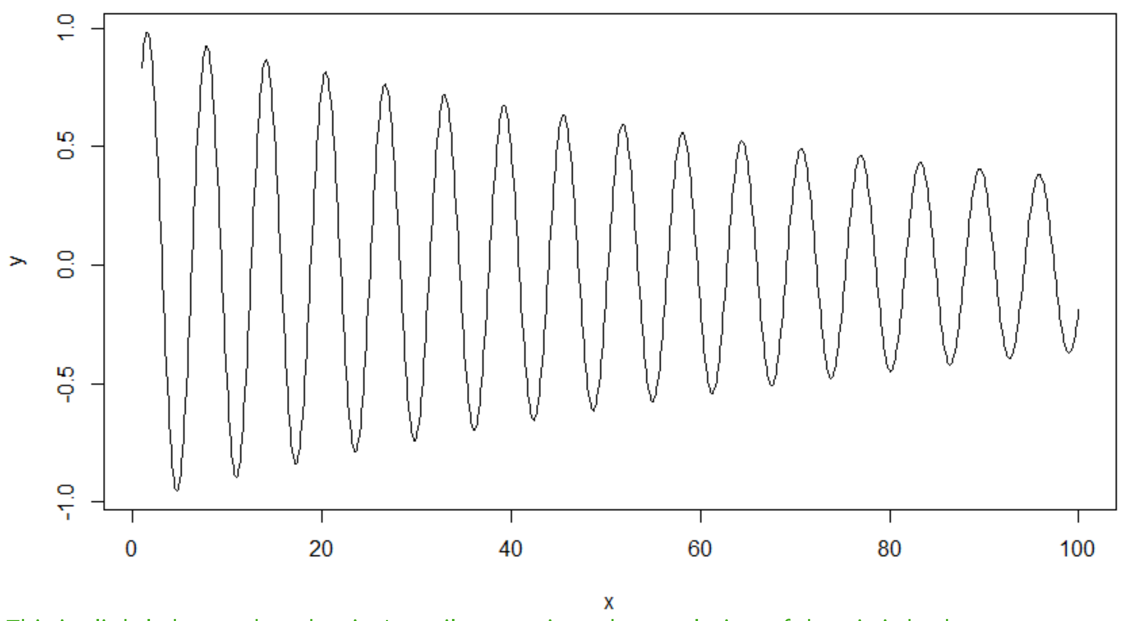
It looks great! Turn it into a line graph by adding type="l" to the function. l is lowercase L.

> plot(x, y, type="l")



```
> x = seq(1, 100, by=0.1)
```
Change x to a sequence with intervals 0.1. For much finer data.





This is slightly better, but that isn't easily seen since the resolution of the pic is bad.

Special Case: Arc functions

> y <- asin(x) asin is arcsin. NaN = Not A Number, like infinity, or in the case of arcsin or arcos, because they go from -1 to 1, the rest aren't technically numbers. R is warning of this.
In asin(x) : NaNs produced

Other functions:
acos(x)
tan(x)
sinh(x)
log(x) defaults to ln
log10(x)
sqrt(x)
x^(#) or #^x

Files

> mydata <- read.table("") to load a simple text file and call the data mydata. File name in quotes. If it has a header, put header=TRUE

> plot(mydata$force, type="l") plot the force data from mydata. Line graph type as before.
> length(mydata$force)
   [1] 320 shows how long this column of data is, 320 numbers in it.

> mydata$x <- 1:320 add a new column into mydata called x and label it from 1 to 320. Plotting the above will look similar to the first one with just mydata$force.

Columns with no labels/headers default to V1, V2, V3, ...

> write.table(mydata, "mydata.txt", row.names=FALSE) save mydata to the file mydata.txt

Plot Command Additions

row.names = FALSE switches off the naming of rows.
In the plot command, add into the brackets bty="n" for box type. Also l, u, o.
col="blue" changes the points/lines to blue. Find colours by writing (colours) in console.

lwd=2 line width is 2x original
cex.axis=1.5 makes numbers on axis 1.5 x bigger.
xlim=c(0,10) upper+lower limits of x axis.
> abline(a=c, b=m) plots linear y=mx+c line

> y.error <- 1.0
> segments(x, y-y.error, x, y+y.error) draws line segments between (x, lower error) and (x, higher error)
> result <- lm(y ~ x) linear model of y as a function of x a.k.a line of best fit. Very useful!

> summary(result) gives more detailed informative summary of result (thing above)
> abline(result) finds the a value and b value for you.

label the x and y axis with xlab="" or ylab=""
There are loads more, just type ?plot or ?par
plot() is a high level command. It is its own thing. Low level commands like segments() or abline() add to plot() or other high level commands.

To save the plot, use the export menu to save it.