Introduction to Sequence Informatics

Institute of Infection and Global Health

Introduction to sequence informatics

Session 3 - Introduction to R

R is a powerful and free scripting environment designed for the statistical analysis and visualisation of data. In many cases it can be considered to be a more advanced alternative to Excel, however, it is a far more powerful program that is capable of performing complex, multistep calculations with extremely large datasets. R is especially well suited to the one off, manual exploration of data that is common in academic research. You can easily load in a dataset, filter down to a subset of samples, calculate a summary statistic and then plot the observed variation.

These capabilities come at the cost of accessibility, R has a reputation for being complicated and hard to learn. This is in part due to its use of a text interface similar to the command line, its idiosyncratic approach to scripting and the sheer number of options available to the user. This session will cover only the very basics of R, should you wish to delve further into it the Advanced Statistics for Epidemiology module run by Dr Rob Christley covers the program in more detail.

Log on to the server as normal and copy the folder called session3 to your home directory. The session3 folder can be found in in /archived/Informatics. Once you have copied it over enter the folder and examine the files that are present.

You can load R by typing the following into the terminal:

R

From now on all of the commands you type into this window will be run in R, which has its own set of functions and options. The start of the line in R is denoted by

>

If you press enter and see

+

Then it means that R thinks you haven't finished your command. This may be because you are within the middle of a function or it may mean that you have missed something, such as a closing quote mark around text.

If you wish to run a program in the terminal you will need to either quit R using

q()

Getting help

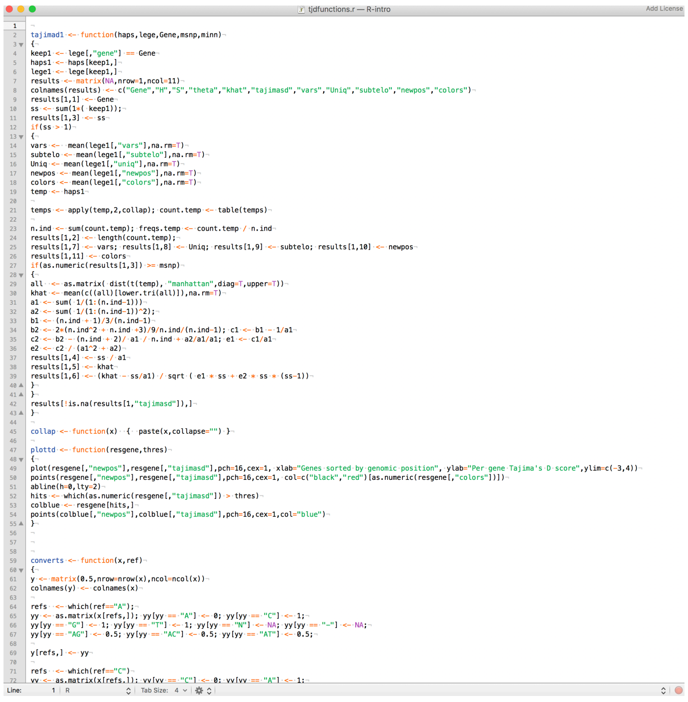
If you continue to use R outside of this introduction you will find yourself running into problems on a regular basis. I say this with certainty because even experienced programmers run into problems and often spend more time debugging their code than they did writing it in the first place. When this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

1. Check the inbuilt help for functions by typing **help(*functionname*)** or by placing a **?** in front of the function name. This help will provide a list of the options for the function and often includes examples of how to use the function.

2. Check your code for mistakes such as spelling errors 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.

3. Keep a copy of your code in a separate file. This will help avoid errors as it prevents you from having to type the same code over and over again. 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. Be careful when using this approach that you update or duplicate your code if you change something, using the wrong variable or code version is a common mistake.

   A number of script editors (Notepad++, TextMate, Atom, Vim) are available which will add markup to your files and highlight individual elements in a variety of scripting languages, making it easier to spot errors and trace workflows.



```r

tajimad1 <- function(haps,lege,Gene,msnp,minn)
{
keep1 <- lege[,"gene"] == Gene
haps1 <- haps[keep1,]
lege1 <- lege[keep1,]
results <- matrix(NA,nrow=1,ncol=11)
colnames(results) <- c("Gene","H","S","theta","khat","tajimasd","vars","Uniq","subtelo","newpos","colors")
results[1,1] <- Gene
ss <- sum(1*( keep1));
results[1,3] <- ss
if(ss > 1)
{
vars <-  mean(lege1[,"vars"],na.rm=T)
subtelo <- mean(lege1[,"subtelo"],na.rm=T)
Uniq <- mean(lege1[,"uniq"],na.rm=T)
newpos <- mean(lege1[,"newpos"],na.rm=T)
colors <- mean(lege1[,"colors"],na.rm=T)
temp <- haps1

temps <- apply(temp,2,collap); count.temp <- table(temps)

n.ind <- sum(count.temp); freqs.temp <- count.temp / n.ind
results[1,2] <- length(count.temp);
results[1,7] <- vars; results[1,8] <- Uniq; results[1,9] <- subtelo; results[1,10] <- newpos
results[1,11] <- colors
if(as.numeric(results[1,3]) >= msnp)
{
all  <- as.matrix( dist(t(temp), "manhattan",diag=T,upper=T))
khat <- mean(c((all)[lower.tri(all)]),na.rm=T)
a1 <- sum( 1/(1:(n.ind-1)))
a2 <- sum( 1/(1:(n.ind-1))^2);
b1 <- (n.ind + 1)/3/(n.ind-1)
b2 <- 2*(n.ind^2 + n.ind +3)/9/n.ind/(n.ind-1); c1 <- b1 - 1/a1
c2 <- b2 - (n.ind + 2)/ a1 / n.ind + a2/a1/a1; e1 <- c1/a1
e2 <- c2 / (a1^2 + a2)
results[1,4] <- ss / a1
results[1,5] <- khat
results[1,6] <- (khat - ss/a1) / sqrt ( e1 * ss + e2 * ss * (ss-1))
}
}
results[!is.na(results[1,"tajimasd"]),]
}

collap <- function(x)  {  paste(x,collapse="") }

plottd <- function(resgene,thres)
{
plot(resgene[,"newpos"],resgene[,"tajimasd"],pch=16,cex=1, xlab="Genes sorted by genomic position", ylab="Per gene Tajima's D score",ylim=c(-3,4))
points(resgene[,"newpos"],resgene[,"tajimasd"],pch=16,cex=1, col=c("black","red")[as.numeric(resgene[,"colors"])])
abline(h=0,lty=2)
hits <- which(as.numeric(resgene[,"tajimasd"]) > thres)
colblue <- resgene[hits,]
points(colblue[,"newpos"],colblue[,"tajimasd"],pch=16,cex=1,col="blue")
}



converts <- function(x,ref)
{
y <- matrix(0.5,nrow=nrow(x),ncol=ncol(x))
colnames(y) <- colnames(x)

refs  <- which(ref=="A");
yy <- as.matrix(x[refs,]); yy[yy == "A"] <- 0; yy[yy == "C"] <- 1;
yy[yy == "G"] <- 1; yy[yy == "T"] <- 1; yy[yy == "N"] <- NA; yy[yy == "-"] <- NA;
yy[yy == "AG"] <- 0.5; yy[yy == "AC"] <- 0.5; yy[yy == "AT"] <- 0.5;

y[refs,] <- yy

refs  <- which(ref=="C")
yy <- as.matrix(x[refs,]); yy[yy == "C"] <- 0; yy[yy == "A"] <- 1;
```

4. Look at the output from each step and check it did what you intended it to. Many mistakes are missed because R returned an output that looked correct at first glance, even though it wasn't.

5. 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.

UNIVERSITY OF LIVERPOOL | INSTITUTE OF INFECTION AND GLOBAL HEALTH

The working directory

The working directory is your current folder and should contain 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, genome.fasta* 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. By default R sets the working directory as whichever folder you opened it from.

To check the current working directory we use the command:

```
getwd()
```

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

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

To set the working directory we use the command:

```
setwd("/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.

We can use the commands

```
list.files()
```

or

```
dir()
```

to show the contents of the current folder (equivalent to running the ls command in the terminal).

If we wished to list the contents within a different folder then we must specify the path to it relative to our current location. For example:

```
dir("..")
```

would list the contents of the folder above our current location while

```
dir("asecondfolder")
```

lists the contents of the folder called asecondfolder located within our current folder.

If we wish to search for files by name we use can specify a pattern to match and R will list any files or folders that contain that pattern

dir(pattern="fish")

which will list anything that contains fish in the file name such as fish.txt, fishes.jpg, onetwothreefish.doc

You should already be working from the folder session3. Using the above commands check that this is correct and list the files that are present within the directory.

Having run those commands you will have hopefully noticed that they all include two important characters - ( ). These brackets indicate that you are running a function - a small piece of code that performs a specific task. With many functions these brackets allow you to specify options. For example:

read.csv("my_data.csv")

Would tell R to load the csv file called "my_data.csv" into memory (but will give an error if you try it because that file does not actually exist).

You are already familiar with using ls on the command line. In R there is a similar function that lists the variables you have loaded into the program and it is also called ls.

Run ls with and without the brackets - why do you think they give two very different results?

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

returns 19.66228. The big difference between R and a simple calculator is that we're not limited just to numbers and that we can store data using variables.

Basic variables

A variable (sometimes referred to as an object) is a piece of data that has been assigned a name and is saved within the memory of the computer. If you have ever done trigonometry (think y = mx+c) then you have used a variable. The difference between doing this on paper and in a computer is that we can save the contents of those variables. 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 thousands of entries nested within one another.

To define a variable we take the data and assign it a name in one of two ways:

a = 15

a <- 15

Here the variable is called 'a' and the data inside it is the number 15. The name of a variable should always start with a letter. 'a' and 'a1' are fine as variable names but '1a' is not. Similarly special characters ($, %, !, [ ] etc) are not allowed to be used anywhere within a variable name as they have special functions.

Any time you want to see what is in a variable you can just type its name into R and it will return the contents. 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 plain text as a string, 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?

More types of variables

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

The next type of variable is a Vector, which groups a set of data together in a set order. One of the ways you can create a vector is by combining data together using the c (combine) function:

b = c(15,6,7)

c = c(fish,"crab","boat")

which tells R to create variables that combine the numbers or words into a vector or list.

Does *c* contain what you expected it to?

What happens if you use *b* in our earlier calculation?

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 called matrices. The easiest way to think of these is that they are equivalent to spreadsheets and are comprised of data organised by rows and columns. 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 combine 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.

Accessing subsets of data

Once we have created a variable we will often then need to view, use or alter it. For simple variables containing only one piece of data (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"

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

Where we have multiple dimensions, such as in matrixone we need to specify both the row number and the column number, separated by a comma. 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.

How would you change all of the data in row 2 using a single command?

If we wanted to bring up data within a range then we need to use the : command, so 2:5 indicates we want entries 2 to 5 inclusive (so 2, 3, 4, 5).

Display columns 1 and 2 of sampledf using the : option

Finally, if we want to get a non-continuous range of data we use the c() to provide a vector of positions. For example:

matrixone[,c(1,3)]

Would display columns 1 and 3 but not column 2.

Factors, lists and dataframes

The other types of variable we will be using are factors and lists.

Factors are special types of vectors where the data is organised into categories called ‘levels.’ Once it has been created a factor may only contain entires that match the specified levels.

Create a new factor called d that is a copy of our vector c using the command:

d = factor(c)

Look at them both? Are there any obvious differences? Lets try to change the data within them. Type the following commands (which we will cover in the next section) and see what happens to c and d

c[1] = “whale”

d[1] = “whale’

Because c is a vector we can easily assign the new entry to it but for d there is no level equal to “whale” and R returns a warning message. Why is this useful?

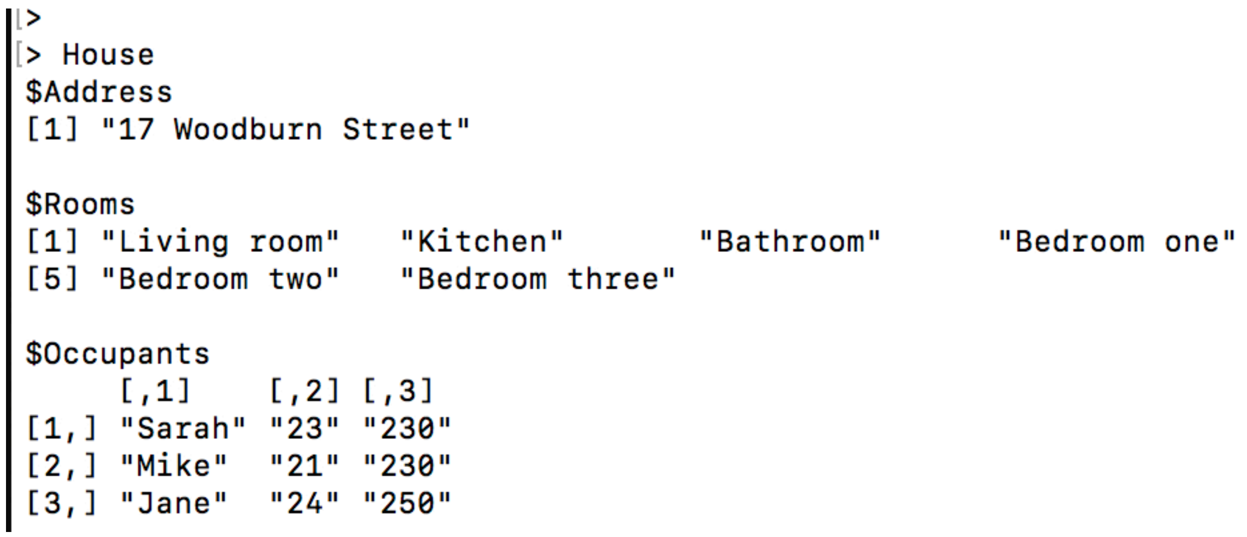
Firstly it makes it harder to make mistakes when entering data, if you have a vector listing the colour of objects and tried to add a “ywllow” entry R will check against the existing levels, not find it and know to return an error.

Secondly it ensures that data is stored in discrete categories which is important for many types of statistical analyses.

Finally it is computationally efficient when using large datasets. When R stores data in a factor the actual data is only stored once (the ‘level’) alongside a list of all the positions where that data should occur. Rather than having to store the same piece of data hundreds or even thousands of times the program simply looks up its location as needed.

The easiest way to remember how factors work is to think of them as like an index in a book. The level is the entry, next to which is the list of all the pages that it can be found on.

The final type of object we will be using is a list. Lists are special in that they may contain data in a combination of formats all at once. For example a list for a house may contain a vector with the address, a second vector with the rooms and a matrix with the details of the occupants. This can all be stored within the single list variable called 'House'



```
>
> House
$Address
[1] "17 Woodburn Street"

$Rooms
[1] "Living room"   "Kitchen"       "Bathroom"      "Bedroom one"
[5] "Bedroom two"   "Bedroom three"

$Occupants
     [,1]    [,2] [,3]
[1,] "Sarah" "23" "230"
[2,] "Mike"  "21" "230"
[3,] "Jane"  "24" "250"
```

Today we will only be using a special type of list called a dataframe. Each element of a dataframe must be a vector or factor and they must all be of equal length, unlike in our above example.

This allows the data to be presented in a 2D format like in a matrix. Each column of dataframe must have a single type of data (numeric, character) but different columns can be assigned different types. Each column may also be associated with a unique name. 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 whale, crab and boat. This works as the $ symbol is calling columns based off of their name rather than position.

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?

Basic functions

Functions are a set of 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.

For example

dim(matrixone)

Returns the dimensions (number of rows and columns) of matrixone

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

Finding subsets of 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.

One way to search through variables is 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 check the contents of an entire row or column of data? (It helps here to imagine data being collected together as a spreadsheet with each row being a single entry). To answer this we might 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 could combine this with the square brackets [ ] we used earlier:

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

It is important that you understand how these two commands work, if you are unsure ask for one of the instructors to go over it with you.

These two commands give the same result but they work in slightly different ways. This is a common feature of programming, there is rarely just one way to do something. Find the approach that works best for you.

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 affecting the original. We do this in the same way as creating a new variable, by using the = sign or using a <- arrow. 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
```

Which will copy the ocean column into a new vector called oceandata. It's important to note that because you are copying only a single column from sampledf into oceandata it no longer has rows or columns, just positions.

Basic Functions Continued

You have already come across a number of functions such as c(), which() or matrix() but these are only the tip of the iceberg as R includes thousands of functions 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 or require that you run them on data that is in a specific format. Most of the time you can find out what these requirements are by putting a ? in front of the function name, eg

?plot

The functions on the next page are some of the most common and will be of use in the exercise at the end. It is also possible to write your own functions or collections of functions to automate complex tasks and analysis. You can publish these functions as a package on the website CRAN and there are currently over 15,000 freely available packages, each of which will provide you with new options for analysing your data.

Common Functions

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

Exercise 1

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.

Check you are in the working directory *session2*

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 individual 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

Using what you have learned and the common functions list try and answer the following questions. You will find that there are multiple ways to answer each of these problems, once you have done so consult with those around you to see if they used a different approach.

1. How many SNPs are there in the dataset?
2. How many chromosomes are there? How many SNPs are on chromosome 5?
3. What is the mean, max and min of the coverage?
4. Why do you think none of the SNPs a reported coverage of 0?
5. Using the hist() function try and plot a histogram of the total coverage for the dataset
6. Copy the SNPs from chromosome 3 into a new dataframe and plot the position of each SNP against its iHS score - you will need to supply these values to the plot function as x-positions and y-positions.

The dataset you are using comes from a real study and was used to identify regions of the genome under directional selection by calculating a statistic called the 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=19, cex=1, 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=19 – tells R to use circles for drawing the points

cex=1 – 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 column

Repeat the plot but this time plot the iHS scores on the x-axis and the total coverage on the y-axis with the dots coloured in two different colours of your choice.

Change the plot back to its original version without retyping the command

Now that you have the original plot back you should be able to see that there are clusters of SNPs with high scores. Go back to the data and try to work out the following

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?

Exercise 2

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

During exercise 1 we plotted the ihs selection statistic from data which had already been analysed. In this data we will start at the beginning, with raw SNP data from a 2008 Gambian dataset (Amambua-Ngwa et al. 2012 PLOS Genetics). This data was previously aligned to the *P. falciparum* 3D7 malaria reference genome before SNPs were extracted for analysis.

Loading the data

Check the required files are present using:

```
list.files()
```

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

The tjdfunctions.r file contains a set of custom functions that 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 less (though you will need to quit R to do so). 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.

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

Use the read.table command to load the data into a variable called snpdata. You will need to set the header option to TRUE.

Have a look at the *snpdata* variable using some of the commands you have already learned.

What are the column names of *snpdata?*

Each row in *snpdata* represents a single SNP and the first 12 columns represent annotation information for that SNP. Can you guess what each of these columns might 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 of the dataframe type of variable we encountered earlier. 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”

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 earlier?

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?

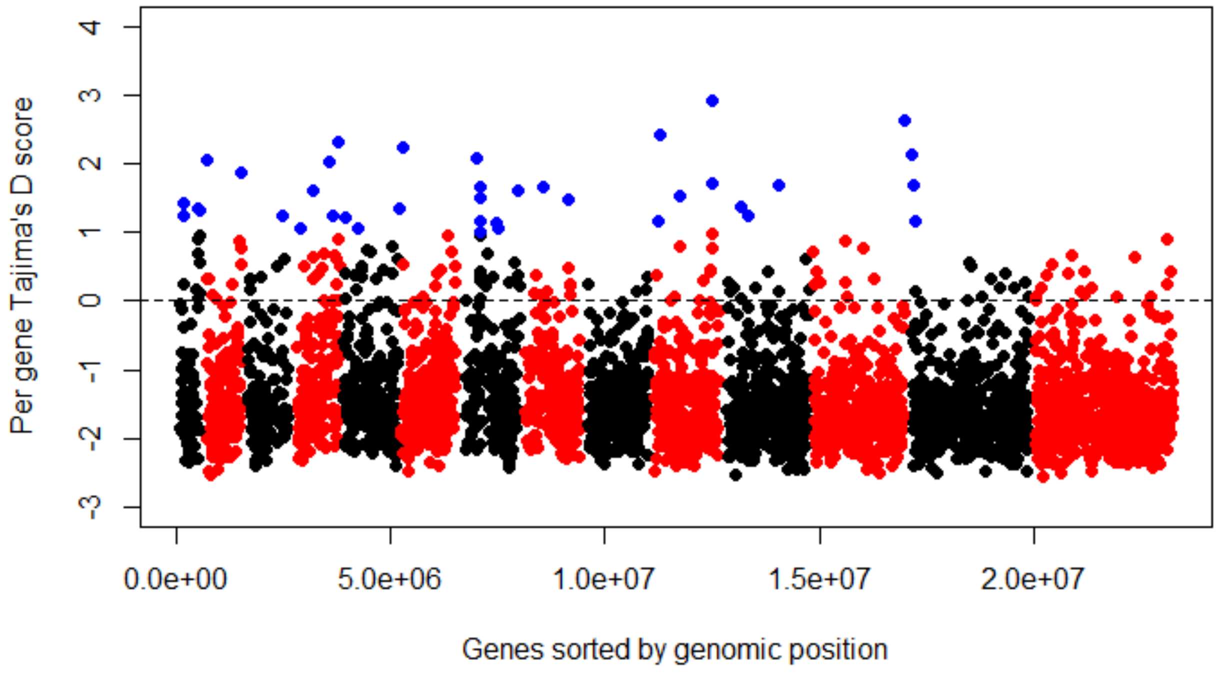
Plotting the data

One of the biggest strengths of R is its ability to plot data, which we're going to do using a custom 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 can see 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.



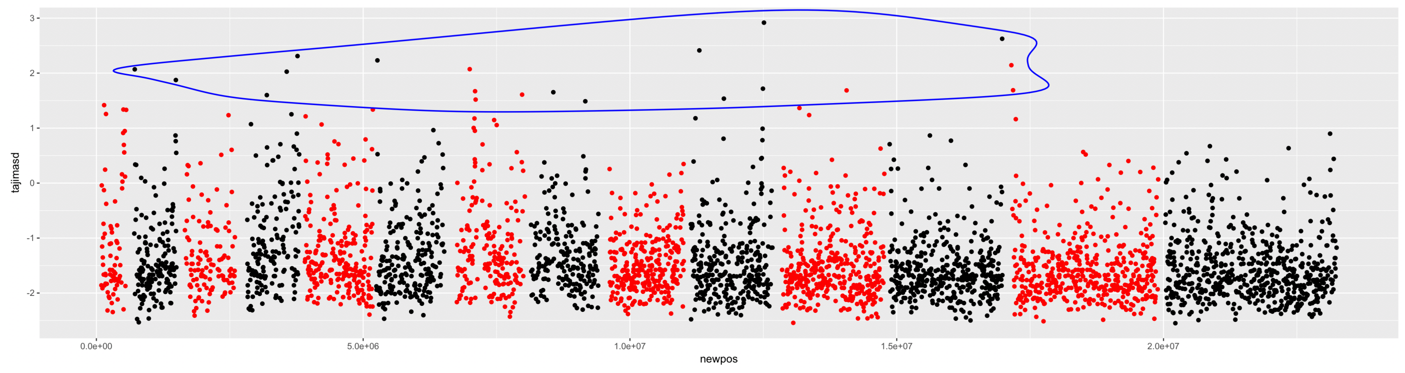
In order to find out which genes have scores above 1 we can use the *which()* function, demonstrated below. Here we are using it to say "which rows in the tajimasd column of tajimascores are greater than or equal to 1" then using that to copy the data into a new variable, called *highscores* (For the 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? Note that all of the entries of tajimascores and highscores have "" around them. This indicates that they are being stored as plain text. When we created the highscores variable R looked at the data and could tell that it was actually a number, thus allowing us to find values > 1. R will always try and do this sometimes it will get it wrong. In these situations you may receive an error but often R will output results as best it can and without proper checks you may not notice the problem until much later on.

We've already mentioned that there are often multiple ways to do the same thing. What if we had wanted to plot the data but highlight all the genes with a Tajima's D > 1.5 by circling them?



One of the possible ways of doing that involves using a plotting package called ggplot2. As we have already discussed there are thousands of packages are available for R from a website called CRAN and you can install them using the command:

install.packages("package name")

We have already installed the packages you need but you need to load them before you can use them. To do this we use the library function:

library(ggplot2)

library(ggalt)

Ggplot is a very popular package that vastly improves upon the plotting functions that come with R. It has its own specific set of commands for plotting that are beyond the scope of this introduction but to produce the above image we would type:

plot2<-ggplot(tmp, aes(newpos, tajimasd, col=as.factor(colors)))+geom_point()

plot2<- plot2+scale_color_manual(values=c("red","black"),guide=FALSE)

plot2<-plot2+geom_encircle(aes(newpos, tajimasd), data=aboveone, color="blue", size=2, expand=0.01)

plot2