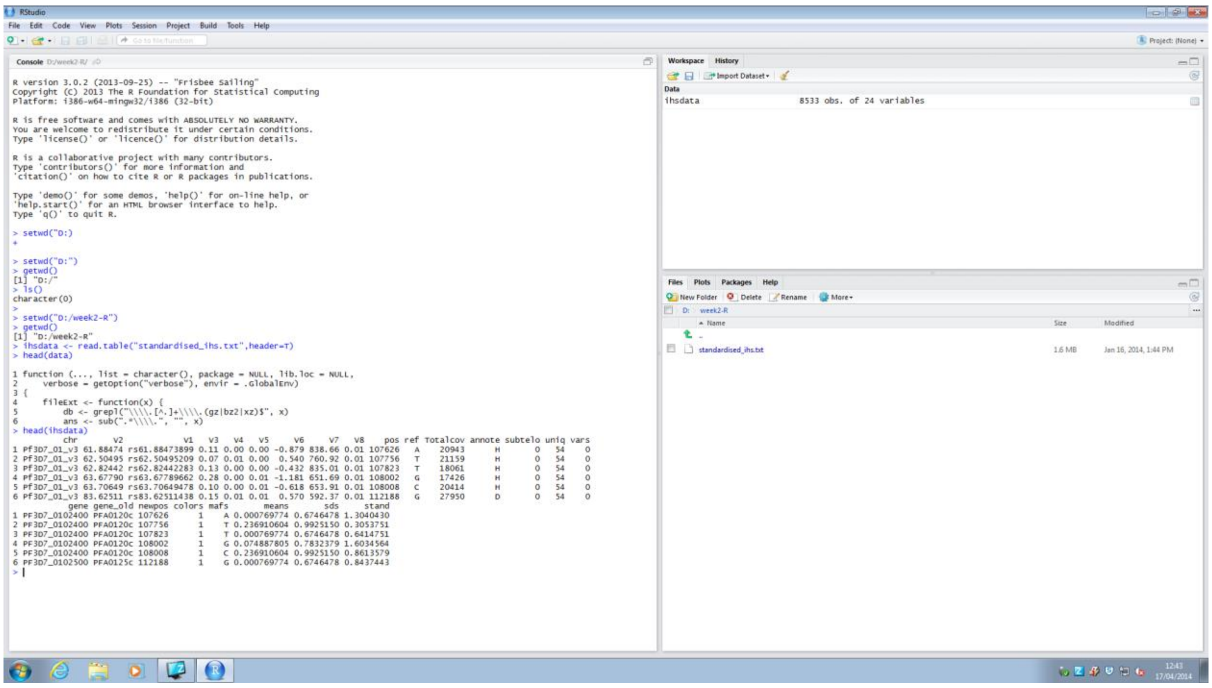
A brief introduction to R – 2017

Overview of R

R is a powerful and free scripting language and environment designed for statistical analysis and visualisation of data. It is suited to the use of large datasets, particularly those formatted along the lines of a spreadsheet. Due to its widespread use guidance and tutorials are easy to find online so if you run into a problem it is likely that a quick web search will yield a solution. For this introduction we shall be using R Studio, which can be downloaded for free from https://www.rstudio.com/.

Layout of R studio



RStudio
File Edit Code View Plots Session Project Build Tools Help
Go to file/function
Project: (None)
Console D:/week2-R/
Workspace History
Import Dataset
Files Plots Packages Help
New Folder Delete Rename More

```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/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.

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.

> setwd("D:)
+

> setwd("D:")
> getwd()
[1] "D:/"
> ls()
character(0)
>
> setwd("D:/week2-R")
> getwd()
[1] "D:/week2-R"
> ihsdata <- read.table("standardised_ihs.txt",header=T)
> head(data)

1 function (..., list = character(), package = NULL, lib.loc = NULL,
2     verbose = getOption("verbose"), envir = .GlobalEnv)
3 {
4     fileExt <- function(x) {
5         db <- grepl("\\\\.[^.]+\\\\.(gz|bz2|xz)$", x)
6         ans <- sub(".*\\\\.", "", x)
> head(ihsdata)
          chr       v2            v1   v3   v4   v5     v6     v7   v8    pos ref Totalcov annote subtelo uniq vars
1 Pf3D7_01_v3 61.88474 rs61.88473899 0.11 0.00 0.00 -0.879 838.66 0.01 107626   A    20943      H       0   54    0
2 Pf3D7_01_v3 62.50495 rs62.50495209 0.07 0.01 0.00  0.540 760.92 0.01 107756   T    21159      H       0   54    0
3 Pf3D7_01_v3 62.82442 rs62.82442283 0.13 0.00 0.00 -0.432 835.01 0.01 107823   T    18061      H       0   54    0
4 Pf3D7_01_v3 63.67790 rs63.67789662 0.28 0.00 0.01 -1.181 651.69 0.01 108002   G    17426      H       0   54    0
5 Pf3D7_01_v3 63.70649 rs63.70649478 0.10 0.00 0.01 -0.618 653.91 0.01 108008   C    20414      H       0   54    0
6 Pf3D7_01_v3 83.62511 rs83.62511438 0.15 0.01 0.01  0.570 592.37 0.01 112188   G    27950      D       0   54    0
           gene gene_old newpos colors mafs       means       sds     stand
1 PF3D7_0102400 PFA0120c 107626      1    A 0.000769774 0.6746478 1.3040430
2 PF3D7_0102400 PFA0120c 107756      1    T 0.236910604 0.9925150 0.3053751
3 PF3D7_0102400 PFA0120c 107823      1    T 0.000769774 0.6746478 0.6414751
4 PF3D7_0102400 PFA0120c 108002      1    G 0.074887805 0.7832379 1.6034564
5 PF3D7_0102400 PFA0120c 108008      1    C 0.236910604 0.9925150 0.8613579
6 PF3D7_0102500 PFA0125c 112188      1    G 0.000769774 0.6746478 0.8437443
> |
```

```
Data
ihsdata                 8533 obs. of 24 variables
```

```
D: week2-R
  Name                     Size     Modified
  ..
  standardised_ihs.txt     1.6 MB   Jan 16, 2014, 1:44 PM
```

When you open R Studio you will be presented with three windows, as shown above. The large window on the left hand side is the console window, where you input command and can view any data that has been created. The top right window is the workspace window, which lists both the recent commands used and all the data that has been loaded into R. The bottom right window contains tabs for files, plots, packages and help. Today we will mostly be using the plots tab. Throughout this introduction you will see text highlighted in the following ways:

Things for you to attempt in R

Commands to type into the console window

Variable, folder and file names

Getting help

If you continue to use R outside of this introduction you will find yourself running into problems on a regular basis. I say this with certainty because even experienced programmers run into problems and often spend more time debugging their code than they did writing it in the first place. When this happens there are a few important steps to solving the issue:

1. Check your code for mistakes such as spelling or putting the wrong number in. If R can run the command you typed it will, even if the result isn't what you wanted.
2. Keep a copy of your code in a separate file. This will help when it comes to finding errors but also makes it much easier to repeat an analysis. In the long run you can transform this into a pipeline, a set of instructions that can describes all the analysis in a project from start to finish.
3. Look at the output from each step and check it did what you intended to. Many mistakes are missed because R returned an output that looked correct at first glance even though it isn't.
4. Search online. If you run into a problem it's likely that somebody else has run into it beforehand, especially when using the most common functions. Learning how to search effectively will save you a lot of time in the long run.

Working directory & loading files

The working directory is the folder on your computer from which you are currently working and contains all the files needed for the current piece of work. For example if you are working on multiple projects you may have folders named *plasmodium, trypanosoma* and *influenza*. All of these folders may contain files with similar names, for example *snps.txt, genomesequence.txt* or *currentcode.r.*

In order for R to know which folder we wish to work from we need to define the current working directory. After doing this R will automatically look in this folder when loading files and save data / plots to this folder unless we tell it otherwise. To check the current working directory we use the command:

```
getwd()
```

which will provide us with a line that looks like this:

```
[1] "D:/folder/asecondfolder"
```

To set the working directory we use the command:

```
setwd("D:/folder/asecondfolder")
```

This will return an error as those folders don't exist. It's important to note that the location is surrounded by "quote marks" which tells R that the content is plain text and not a variable.

Try setting the working directory to *D:/R-intro*

Using R as a calculator

R can be used as a powerful calculator due to its ability to store and manipulate data and results. For example inputting the following:

```
((15 * 2.59^3)-5)/13
```

quickly returns 19.66228

Basic variables

A variable is a piece of data that has been assigned an name, which we can use to tell R which data to use when running scripts. The advantage here is that once data has been stored in a variable it can be recalled, copied and manipulated using its name rather than having to type it all in again. The simplest variables contain just 1 piece of data but more complex variables could contain an entire spreadsheet.

To define a variable we take the data and assign it a name in one of two ways (they do exactly the same thing):

```
a = 15

a <- 15
```

Here the variable is called ‘a’ and the data inside it is the number 15.

Any time you want to see what is in a variable you can just type its name into R. Variables can be used in place of the data they contain, for example if we replace the 15 in our earlier calculation with the variable:

```
((a*2.59^3)-5)/13
```

it will still return 19.66228. We can also use variables to store text, such as:

```
fish = “salmon”
```

Note the quote marks, which tells R we wish to store the text “salmon” in the variable fish.

```
fish = salmon
```

If you type the above what error do you get? What do you think this means?

Basic functions

Functions are predefined instructions that tell R to perform a specific operation. An individual function can perform multiple actions using a single command, saving a lot of time when it comes to writing scripts. It is possible to define your own functions in addition to the hundreds that are installed as part of R. A typical function takes the form of

```
functionname(variable, variable, etc)
```

You can often find out what a function does by typing

```
?functionname
```

While typing

```
functionname()
```

without any variables will show the individual actions that are performed when the function is run.

What does the class function do? Try using it with your a and fish variables

Expanding variables further

Most variables are more complex than the *a* and *fish* variables we defined above. Those were examples of scalar variables, which means they contains a single number or block of text.

Vectors are like a row of numbers or words which are collected together into a single place. One of the ways you can create them is by concatenating data together using the c function:

```
b = c(15,6,7)

c = c(fish,”crab”,”boat”)
```

which tells R to create a variables that groups the numbers or words together.

Does *c* contain what you expected it to?

What happens if you use *b* in our equation from earlier? What happens if you try and use *c* in the equation?

What is the difference between c and c()?

Variables can be expanded further by adding extra dimensions, generating tables such as those found in spreadsheets. We can do this using the matrix() command:

```
matrixone = matrix(data = c(b,4,5,6,15,2,”fish”,7,44,2), ncol=3)
```

What does matrixone look like? What do you think the ncol=3 is doing?

If you look closely at the above you can see that inside the matrix function we have also used the concatenate c() function to group our data together, including the variable *b* which we defined earlier. Learning to combine functions and variables within one another is a core skill in scripting. Don’t be afraid to try different combinations but always check the results to see if it did what you wanted it to do.

Dataframes are a special type of matrix where the columns are all assigned names in addition to the column number. This allows you to call or use a column without knowing the column number or order of all the columns. To create a dataframe use the following:

```
sampledf = data.frame(“numbers” = b, “ocean” = c, “morenumbers” = c(11,33,65))
```

In order to call one individual column you use the $ sign as so:

```
sampledf$ocean
```

which should return salmon, crab and boat. This only works with dataframes as the $ symbol is pulling up columns based off of their name.

Try creating a new dataframe called Continents that contains the columns Europe, Asia, Africa and NorthAmerica each containing 3 countries that can be found in these continents.

Why did we call the 4th column NorthAmerica and not North America?

Accessing data

Once we have created a variable we will often then need to view, use or alter it. Using the $ symbol in our dataframe is one way of doing this. For scalars (such as *a*) this means just using the ID we gave it but what if we wanted to change the "boat" in *c* into "ship"? Here we use square brackets [ ] to tell R the position of the item we wish to change. "boat" is the third entry of *c*, so to get just that entry we would write:

```
c[3]
```

while to change it to "ship" we would do:

```
c[3] = "ship"
or
c[3] <- "ship"
```

where we have multiple dimensions, such as in matrixone we need to specify the position of each dimension.

When we created matrixone we used the variable *c* to fill it with data. Has changing the contents of *c* changed *matrixone*?

For a table we do this in the order of row number then column number. For example:

```
matrixone[1,2]
```

would return the number 5. If you wished to get all the data in column 2 you would remove the row number as so but leave the , in place so that R knows the 2 is the column number:

```
matrixone[,2]
```

Try changing the "fish" in *matrixone* into the number 42.

Searching data

When dealing with large datasets it is often necessary to search through them to find only the small number of entries you are interested in. While it is possible to use the above approach of listing specific rows or columns this isn't feasible when you have thousands of possible entries to check.

To search through variables we need to tell R to match an entry in the data to the number or piece of text we are looking for. The simplest way of doing this is to use a double equals sign == which means we want R to check if what is on both sides is the same, in other words do they match. For example:

```
2 == 5

"fish" == "fish"

fish == "salmon"
```

would return FALSE, TRUE and TRUE respectively.

The last two examples above both return TRUE, what is the difference between them?

Using this we can search data to find cases that return TRUE. If we go back to *sampledf* we can try to search for the entries that contain the word crab using:

```
sampledf == "crab"
```

What does this return? What do these results correspond to?

What if we wanted to know all the data in the row associated with "crab"? (It helps here to imagine data being collected together as a spreadsheet with each row being a single entry). To answer this we need to specify that we're looking for the word "crab" in the column called "ocean" which we do using one of these two ways:

```
sampledf$ocean == "crab"
which(sampledf$ocean == "crab")
```

This will either return a FALSE TRUE FALSE or just the number 2, both of which tell us that "crab" is the second entry (which in this case is a row) in the "ocean" column. To get the entire row we combine this with the square brackets [ ] we used earlier:

```
sampledf[sampledf$ocean == "crab",]
sampledf[which(sampledf$ocean == "crab"),]
```

These should both give the same result, though for large datasets using the which() function is often quicker and more efficient.

Copying data

Often we will want to copy data from one variable to another. The most common reason for this is that we are interested in only a small part of the data and want to copy it to another variable where we can change it without changing the original. We do this in the same way as creating a new variable, by using the = sign. For example:

```
matrixtwo = matrixone
```

which will create an identical copy of matrixone called matrixtwo. This is particularly useful for creating a backup of your data in case you make a mistake later on. Alternatively we may want to copy just a portion of the data, which we can do as below:

```
oceandata = sampledf$ocean
```

More Functions – Try and put each of these to use

Functions are a way of automating a process in order to speed up what you are doing, we've already seen examples of this with c() and matrix(). Many additional functions exist and the best way to learn how they work is to try them and see what they do. Most functions have multiple options that you can include, you can find out what these are by putting a ? in front of the function name, eg ?plot

The following functions are some of the most commonly used and will be of use in the exercise at the end. It is also possible to write your own functions, allowing you to automate complex tasks that you perform on a regular basis.

read.table() – Reads a file that is organised as a table. Normally we will want to then store this in a variable using variableID <- read.table("*filename.txt*")

head() – Will return the first 6 row of a matrix or dataframe, useful for when there are hundreds or thousands of rows present. If you want more or less rows then use head(matrix, n=number)

tail() – Will return the last 6 row of a matrix or dataframe, useful for when there are hundreds or thousands of rows present.

dim() – Returns the dimensions of a matrix or dataframe, which will normally be the number of rows and columns.

nrow() – Returns the number of rows in a matrix or dataframe

ncol() – Returns the number of columns in a matrix or dataframe

colnames() – Returns the column names for a dataframe

length() – Returns the length of a vector or number of entries in a matrix / dataframe

max() – Returns the largest value from a vector of numbers

min() – Returns the smallest value from a vector of numbers

sum() – Adds all the numbers in a vector together

mean() – Calculates the mean of the numbers in a vector

median() – Calculates the median value of the numbers in a vector

table() – Summarises a vector as a table listing all the unique entries and the frequency with which they appear

plot() – Attempts to plot the data in a manner that R thinks is correct. This is a powerful function with many inbuilt options

barplot() – Attempts to plot a barplot of the data

hist() – Attempts to plot a histogram of the data

abline(h=X) or abline(v=X) – Add a horizontal or vertical line to a plot at position X

After this little intro (thanks to Craig Duffy), we have now three exercises.

1. Learn a bit about different plots. We should have done these during the presentation, so now you have time to try it on your own.
2. Work with SNP data from a publication – page 23.
3. Do a Tajima's D something that population genetics love to do. We don't expect you do to it, but if you are VERY fast, something to not get bored.

IMPORTANT:
All the relevant data can be found at

1. On the HPC

/export/projects/bioinfo3/to16r/BioinfoWorkshop/Data/Module_R/
You can copy the files over.

2. Webpage (just assessable within the university (use VPN if needed))

https://tinyurl.com/3Ibioinfo
The link on the Tuesday afternoon, Rstuff.zip or directly:
https://tinyurl.com/3Ibioinfo/Rstuff.zip

3. On the ftp site:

ftp://ftp.sanger.ac.uk/pub/project/pathogens/tdo/Exercise/Rstuff.zip

Enjoy, and ask questions when stuck!

PS: Please excuse any typos!

R tutorial - Graphics

January 5, 2018

1 Examples from reads counts

This short tutorial will recap what was done in the demonstration.

As dataset we are going to work with the first single cell data of Plasmodium falciparum (https://www.biorxiv.org/content/early/2017/02/10/105015). Plasmodium cells (or parasites as unicellular) are from asexual and gametocyte parasites (supplemental table 6). The columns represent the cells and the rows the genes. The number in each cell are the readcounts.

IMPORTANT: We could have used any type of RNA-Seq data. The main aim of this exercise is for you to explore some visualization options of R and understand the underlying mechanimns.

First let's load the readcounts file and make a little heatmap.

1.1 Loading the data

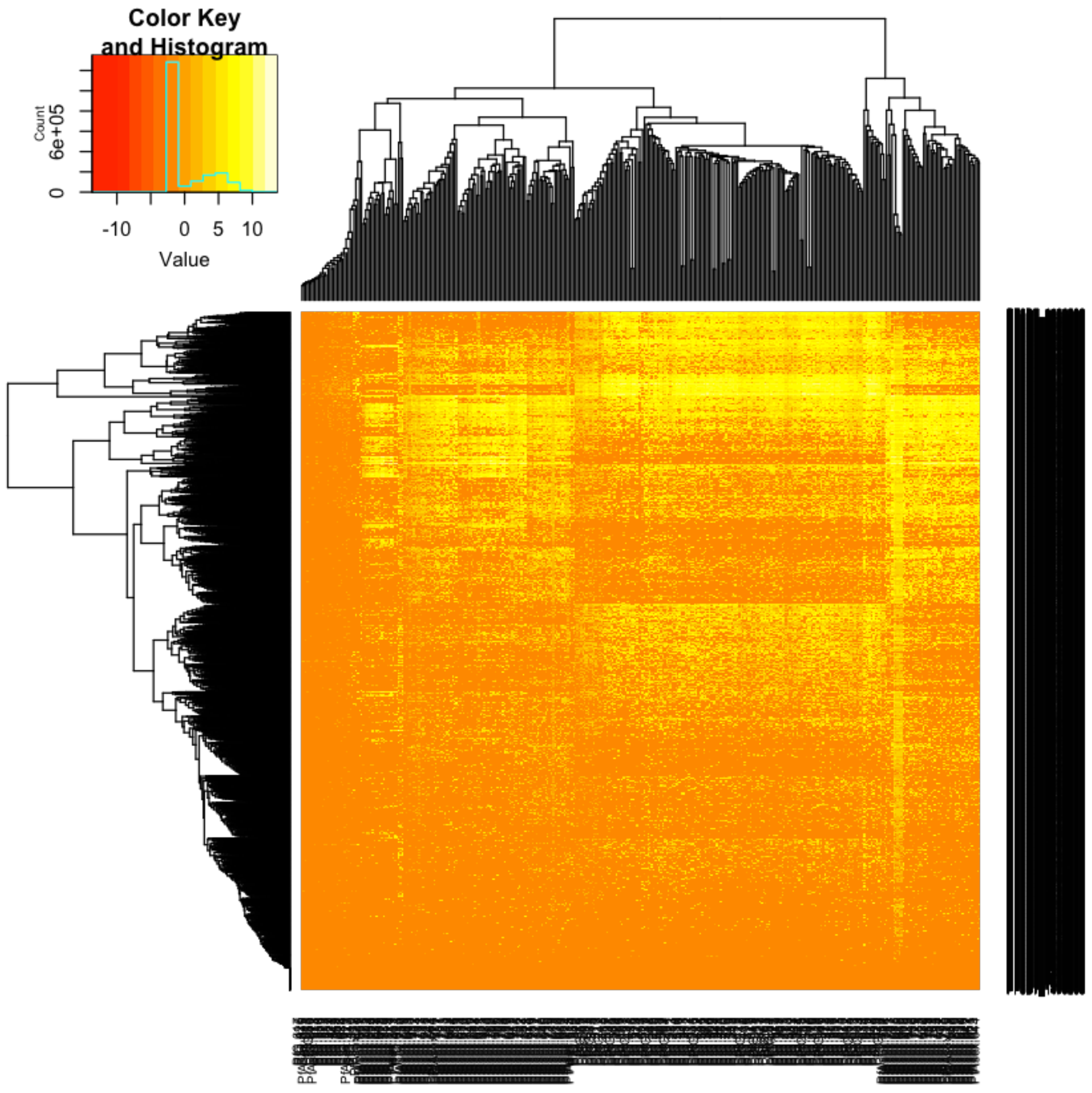
```
In [8]: d<-read.table("readcounts.txt", header=TRUE, row.names=1);
        head(d)
```

	PfG.43	PfG.44	PfG.45	PfG.46	PfG.47	PfG.48	PfG.49	PfG.53	PfG.54	PfG.55
PF3D7_1478800	0	0	0	0	0	0	0	0	0	0
PF3D7_1478600	0	0	0	0	0	0	0	0	0	315
PF3D7_1478100	0	0	0	303	56	0	0	1	0	0
PF3D7_1478000	0	0	0	0	0	0	0	0	0	0
PF3D7_1477800	0	0	0	0	0	0	0	0	0	0
PF3D7_1477700	0	0	0	0	0	0	0	0	0	0

This looks very easy, but from experience, to get the data in the correct format in the first place can be very challenging. We are going to practice this on Thursday afternoon.

1.2 heatmaps and PCA

```
In [9]: library(gplots)
        heatmap.2(log(0.1+as.matrix(d)),trace="none")
```



You can see several pattern between the cells, where genes seem to have different reads counts. Any idea why we used a log transformation?

heatmap.2 is VERY powerfull. You can see the options with

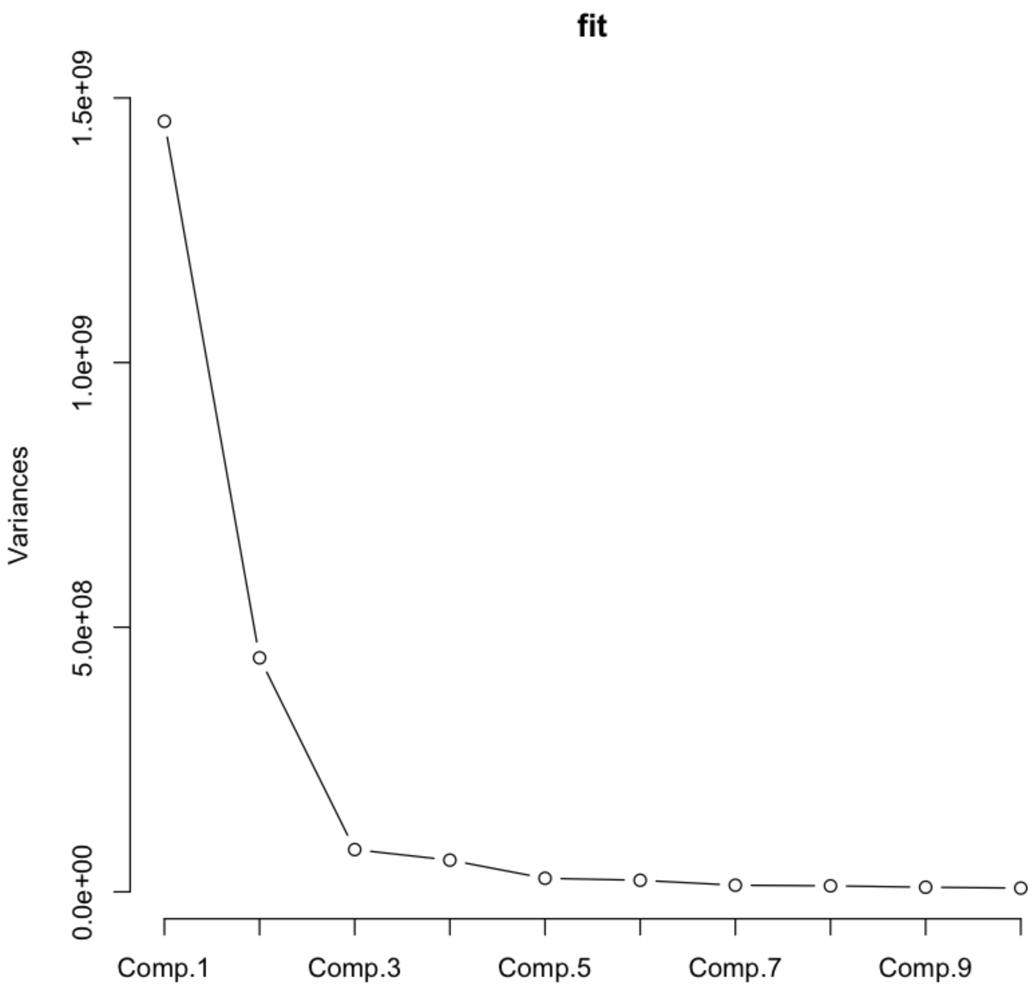
```
In [10]: ?heatmap.2
```

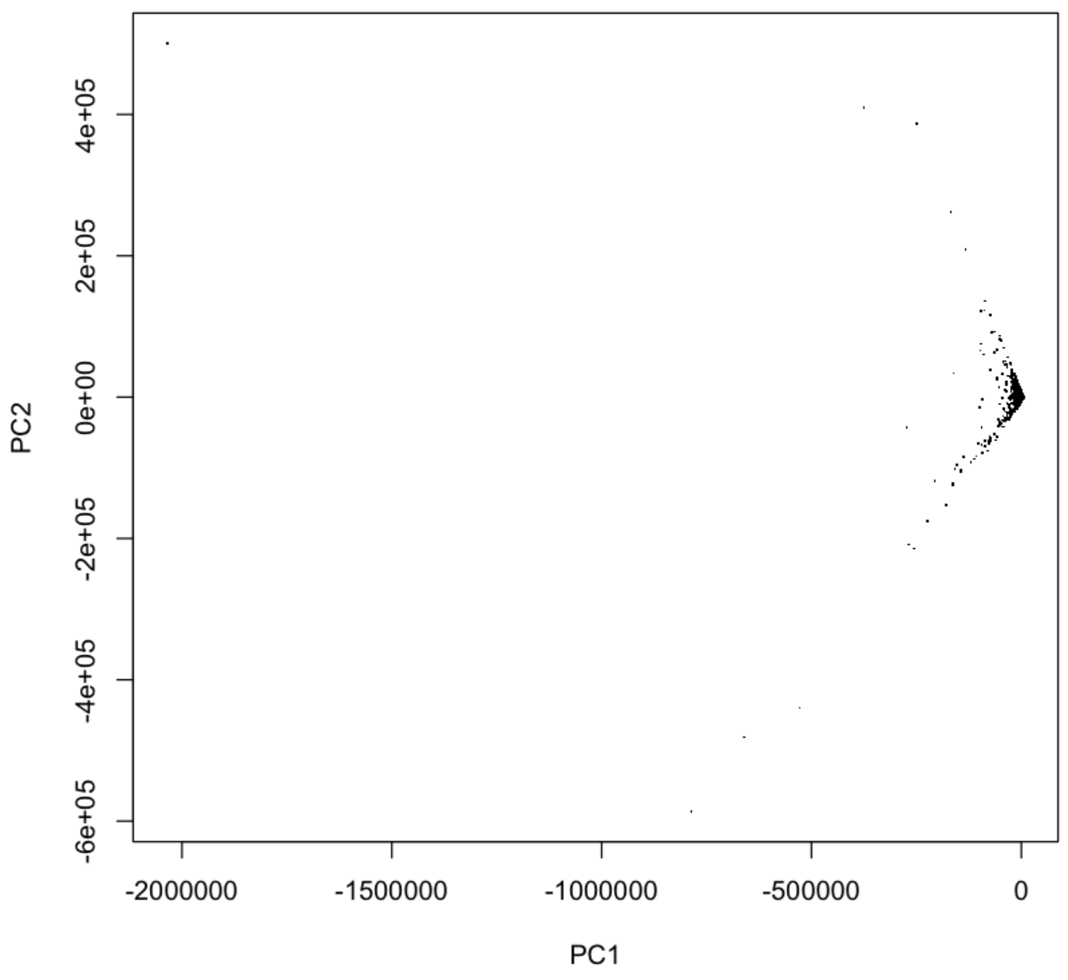
In the heatmap you can see several cells and genes without any readcounts. How can we get rid of them?

```
In [24]: keep <- rowSums(d) >= 1000 ### keep row/genes with 1000 reads mapped over 400 cells
         dnew <-d[keep,]
         keep <- colSums(d) >= 50000 # keep columns/cells with at least 50k mapped reads (5500
         dnew <- dnew[,keep]
         dim(d) # first number is the dimension of the orignal files
         dim(dnew) # the dimension of the reduced matrix
```

1. 5266 2. 365
1. 4625 2. 325
Try to do the heatmap again. Did it change anything?
What about doing a PCA (principal component analysis)?

```
In [48]: fit <- princomp(as.matrix(dnew), cor=F)
         plot(fit,type="lines") # scree plot
         gsa.pred <- predict(fit)
         plot(gsa.pred[, 1:2],  xlab = "PC1", ylab="PC2",pch=".")
```





Does it make a lot of sense? NO! we are looking at genes, rather then the cells here. But with just some lines you can generate a PCA. Tomorrow afternoon we will come back to this!

1.3 Correlation and QC

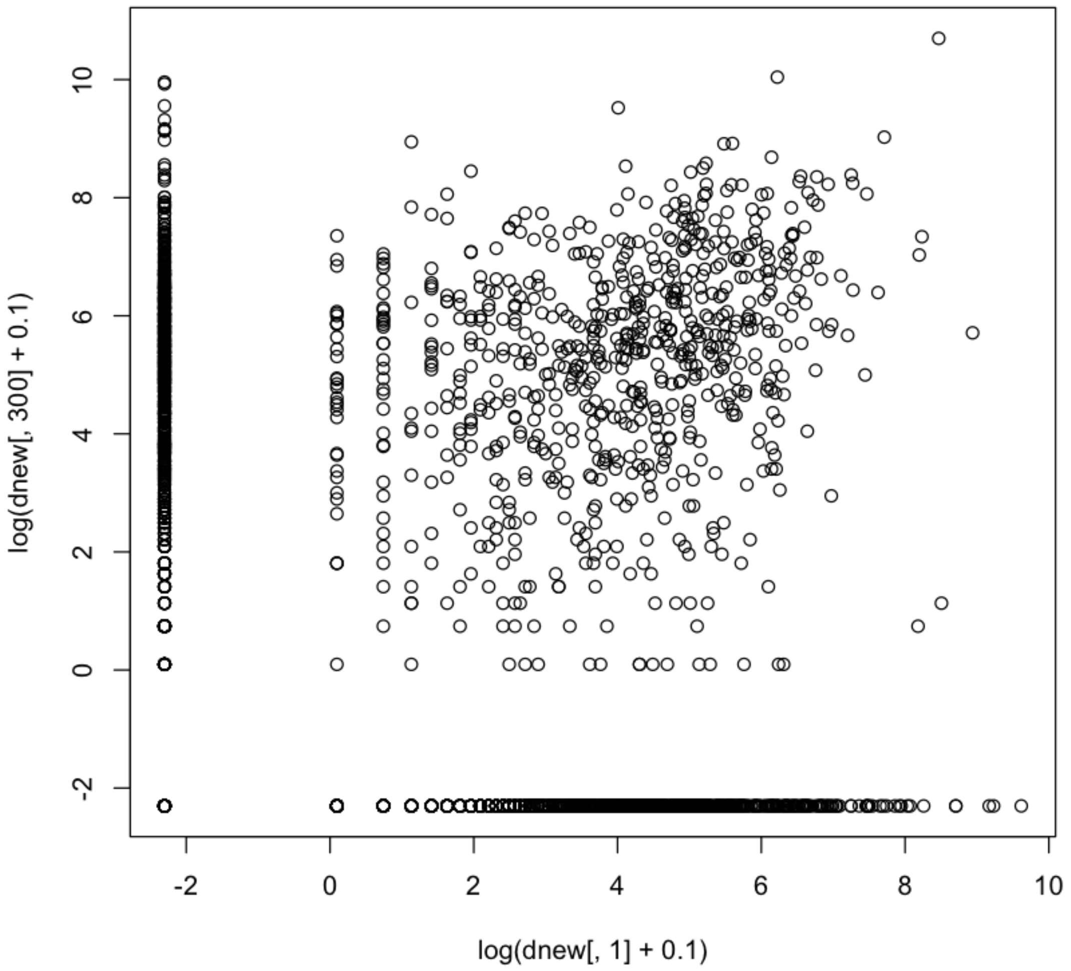
Interestingly, we went directly into the heatmap and the tough stuff!

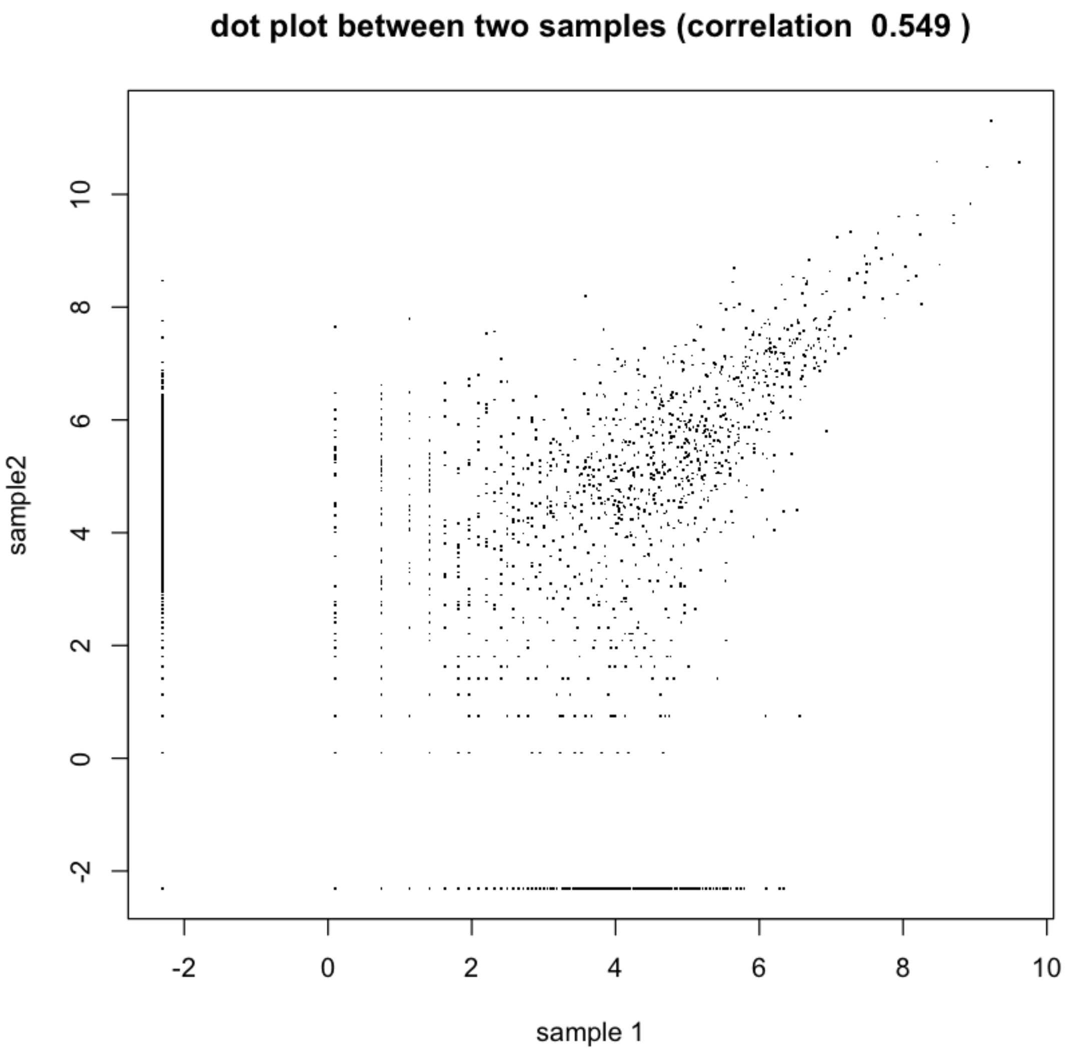
Sorry, first we should have done some QC. We could count how many genes have not reads mapped, the higher read count, and look for correlation between the samples. Let's start with the correlation:

```
In [33]: cor(log(dnew[,1]+0.1),log(dnew[,300]+0.1))
         cor(log(dnew[,1]+0.1),log(dnew[,2]+0.1))
         plot(log(dnew[,1]+0.1),log(dnew[,300]+0.1))
         c<-cor(log(dnew[,1]+0.1),log(dnew[,2]+0.1))
```

```
c<-round(c,digits=3)
plot(log(dnew[,1]+0.1),log(dnew[,2]+0.1),
     main=paste("dot plot between two samples (correlation ",c,")"),
     xlab="sample 1", ylab="sample2",pch=".")
```

```
0.19289411349222
0.549042856275539
```





Yes, a lot of things are going on here. First we are looking at the correlation between two samples. Next, we ploted the two columns. The second plots looks nices, as it has label and a title. We also include the correlation into the title…

But how do you know which samples 1, 2 or 300 are?

```
In [35]: head(d[,c(1,2,300)],1)
```

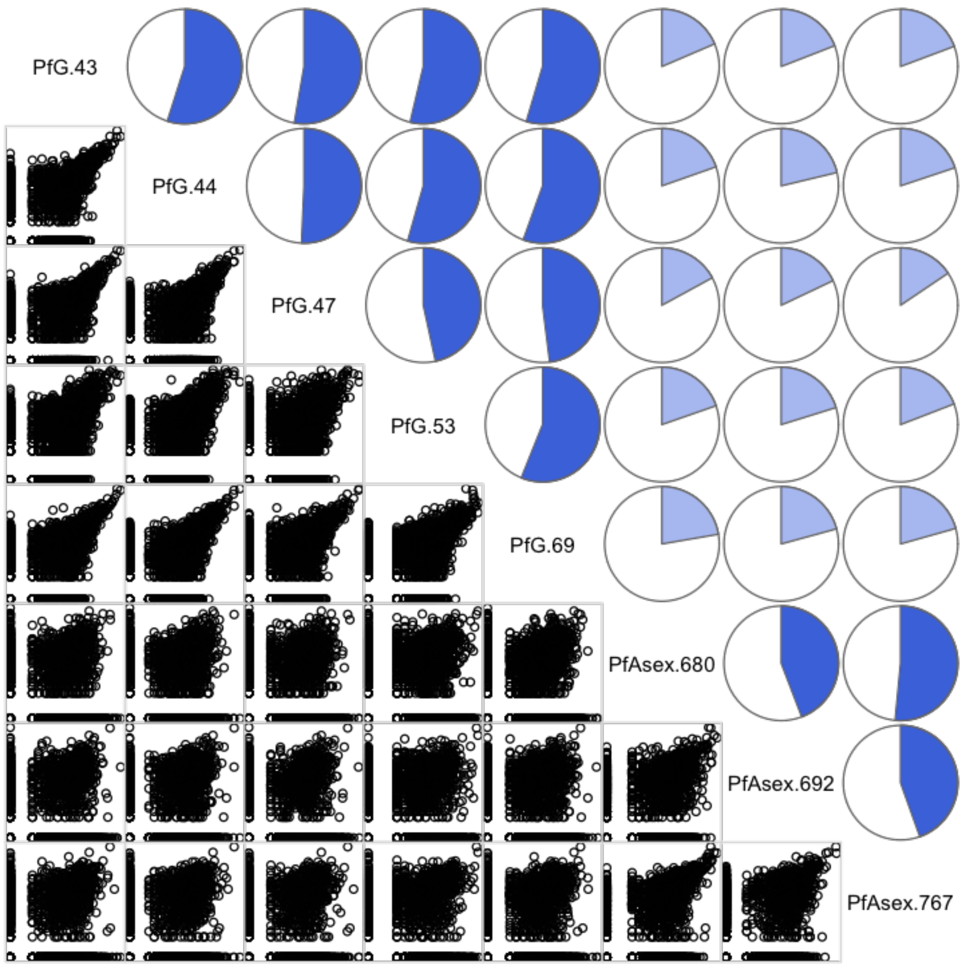
	PfG.43	PfG.44	PfAsex.721
PF3D7_1478800	0	0	0

This is not very intuitive and it would be great to have the header (PfG.43 etc) in a variable, but this is why I hate R from time to time… it is not so easy. We would need to read the table again from the file… or do you know a better way?

But let's look at more correlations:

```
In [39]: install.packages("corrgram")
         library(corrgram)  # you might need to install it with install.packages("corrgram")in
         corrgram(log(dnew[,c(1,2,5,8,20,240,250,300)]+0.1),lower.panel=panel.pts, upper.panel=
```

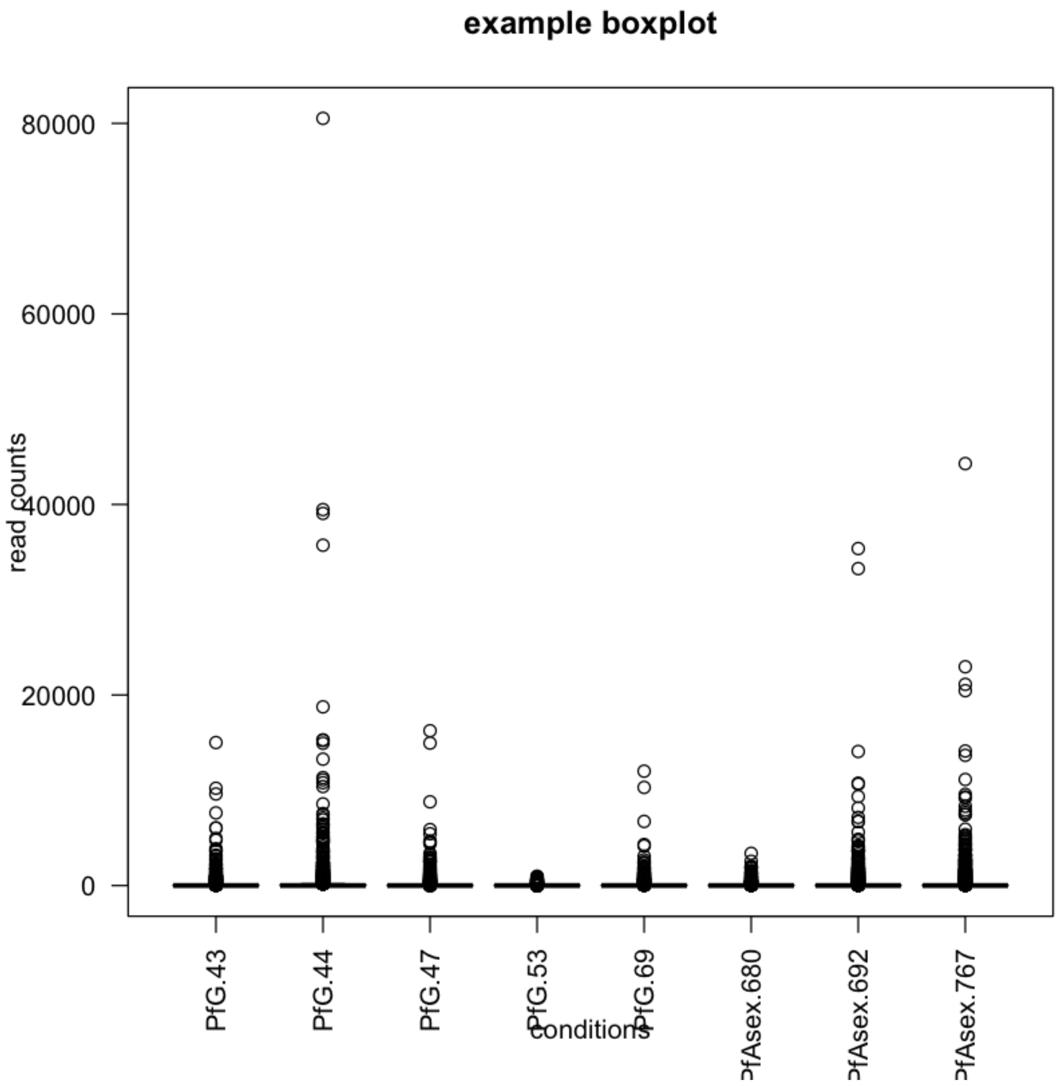
```
The downloaded binary packages are in
        /var/folders/rl/snk6w64922336q3fzybd35ym0000gn/T//RtmpMs2Ikd/downloaded_packages
```

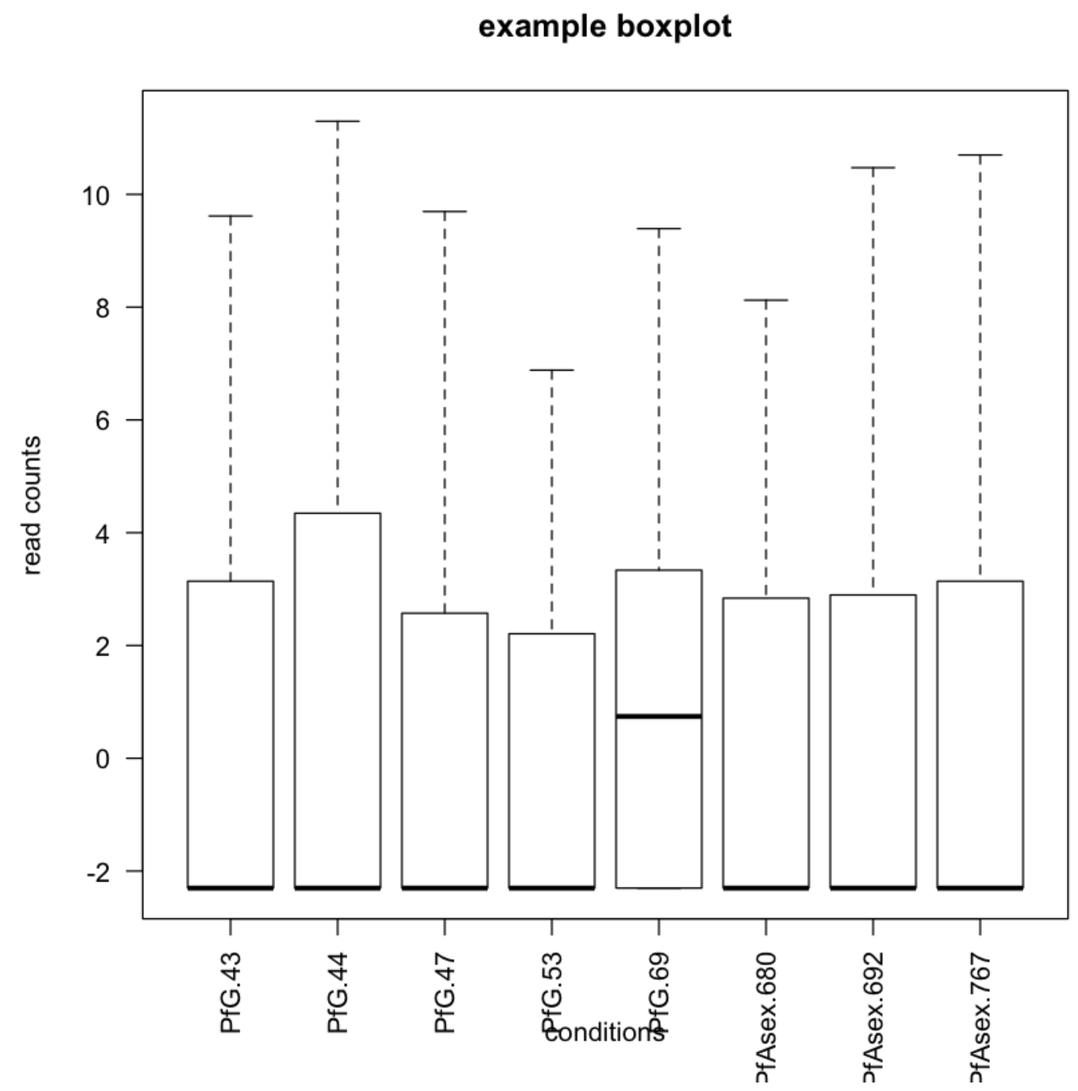


So the correlation between the different life stages is higher… makes sense. But overall the correlation does not seem perfect, at least compared to normal RNA-Seq. Reasons are likely to be the single cell methods -, but do you like the plot? Google R corrgram to see other examples.

Let’s know look at other plots:

```
In [44]: d2<-dnew[,c(1,2,5,8,20,240,250,300)]
         boxplot(d2,las=2,main="example boxplot",ylab="read counts",xlab="conditions");
         boxplot(log(d2+0.1),las=2,main="example boxplot",ylab="read counts (log)",xlab="condi
```





Those plots give us an idea that most of the genes do not have expression - or it was not captured by the single cell method (SMART-Seq2). Never the less, it will be published in a good journal

1.4 ggplot2 - high quality graphics?

A nice library to do plots is ggplot2. It is a bit painful to run, but the plots are really for publication (at least for my publications). So once you have the data in the correct format, things are easy.

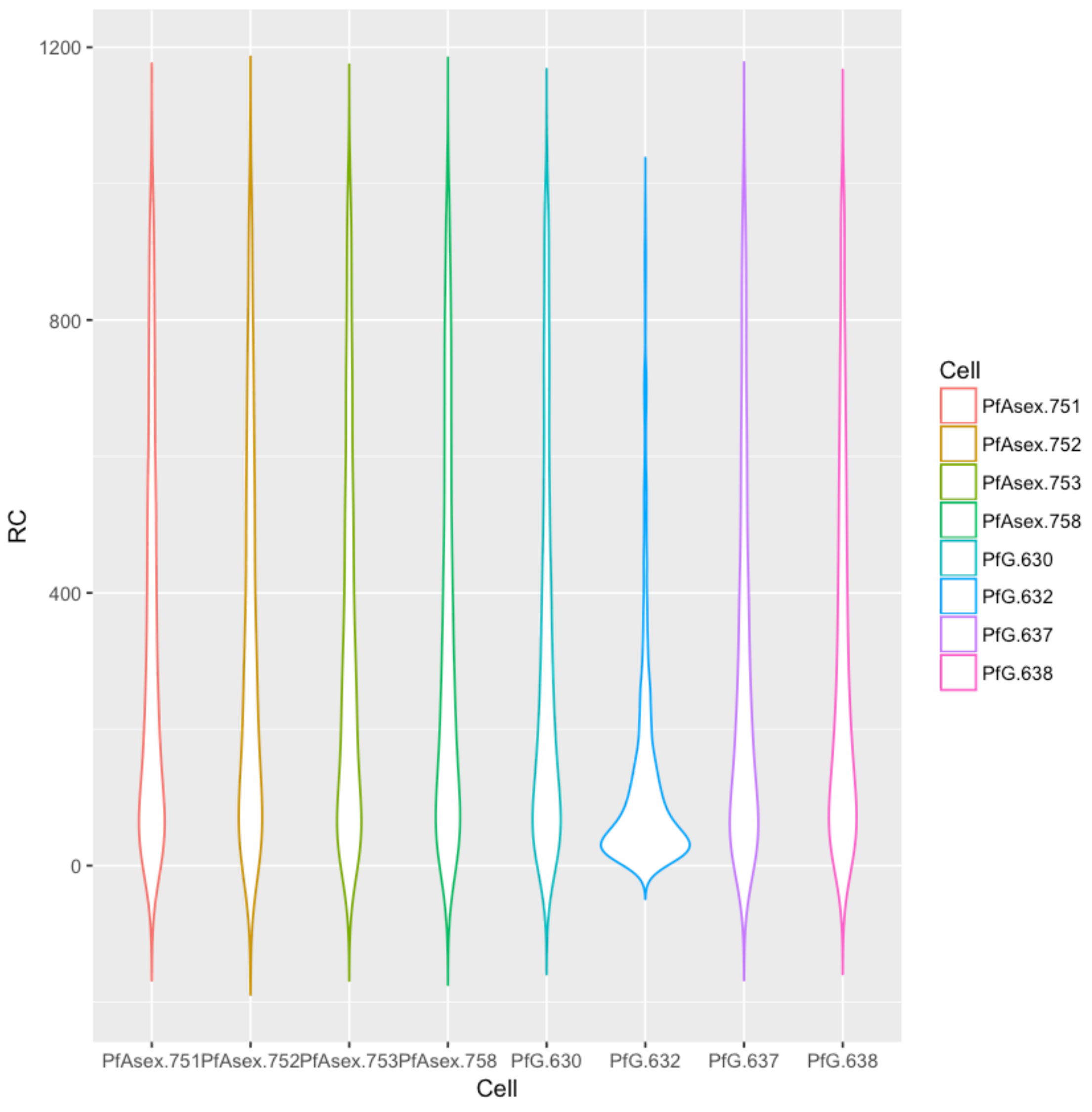
```
In [52]: install.packages("ggplot2")
         library(ggplot2) # this installs and loads the packages
```

```
The downloaded binary packages are in
```

```
/var/folders/rl/snk6w64922336q3fzybd35ym0000gn/T//RtmpMs2Ikd/downloaded_packages
```

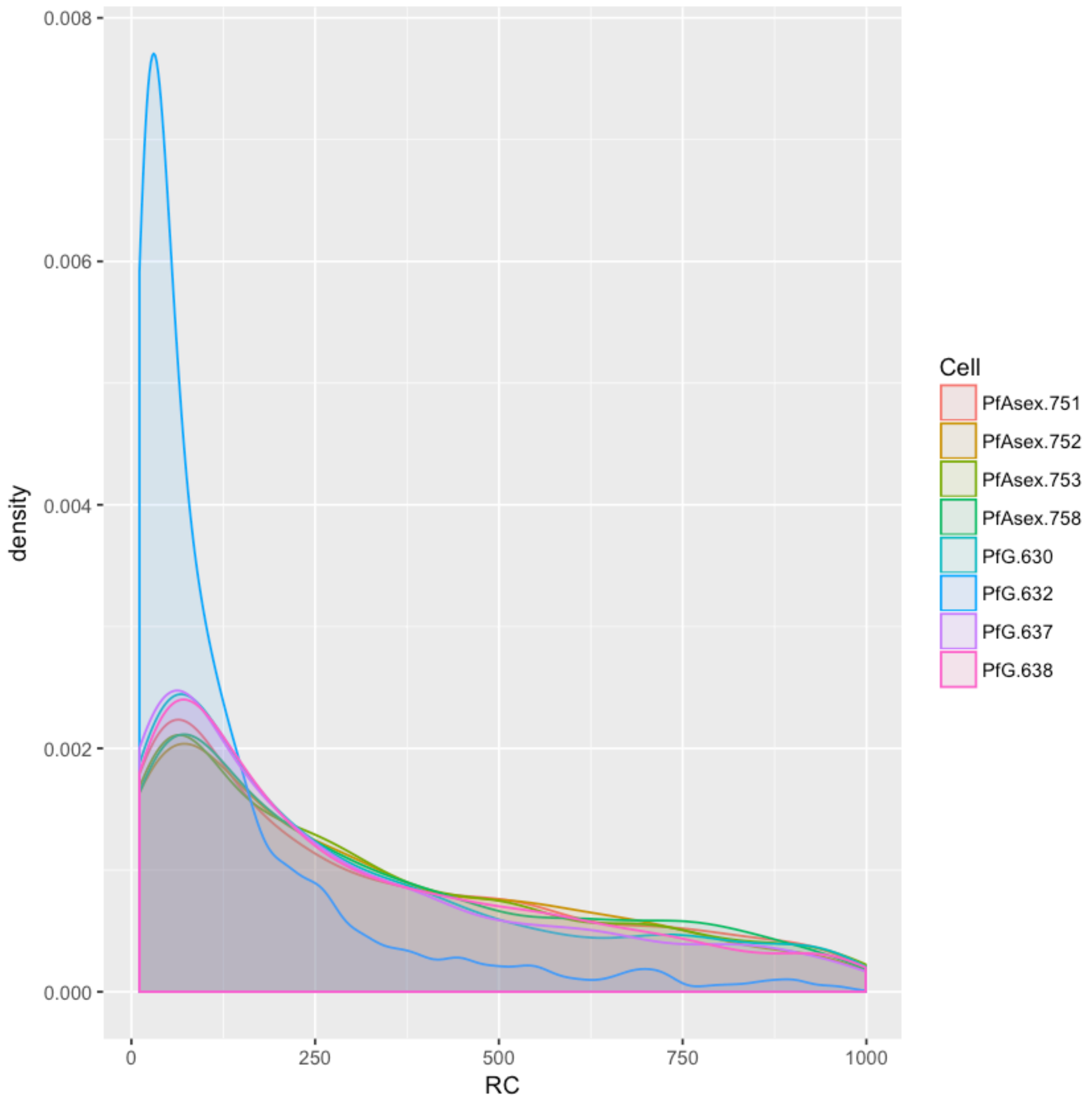
The file format is basicaly a list in Linux: >head GG.RC.txt Cell Gene RC PfG.637 PF3D7_1478100 181 PfAsex.758 PF3D7_1478100 105 PfAsex.751 PF3D7_1476800 798 PfAsex.753 PF3D7_1476800 982 PfAsex.758 PF3D7_1476800 203

```
In [77]: dat<-read.table("GG.RC.txt", header=TRUE);
         p <- ggplot(dat, aes(x=Cell, y=RC, color=Cell)) + geom_violin(trim=FALSE)
         p
```

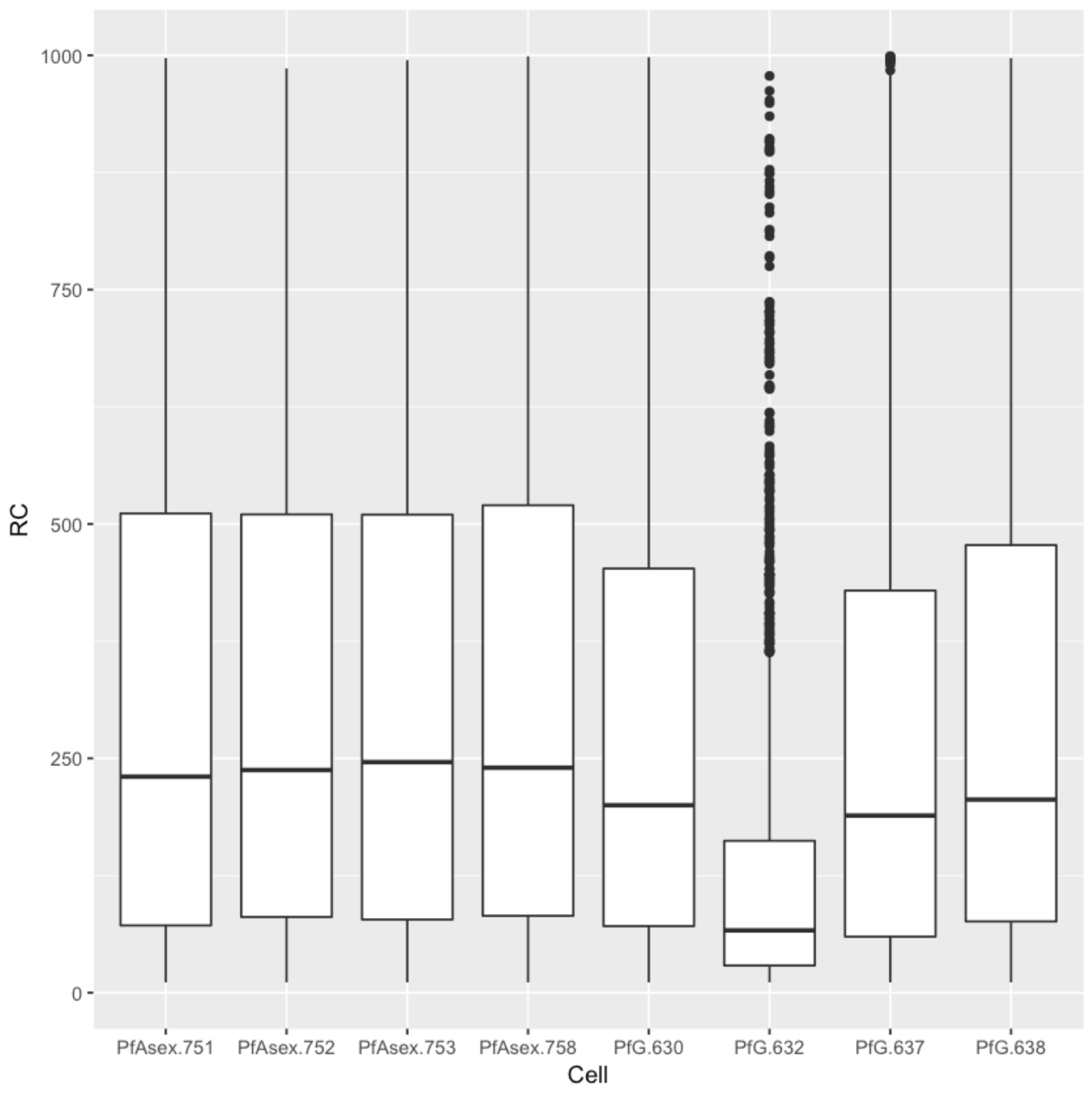


The dataset has just values between 10 and 1000 read counts if not the plot would look weird. Another plot

```
In [78]: p<- ggplot(dat, aes(RC, fill = Cell, colour = Cell))
         p + geom_density(alpha = 0.1)
```



```
In [80]: p <- ggplot(dat, aes(x = Cell, y = RC))
         p + geom_boxplot()
```



I would agree with you, that the data might not be the best to obtain gobsmacking graphics. But I hope you see that R is like a lego game, where you have to put the right briggs together… don't worry if you don't understand each command 100% you are not alone. And I also find myself googeling often for the correct command and syntax.

Exercise

The following exercise is designed to give you an opportunity to learn how to use R with a dataset from a study into genome wide selection in malaria (http://mbe.oxfordjournals.org/content/early/2014/04/08/molbev.msu106). The exercise will require you to put the prior examples to use, changing the input as required. For the most part we have avoided providing the actual commands you will need to type as working out how to structure the input is the best approach to learning how to use R. All of the tasks in this exercise can be completed using the functions and information you have already been provided with so try experimenting with them to work out what they do to the data.

Set the working directory to *D:/R-intro/*

Load the provided data in this folder into a variable called *ihsdata* using

```
ihsdata <- read.table(“standardised_ihs.txt”, header=T)
```

here the header=T tells R to treat the first line as column names and load the data as a dataframe.

Have a look at the dataframe to get an idea of how it is organised.

Each row in *ihsdata* represents a single SNP in the genome of the human malaria parasite *P. falciparum* while the columns contain information about that SNP. The columns are:

1. chr – which chromosome the SNP is on
2. pos – the position of the SNP on the chromosome
3. ref – the base present in the genome reference strain at this position
4. Totalcov – the total sequencing coverage at this position from the 100 isolates this data was generated from
5. gene – the ID of the gene that this SNP is positioned within (if within a gene)
6. gene_old – the previous version of the gene ID (useful if you are looking up older papers)
7. genpos – the position of the SNP in the genome
8. colors – a column we’ll use to colour the chromosome later
9. ihs – the ihs score for this SNP, which is a measure of direction selectional at this locus

How many SNPs are there in the dataset?

How many chromosomes are there? How many SNPs are on chromosome 5?

What is the mean, max and min of the coverage? Why do you think none have a coverage of 0?

Using the hist() function try and plot a histogram of the total coverage for the dataset

This dataset was used to identify regions of the genome under directional selection by calculating their integrated haplotype score (ihs) for each SNP and plotting these scores genome wide. To plot the scores across the genome, with the chromosomes coloured in an alternating red / black pattern use the following:

```
plot(ihsdata$genpos, ihsdata$ihs, pch='.', cex=2, col=c("black","red")[ihsdata$colors])
```

which can be broken down as follows:

plot – The plot function which is used to plot basic figures in R

ihsdata$genpos – the position of each point of the x-axis, defined by the genpos column of *ihsdata*

ihsdata$ihs – the position of each point on the y-axis, defined by the ihs column of *ihsdata*

pch='.' – tells R to use . for drawing the points

cex=2 – tells R how big to make each point

col=c("black","red")[ihsdata$colors] – tells R to use the colours black and red determined by the values in the ihsdata$colors

Now that you have the plot you should be able to see that there are clusters of SNPs with high scores.

How many SNPs in *ihsdata* have ihs scores > 3?

Only 4 SNPs have ihs scores > 7. Which genes are they in?

The chloroquine resistance transporter (gene ID PF3D7_0709000) is the main gene responsible for chloroquine resistance in malaria and is located on chromosome 7.

How many SNPs are present within this gene?

Can you add a line to the plot to mark the position of this gene?

Finally can you plot only the SNPs that are on chromosome 7?

Computer Practical – 2017

Use of R for the calculation and analysis of Tajima's D

Purpose and Scope

Using the DNAsp program you have already investigated sequence polymorphism in six *P. falciparum* genes using data from a Kenyan population. There are, however, over 5000 genes spread across the entire genome and analysing each of these using DNAsp would be a laborious process. In this exercise we will therefore utilise genome wide SNP data from a 2008 Gambian dataset (Amambua-Ngwa et al. 2012 PLOS Genetics). This data was aligned to the *P. falciparum* 3D7 reference genome using the same principles you learnt during week 1 of the module with SNPs subsequently called and filtered from the BAM files which were generated.

Loading the data

After opening R studio the first thing you need to do is to tell it which folder you will be working from by setting the working directory. For this exercise you'll be working from the folder called *R-intro* which is located on the D drive. To set the working directory use the following command:

```
setwd("D:/R-intro")
```

You can check which directory you are in by typing the following command into the console:

```
getwd()
```

We're also going to check the required files are present using:

```
list.files()
```

This should list files called *tjddata.txt*, *tjdfunctions.r* and *standardised_ihs.txt*.

The tjdfunctions.r file contains the custom functions we are going to be using in this exercise, if you wish to see how custom functions are written you can open this file in notepad or wordpad to see some examples. To use these functions we need to tell R to load these into memory, making them available for later. This is done with the command:

```
source("tjdfunctions.r")
```

As we have already set the working directory R will automatically look there for the tjdfunctions.r file, if it was in a different folder we would need to tell R that. After loading the file you should see a list of functions appear in the workspace window in the top right of R studio.

In addition to the functions we also need to load the data from *tjddata.txt* into a variable called *snpdata* using the command:

```
snpdata <- read.table(“tjddata.txt”, header=T)
```

Have a look at the *snpdata* variable using some of the commands you learnt in the introduction to R.

Each row in *snpdata* represents a single SNP and the first 12 columns represent annotation information for that SNP. Can you work out what each of these columns represent? What do columns 13 onwards contain?

How many SNPs are present in this dataset? How many samples are there?

As you have seen when using a function we typically type the command functionname(argument) where the argument may be a variable, such as snpdata or a file such as data.txt (not all functions need an argument however).

Using head(snpdata) you should have been able to view the first few rows of snpdata, which is a type of variable we encountered in the introduction to R called a data.frame. It is organised in a manner very similar to an excel spreadsheet with rows and columns. Remember that we can navigate through a subset of the data using [‘row number’ , ‘column number’] or [‘row name’ , ‘column name’], for example:

```
snpdata[30,9]

snpdata[30,”gene”]
```

will both bring up the data from row 30, column 9 as column 9 has been named “gene”

If we want to get a range of rows or columns we can use the a colon (:) to specify a range of rows or columns, so

```
snpdata[5:20,]
```

would return rows 5 through to 20 (inclusive) of snpdata.

What are the limitations of retrieving data in this way?

Using this method try and find the position of the gene PF3D7_0800600

Can you find the gene using one of the approaches from the introduction to R?

Manipulating the data

Just as we can view data by using square brackets [ ] we can copy the results to a new variable, which we're going to do by creating two new variables, the first of which is called datalegend:

```
datalegend <- snpdata[,1:12]
```

As we haven't specified any rows before the comma datalegend now contains every row from columns 1 to 12 of snpdata.

Create a second variable called *datagenotypes* and then copy columns 13 to 64 of *snpdata* into it.

Check it by comparing the contents of *datagenotypes* to *snpdata*.

How many columns are there in *datagenotypes*? What is the first column name of *datagenotypes*?

Before we can calculate our Tajima's D scores we need to convert the data into a more usable format, which we do using the following two commands:

```
genos <- converts(datagenotypes, as.character(datalegend[,3]))

genos <- apply(genos, 2, as.numeric)
```

Examine the *genos* variable and compare it to the *datagenotypes* and *datalegend* variables, can you work out what these two commands are doing?

As we need to calculate Tajima's D for each individual gene we also need to extract a list of the gene names using the following two commands:

```
genenames <- names(table(datalegend[,"gene"]))

genenames <- genenames[genenames != "-"]
```

How many genes are listed in the genenames variable?

You can use the length() functions to check this.

Why might there be less genes listed than the 5772 that are present in the reference sequence?

Calculating Tajima’s D

We’re now ready to calculate Tajima’s D for our dataset, before we do that have a closer look at the tajmad1 function we’re going to be using by clicking on it in the workspace window. This should bring up multiple lines of code, which would be a lot to type out each time we wished to calculate Tajima’s D. By placing this code into a function we are able to run it simply by entering tajimad1(arguments) into the console.

To calculate Tajima’s D you need to enter the following two lines of code:

```
tajimascores <- NULL

for (i in genenames){tajimascores <- rbind(tajimascores, tajimad1(genos,
datalegend, i, 3, 1))}
```

which will calculate the Tajima's D scores for each gene and store them in the variable called tajimascores.

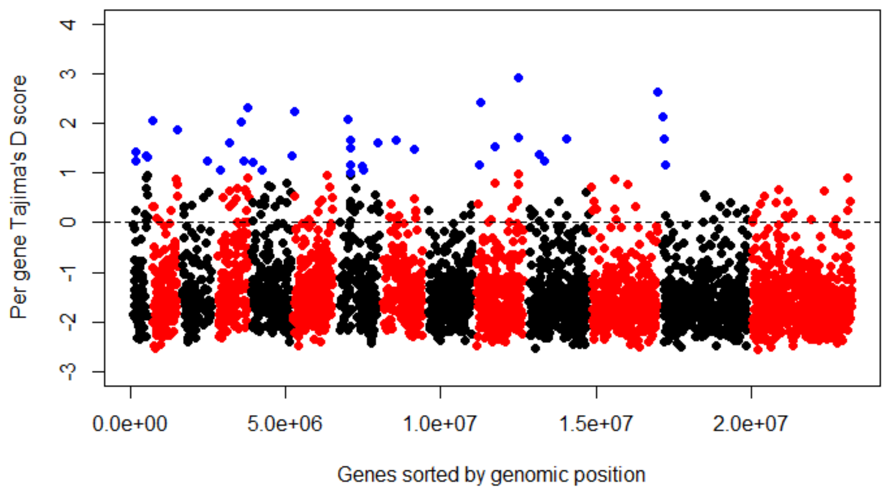
How many genes was Tajima’s D calculated for?

Plotting the data

One of the biggest strengths of R is its ability to plot data, which we’re going to do using a custom version of the plot function called *plottd*:

```
plottd(tajimascores, 1)
```

This will plot the Tajima’s D score for each gene on the Y axis while the X axis indicates the position of the gene in the genome. For this visualisation we have coloured each chromosome in alternating colours (black and red) while genes with a score of above 1 are coloured in blue. It should look like this:



We now know that the majority of genes have a negative Tajima's D score and only a small number have a score above 1. The negative scores across much of the genome is due to the presence of an excess of rare alleles compared to that expected under a neutral model of evolution. In malaria this was caused by a historical population expansion, with the rare SNPs having entered the population subsequently.

In order to find out which genes have scores above 1 we can use the *which()* function, demonstrated below. Here we are using it to say "which rows in the tajimasd column of tajimascores are greater than or equal to 1" then using that to copy the data into a new variable, called *highscores* (For the purpose of this module you don't need to understand exactly how which works, just that it is possible to select data in this manner).

```
highscores <- tajimascores[which(tajimascores[,"tajimasd"] >= 1),c(1:3,6)]
```

How many genes have a Tajima's D score of >= 1?

What is the gene with the highest scoring Tajima's D score? What is its function? (the website www.plasmodb.org will be useful here).

Genes with high Tajima's D scores are predicted to be under balancing selection, indicating that there is an excess number of alleles with intermediate frequencies at these loci.

What sort of genes in malaria might be subject to this type of selection?

What processes might drive balancing selection? Do you think the top scoring gene fits this model?