Intro to R and testing genetic interactions with ANOVA

R Clay Wright

5/23/2017

Table of Contents

# Using this document

The goal of this particular tutorial is to teach you how to efficiently and effectively use RStudio to teach yourself how to do anything you may want to do with R. If you would like a more in-depth R tutorial check out [Code School](http://tryr.codeschool.com/). If you’re new to R check out these nifty [cheatsheets](https://www.google.com/webhp?sourceid=chrome-instant&rlz=1C5CHFA_enUS686US686&ion=1&espv=2&ie=UTF-8#q=r+cheat+sheet). These are great references to download and have available when coding.

Keep in mind while navigating this tutorial that you are supposed to be experimenting with R, making mistakes, causing errors and figuring out why. Nothing you do in RStudio will break your computer. Coding is a trial-and-error process, especially as you are learning. Try anything! See what happens! Learn from everything!

##### ???Question???

Throughout the tutorial you will find several sections with this label followed by questions for you to answer. The answers to these questions or the methods by which you can find the answers have already been covered in the above sections. Replace the *answer here* text with your answer along with any code you may have used to arrive at that answer and or a plot you have produced.

The text within this tutorial is formatted to differentiate code from dialog. text that looks like this is R code or keyboard commands. You can copy and paste this code into an R console and execute it. Within multiline blocks of code functions will be colored blue, comments will be colored green, strings will be teal and Booleans (True or False values) will be in red. If you don’t know what any of these terms mean yet, don’t worry, we’ll get to them.

In some cases the output from the code will be included in blocks of code. This output will be prefixed with #>.

Let’s get started!

# The R programming language

R is a programming language for statistical computing and graphics generation. R is based on an older statistical computing language called S. It is freely available and open source. There is a huge community of people writing and sharing their R code. R can be downloaded for your personal computer [here](http://cloud.r-project.org).

R is a line-oriented language, meaning each line of code is run or interpreted independently (if you are familiar with C/C++ you will know each bit of code must be followed by a semicolon).

### Why use R, when I already know Excel?

In addition to being a great free and open-source software, R provides much more flexibility than Excel. R can also read nearly any type of data without changing it (often Excel will convert data that might look like dates or scientific notation, whether you like it or not). Probably the most important reason to use R is transparency and reproducibility. In Excel, it is often very difficult to follow someones calculations as steps may be spread across different cells. Hence, Excel lacks transparency. In R each calculation is perfectly described by a line of code. This allows us to write a set of analysis code that can easily reproduce the same analysis with new data. We can also easily trace any changes made to the original dataset, or never change the original dataset but simply load it into memory and run the analysis code whenever we want to use it. In large experiments involving many collaborators, using consistent datasets and analysis scripts are very important.

Another of the many great things about R is its beautiful integrated development environment (IDE, a program that provides a clean work space and numerous tools to assist you in coding). A good IDE can make learning a language much easier, and even if you are an experienced coder a good IDE can make you much more efficient.

## RStudio

This wonderful IDE for R is called RStudio. For your personal computer it can be downloaded [here](http://www.rstudio.com). (You will need to install R before installing RStudio).

Let’s take a look around the RStudio window.

You will see several panes within the RStudio window.

### The Console pane

We’ll start with the **Console** pane. It is in the bottom-left corner labeled **Console** at the top left. (If the **Source** pane is closed, the **Console** pane may take up the whole left side of the window). The **Console** is essentially an R command line window. You can type in anything following the > then press return (or Enter) and whatever you typed will be interpreted by R and the appropriate output will be returned. R understands all basic algebra as well as logical expressions (aka Boolean expressions, such as 5<7 or True & False = False). Try running the following lines in the console!

2+2

#> [1] 4

9/3

#> [1] 3

3^3

#> [1] 27

2>3

#> [1] FALSE

2<3

#> [1] TRUE

### The Source pane

In the top left of the screen is the **Source** pane. (If you don’t see this pane, create a new .R file by pressing shift + command(or ctrl) + N, or click the universal ‘New Document’ icon and select ‘R script’). **Source** shows the ‘.R’ files containing the scripts (multiple lines of related code) with which you are currently working. This is where you will spend most of your time building and testing lines of code and then combining them to make scripts for data analysis.

In addition to being a basic calculator, R interprets functions or variables that can be assigned to and represented by words. Variables (any piece of data of any variety) will be single words possibly followed by a $ or square brackets [] if the variable is a matrix (a two dimensional array of data, where matrix$y would return column y and matrix[x,y] would return the piece of data in row x and column y). Functions (blocks of code that perform a specific function) are followed by parentheses containing the arguments/parameters on which that function will operate.

#### Variables and Functions

R is an object oriented language. This means we can use R to create abstracted objects that contain data (of any type, shape or size) called variables or procedures/methods (individual blocks of code) called functions. There are numerous functions and datasets included in the base R installation. Also, as an open source language countless programmers in the R community have written useful functions and created useful datasets that are freely available in the form of R-packages (more on these later). You can also write your own!

The big difference between using RStudio and running R from the command line is that this pane has an auto-complete feature. Try typing pri into R-script in the **Source** pane and pressing tab. RStudio automatically provides you with a list of all the available functions and variables beginning with ‘pri’!

You can navigate this list using the arrow keys or your mouse. When you select a particular object, RStudio also gives you some information about that object. Navigate down to print (if there are multiple, select the one that has {base} at the far right) and press tab. You will see that RStudio has completed print in the console and added a set of parentheses because print is a function, print(). Now we can add arguments that this function will operate on, within the parentheses. But what does this function do? To figure out type ?print in the **Console** and press return. This opens the documentation for this function in the **Help** pane. A ? before any function name, or passing a function name to the help() function will do the same.

### The Help pane

This pane is essentially a browser window for R documentation. You can also search for functions or variables in R and all of the installed packages on your computer using the search box at the top. You can search within a documentation page using the *Find in Topic* box.

Using this pane you should be able to answer almost any question you have about any R function.

All R documentation follows standard formatting. **Description** is pretty self explanatory. **Usage** demonstrates how you use the function, sometimes with specifics for different variable types. For print this shows us that print takes the input argument x (an argument is just variable that is used in a function). If x is a ‘factor’ or a ‘table’, print will also take some additional arguments. In the **Usage** section the default value of each argument is listed (e.g. FALSE is the default value for argument quote). A description of each argument is listed below in the **Arguments** section. **Value** is the type of data returned by the function. There are a few other self-explanatory sections and finally **Examples**. This is often one of the most useful sections as it shows you how to use the function. The code in **Examples** can be copied and pasted into the console and run.

### The Files/Plots/Packages/Help/Viewer pane

The **Help** pane contains additional tabs that can also be quite helpful. **Files** allows you to navigate through folders on your computer and open files. **Plots** shows you the most recent plot your code has produced and allows you to save it. **Packages** allows you to install and load packages into memory. Packages are bundles of code that other people have written and shared with the community (more about packages later).

##### ???Questions???

Choose any function (type in your favorite letter and press tab to complete the function name) and open its documentation in the **Help** pane. What function did you choose? *answer here*

Copy the **Usage** section of your chosen function below.  
*answer here*

Additionally, there is a search engine specific to R resources including the documentation, blogs, books and questions users have asked on discussion boards. This invaluable resource is at [Rseek.org](http://rseek.org). This is especially helpful if you want to find a function to perform a specific task.

#### Comments in code

To take notes about what your code does or how to use your code you can make “comments” by prepending any text with a #. This can also be useful to prevent a line code from being executed.

#### OK, back to the Source pane…

You should have print() in there now.  
Put your cursor in the middle of the parentheses and press tab. RStudio will feed you all of the arguments of this function using auto-complete! Press tab again and x = will appear in the parentheses. Type "hello world" and press command/ctrl + return/enter. This will copy your line of code into the console and execute it. Amazing! Your first line of R code worked, hopefully…

print(x = "hello world") # prints hello world in the console

#> [1] "hello world"

Take note that if you are missing the quotes around hello world R will look for a variable named hello and return an error.

If your code didn’t work, try to fix it and run it again.

command + return (ctrl + enter on Windows machines) can be used to execute a whole line or any selected code.

Now just highlight x = "hello world" within the print(x = "hello world") line and press command + return (or ctrl + enter). This will just execute the highlighted text. If everything worked properly, you have just created your first variable object in R!

### The Environment pane

See, over on the top right next to x (the variable object name) is “hello world” (the value assigned to that variable). You can now execute just print(x), and you will get [1] "hello world"! The **Environment** pane shows all of the objects you have created or stored in memory. You can view data sets or functions by clicking on them, but at the moment we only have the simple variable x. Don’t worry, we’ll practice this later.

## Variables and data types

You can create objects (variables~values, large data structures~think spreadsheets and databases, and functions) using the =, <- or -> operators. You can see what type of data (or data type) a variable is using the class function. Go ahead, try class(x). Data in R can be of several different, basic types:

|  |  |  |
| --- | --- | --- |
| Data Type | aka | Example |
| Logical | Boolean | TRUE, FALSE |
| Numeric | float | 42, 3.14, |
| Character | string | ‘a’ , “good”, “TRUE”, ‘23.4’ |
| Integer |  | 2L, 34L, 0L |
| Complex |  | 3 + 2i |
| Raw | hexadecimal | “Hello” is stored as 48 65 6c 6c 6f |

### Vectors

Vectors in R are simply ordered lists of values. These values can be of any type (strings, numerics, Boolean, etc), **but they must all be of the same type, or R will force them to be the same.** We can construct vectors using the c() function.

Let’s run through a quick example:

col\_names <- c('plant', 'genotype')  
col\_names

#> [1] "plant" "genotype"

##### ???Question???

What is c abbreviating? (i.e. what is the title of the c() function?) *answer here*

What are the arguments that you can pass to c()? *answer here*

Now we have a vector of strings. We can access the individual elements (the values we put in our vector) using the square bracket operator.

col\_names[1]

#> [1] "plant"

col\_names[2]

#> [1] "genotype"

#Note that the indices begin at 1 in R!!!  
col\_names[0]

#> character(0)

We can also change elements or add elements to the vector using the bracket operator.

col\_names[2] <- 'phenotype'  
col\_names[3] <- 'root\_length'  
col\_names

#> [1] "plant" "phenotype" "root\_length"

col\_names[4] <- FALSE  
col\_names

#> [1] "plant" "phenotype" "root\_length" "FALSE"

##### ???Question???

What happened to FALSE (is it a Boolean)? *answer here*

Write a block of code to test what would happen if we instead added a character string to a vector of logical values (i.e. make a new variable containing a few Boolean values, then add a string to that vector)! What happens? *answer here*

We can also do mathematical or logical operations on entire vectors.

col\_names == FALSE

#> [1] FALSE FALSE FALSE TRUE

vector <- c(1, 2, 3, 4, 5)  
6\*vector

#> [1] 6 12 18 24 30

vector^2

#> [1] 1 4 9 16 25

vector > 2

#> [1] FALSE FALSE TRUE TRUE TRUE

### Matrices, Arrays and Lists

**Matrices** are two dimensional data sets and **Arrays** are N-dimensional data sets. Like vectors these must be made of a single data type. For more info ?matrix and ?array.

Lists are more complex data structures that are similar to vectors but allow multiple data types. Lists can contain vectors as elements and even other lists! This makes them potentially N-dimensional but clunky to work with. You might encounter them if you use R in the future. For more info ?list.

### Data frames

Variables in R are not limited to just strings or integers or even matrices. You can store and operate on entire spreadsheets with columns of defined data types, using what R calls ‘data frames’. Data frames have columns that are made of vectors. The data frame is one of the most fundamental data structures used in R. ?data.frame provides a wealth of knowledge about data frames, but let’s just go ahead and make one! Run the following code.

L3 <- LETTERS[1:3]  
fac <- sample(L3, 10, replace = TRUE)  
d <- data.frame(x = 1, y = 1:10, fac = fac)  
#notice how the columns of the data frame can be named using '=', just as if we were creating individual vectors  
d

#> x y fac  
#> 1 1 1 C  
#> 2 1 2 C  
#> 3 1 3 C  
#> 4 1 4 B  
#> 5 1 5 A  
#> 6 1 6 B  
#> 7 1 7 C  
#> 8 1 8 A  
#> 9 1 9 B  
#> 10 1 10 A

class(d)

#> [1] "data.frame"

##### ???Questions???

What is LETTERS? What is L3? *answer here*

What does sample do? *answer here*

What does setting the replace argument of sample to TRUE do? Try sample(L3, 10, replace = FALSE)

Now we have a data frame d with 10 rows and 3 columns. You can retrieve individual columns using the $ operator. Try it, d$fac!. Wait a minute, why is this no longer a column? The columns of a data frame are actually just vectors.

##### ???Question???

What class of data is d$fac? *answer here*

You can also create new columns using the $ operator. For example we could make a column called new\_column that contains "new\_column" by executing d$new\_column <- "new\_column".

#### Factors

Factors used to be an efficient way of storing large vectors of repetitive discrete or categorical data. Factors do this by translating the potentially long individual pieces of data into integers, using a table called levels. Try levels(d$fac). This gives us a list of all the unique possible values in d$fac. R creates a key (1 = A, 2 = B, 3 = C) to read and write this factor. In this way long level values, like sentences, or large datasets, like thousands of lines, are compressed. To see the compressed version of d$fac we can use as.integer(d$fac). R now stores large data structures by indexing values like this regardless of whether it’s a factor or not. Despite this fact there are still some useful features of factors. For example, if we are adding a dataset from a new replicate of an experiment to an existing dataset, columns that are factors will only allow us to add values that match our existing levels. This will often help you find typos in your dataset. Additionally, some functions require factor variables, like the ANOVA functions we will use later.

Giving ?factor a look, you will see that we can also assign a particular order to the levels of a factor. This can be handy for ordering variables when plotting. We can also assign labels to the levels, just in case your level names are too abstracted to be understandable.

However when manipulating data frames containing factors you must be careful because some functions may interpret factors as their integer values! We could also avoid creating a factor in our data frame and just keep this column as characters by including stringsAsFactors = F in our call to data.frame().

Going back to our data frame d, similar to vectors we can access rows, columns and elements of the data frame using the square bracket operator. I’ll suppress the output below and let you run these examples yourself.

#get the first row of d  
d[1,]  
#get the first column of d  
d[,1]  
#get the column named 'fac'  
d[,'fac']  
#or  
d[['fac']]  
#or (most efficient and readable)  
d$fac  
#get the element in the 5th row and 3rd column  
d[5,3]

We can also perform calculations or other operations on the elements of a data frame.

d[,2] + 1  
d[[2]] + 1  
d[,2] \* 2  
#similarly for logical operations, note that logical 'is equivalent to' is '=='  
d[,3] == 'B'  
d$y <= 5  
  
#we can also use functions to perform complex calculations  
mean(d$y)  
median(d$y)  
sum(d$y)

Just like with vectors we can change elements or add elements to a data frame.

##### ???Question???

How would you add a column to d with the integer values representing d$fac? *answer here*

What is the mean of your new column of d? Copy the code you used. *answer here*

What is the median or your new column of d? Copy the code you used. *answer here*

What is the sum of your new column? Copy the code you used. *answer here*

What fraction of the sum of your new column is each row’s value? Make a new column for d showing this fraction. Copy the code you used. *answer here*

## Saving and reading data

Once we are done manipulating data in R we often want to save the data for safe keeping or to share with our collaborator or boss. To save data we can use write of save functions of which there are several varieties. For dataframes, we generally want to save as a .csv file using the write.csv function.

# save d as "d.csv"  
write.csv(d, "d.csv")

But what about if we want to read a .csv file in to R? To do this we will use the read.csv function. We’ll have to assign the results of this function to a variable.

d <- read.csv("d.csv")

So using this function we can read in any data after saving it as a .csv in Excel. If we don’t know the exact path of our dataset but would like to bring up a file browser window to choose it we can use the file.choose function.

d <- read.csv(file.choose())

# Data analysis and visualization

This lesson in basic analysis of variance focuses and extends upon an example originally published in [Brady et al. 2015](https://www.ncbi.nlm.nih.gov/pubmed/26220933). Read through the introduction of this paper and come back when you’re at the “ANOVA PROPERLY TESTS GENETIC INTERACTIONS” section.

## ANOVA PROPERLY TESTS GENETIC INTERACTIONS

As you have read analysis of variance or ANOVA facilitates comparisions between and among treatments, genotypes, and environments as well as interactions between these variables.

Let’s simulate an experiment in which we want to determine if two genes interact to contribute to a particular phenotype. These genes could be redundant or interact in some epistatic manner. To figure this out we will need to measure the phenotype of interest for the two single mutants, the wild type, and the double mutant. For this example, let’s use root length as the phenotype. Suppose we have two mutants in two genes that we suspect have an effect on root length, as well as a double mutant containing mutations in both genes.

First, we will simulate an experiment by drawing root length values from a random normal distribution for each genotype, and then compile these values we drew into a data frame. We will pick means and standard deviations for the normal distributions from thin-air for now. Feel free to come back later and change these values to see how differences in means and standard deviations will affect the outcomes. ##Experiment 1

# Draw 3 measurements from a random normal distribution for each plant line  
WT <- rnorm(n = 3, mean=100, sd=10) # WT genotype  
M1 <- rnorm(n = 3, mean=85, sd=10) # Single mutant in Gene 1  
M2 <- rnorm(n = 3, mean=85, sd=10) # Single mutant in Gene 2  
DM <- rnorm(n = 3, mean=70, sd=10) # Double mutant in Gene 1 and Gene 2  
# Create a column of the simulated phenotype measurements that we randomly drew above  
Phen <- c(WT, M1, M2, DM)

Now we will create a dataframe with columns for the status of each gene, that this is the first experiment, and the measurement of phenotype. Then we will combine these columns with our root length measurements in a data frame.

# Create a column for the genotype at gene 1  
# 3 wild-types, 3 single mutants in Gene 1, 3 single mutants in Gene 2, 3 double mutants  
M1G <- c("WT", "WT", "WT", "M1", "M1", "M1", "WT", "WT", "WT", "M1", "M1", "M1")  
# Create a column for the genotype at gene 2  
M2G <- c("WT", "WT", "WT", "WT", "WT", "WT", "M2", "M2", "M2", "M2", "M2", "M2")   
# Create a column for the repeat of the experiment, as we will be repeating this process several times  
Exp <- c("1", "1", "1", "1", "1", "1", "1", "1", "1", "1", "1", "1")  
  
Rep1 <- data.frame(Exp,M1G,M2G,Phen)  
Rep1

#> Exp M1G M2G Phen  
#> 1 1 WT WT 100.72028  
#> 2 1 WT WT 119.09597  
#> 3 1 WT WT 102.18917  
#> 4 1 M1 WT 75.29847  
#> 5 1 M1 WT 94.74115  
#> 6 1 M1 WT 94.93377  
#> 7 1 WT M2 89.60451  
#> 8 1 WT M2 88.12632  
#> 9 1 WT M2 77.26188  
#> 10 1 M1 M2 62.95257  
#> 11 1 M1 M2 92.10125  
#> 12 1 M1 M2 65.32470

Note that the values you drew will be different from those above and different from everyone else. Now we also need to convert each of these independent variables from strings to factors, as required by the ANOVA function.

class(Rep1$M1G)

#> [1] "factor"

Rep1$M1G <- factor(Rep1$M1G)   
Rep1$M2G <- factor(Rep1$M2G)   
Rep1$Exp <- factor(Rep1$Exp)

Now that we have our data frame set up we can visualize our data and test our hypothesis.

First, let’s plot these data to get an idea of what they look like

### Plotting with ggplot2

The ggplot2 package provides a very powerful and intuitive (once you get the hang of it) way of visualizing data. This can help us present this data in the most clear and understandable way. But first, we will need to download and install this package.

#### Packages

There are three places where R packages are available:  
1. [CRAN](http://cran.r-project.org/web/packages/) contains a huge variety of general packages (including ggplot2 and several other packages we will use here),  
2. [bioConductor](http://bioconductor.org/) contains packages related to high throughput biological (mostly -omics) data,  
3. Packages in development may be available from code repositories such as [GitHub](http://github.com), [Bitbucket](http://bitbucket.org) or others.

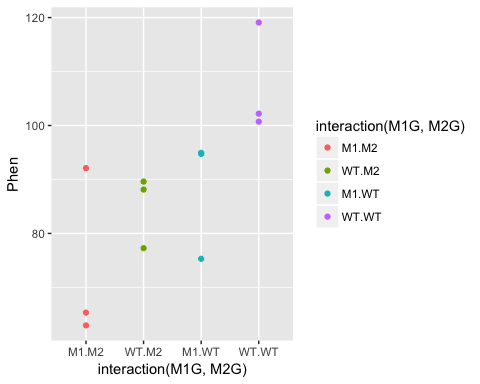
#install a package from CRAN   
install.packages('ggplot2')  
  
  
#we will also need the 'magrittr', ‘dplyr’, 'forcats', 'car', 'agricolae', and 'plyr' packages later on  
install.packages('magrittr')  
install.packages('dplyr')  
install.packages("forcats")  
install.packages("car")  
install.packages("agricolae")  
install.packages("plyr")

The ‘gg’ in ggplot2 stands for ‘grammar of graphics’. This grammar allows us to build visual representations of data to in a manner similar to how grammar allows us to build sentences to communicate abstract concepts. The grammar of graphics allows precise control over each item in a graph, and also provides convenient defaults. The qplot function uses these default values to make quick plots. Similar to how grammar of language directs our use of nouns, verbs and adjectives, the grammar of graphics directs our use of aesthetics, geometries and layers.

Within the grammar of graphics aesthetics are visual parameters to which you can map a variable. So for example here, x, y and color are aesthetics to which we want to map the variables genotype and phenotype. If we extend the language grammar metaphor, aesthetics are like nouns, you can have several nouns in a sentence. But we also need a verb to make a sentence, right? The verb-equivalents in the grammar of graphics are called geometries. These geometries are the methods of representing data, e.g. bar plots, scatter plots, and box-and-whisker plots. In ggplot2 these geometries are created using which always begin with geom\_, so you can type ‘geom’ into the **Help** pane and see a list of all the possibilities. In qplot we just leave off the geom\_ when specifying the geom argument

Let’s explore this data a bit using the qplot function.

#before using any package we must load it into our workspace  
library('ggplot2')  
qplot(x = interaction(M1G, M2G), y = Phen, data = Rep1, color = interaction(M1G, M2G), geom = "point")



##### ???Question???

What can we say about this data? Write a quick summary of this graph and formulate a hypothesis. *answer here*

Is one gene contributing more to the root length phenotype than another? *answer here*

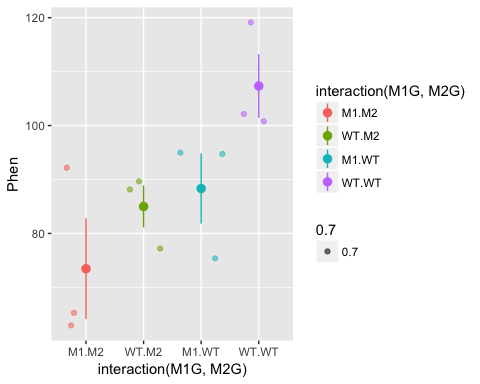
Does the double mutant have a more or less severe phenotype than the individual mutants? *answer here*

Do you think the genes interact to contribute to the phenotype?  
*answer here*

OK! Back to our analysis of variance. We want to ask whether or not the root length values for wildtype plants, the single mutants in each gene, and the double mutant all came from the same distribution. To ask this question analysis of variance, as the name implies, compares different variances in our experiment. **ANOVA compares the variance between the groups we are comparing to the variance within each group. So in this case we want to compare the phenotypic variance between the means of the WT and each mutant to the phenotypic variance within each of these groups. Another way of thinking about this is that we are comparing the effect of the variable that separates the groups, to the precision of our measurment of this effect.** The variance between groups is called the systematic variance and the variance within groups is called the error variance.

Let’s visually explore this ANOVA idea a bit. For each genotype we can calculate the mean phenotype and the variance around the mean. These is the “within” group variance. We’ll add means and standard error to this plot using the stat\_summary function to represent “within” variance. We can use the alpha aesthetic to change the saturation of the colors to make the mean stand out a bit more and hide the raw data somewhat. We can also use the geom argument jitter in the qplot function to prevent the mean and standard error from being plotted overtop of the raw data.

qplot(interaction(M1G, M2G), y = Phen, data = Rep1, color = interaction(M1G, M2G), alpha = 0.7, geom = "jitter") + stat\_summary(fun.data = "mean\_se", alpha = 1)



The “between” group mean and variance is calculated using the mean for each group. Here, we’ll split up our data frame Rep1 into groups by the experiment, genotype at gene1, and genotype at gene2. Then we will calculate the mean for each of these groups. To do this we will use the dplyr package. We will also use the pipe function %>% from the magrittr package. The pipe function passes the result of the code on the left side to the first argument of the code on the right side.

# Load the necessary packages   
library(dplyr)  
library(magrittr)  
# Take the Rep1 dataset and split it into groups  
Rep1 %>% group\_by(Exp, M1G, M2G) %>%  
# Note that we don't need to use quotes with functions from the dplyr package  
# Technically each of Exp, M1G, and M2G are passed to the "..." argument of the group\_by function  
# We added a pipe function to the end of this line so we can pass these groups to the `summarise` function to calculation the mean of each group  
# We can then use the right arrow to assign this new data frame of the means to an object  
 summarise(Phen = mean(Phen)) -> means  
means$M1G <- "mean"  
means$M2G <- "mean"  
# Create a new data frame with the mean rows combined with our data  
Rep1means <- bind\_rows(Rep1, means)

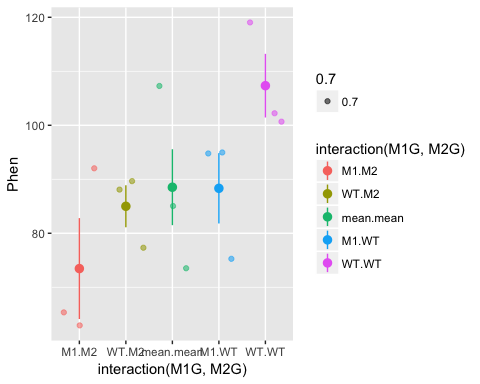
#> Warning in bind\_rows\_(x, .id): binding factor and character vector,  
#> coercing into character vector

#> Warning in bind\_rows\_(x, .id): binding character and factor vector,  
#> coercing into character vector

#> Warning in bind\_rows\_(x, .id): binding factor and character vector,  
#> coercing into character vector

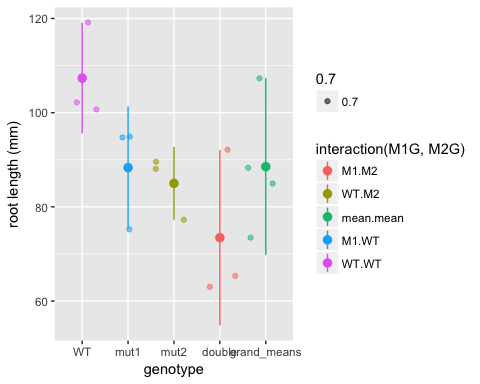
#> Warning in bind\_rows\_(x, .id): binding character and factor vector,  
#> coercing into character vector

qplot(x = interaction(M1G, M2G), y = Phen, data = Rep1means, alpha = 0.7, color = interaction(M1G, M2G), geom = "jitter") + stat\_summary(fun.data = "mean\_se", alpha = 1)



So now the mean.mean column contains four points representing the means of the four groups. The mean of these means, called the grand mean, is represented by the large point and the variance of the group means about the grand mean represents the between groups variance, but here we’ve used the mean\_se function to calculate standard error. The true value of the variance is the difference between each data point and the mean. We can write our own function to calculate the true variance. Don’t get bogged down in the code here (unless you want to :), just pay attention to the graphs. We can also use the forcats package to move the mean.mean column to the end of the graph and relabel each of the columns. In general the forcats package has functions that allow us to arrange factors (forcats is a rearrangement of factor, haha!). Let’s also clean up the axis labels by adding labs to the plot.

library(forcats)  
qplot(x = fct\_recode(fct\_relevel(fct\_rev(interaction(M1G, M2G)), "mean.mean", after = Inf), double = "M1.M2", mut2 = "WT.M2", mut1 = "M1.WT", WT = "WT.WT", grand\_means = "mean.mean"), y = Phen, data = Rep1means, alpha = 0.7, color = interaction(M1G, M2G), geom = "jitter") +   
 stat\_summary(fun.data = function(y){  
 data.frame(  
 ymin = mean(y) - sum(mean(y) - y[which(y<mean(y))]),  
 y = mean(y),  
 ymax = mean(y) + sum(y[which(y>mean(y))] - mean(y)))  
 }, alpha = 1) +   
 labs(x = "genotype", y = "root length (mm)")

 Now we have a pretty graph showing the mean and variance for each genotype (the first four columns) as well as the grand mean and variance (the last column).

##### ???Question???

How does the variance in each group compare to the variance in the grand mean? *answer here*

Now, let’s formulate that hypothesis as a mathematical model so that we can perform the ANOVA analysis in R.

Generally, we know that we have a response variable that we are measuring, that is a function of some set of predictors.

The ~ here means that this is a hypothetical or approximate relationship between the response and predictors as there are certainly other factors that we cannot predict or error that we cannot account for in our model.

##### ???Question???

What is the dependent variable in our simulated experiment? This is also known as the response variable and is typically continuous. *answer here*

What are the independent or predictor variables of that response? These are typically discrete or categorical for ANOVA analysis. *answer here*

Expanding this function a bit, we can assess each individual genes functional contribution to the phenotype as well as the extent to which they interact (or interfere) with one another. Therefore, our final model that we would like to test is

This is called a general linear model. This should remind you of a linear regression analysis from way back in Algebra II. We have written out an equation that we think should fit our data. ANOVA assigns the deviation or error between our model and our data into groups based on the predictors. It does this by comparing the variance between groups of predictors to the variance within groups of predictors.

Now that we have our model formulated, let’s set this up in the ANOVA framework in R to finally get an answer to our question: “Are these genes interacting to regulate to root length?”

?aov  
Exp1 <- aov(Phen ~ M1G + M2G + M1G:M2G , data=Rep1)   
summary(Exp1)

#> Df Sum Sq Mean Sq F value Pr(>F)   
#> M1G 1 699.9 699.9 5.194 0.0521 .  
#> M2G 1 1038.0 1038.0 7.703 0.0241 \*  
#> M1G:M2G 1 41.9 41.9 0.311 0.5924   
#> Residuals 8 1078.0 134.8   
#> ---  
#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

So what does this summary tell us? Sum Sq stands for sum of squares. This is the sum of the squared deviation between the groups of the variable. Mean Sq is Sum Sq divided by the degrees of freedom or Df in that variable and is essentially the variance of that variable. The degrees of freedom are the number of groups for that variable minus one. Residuals is the total within groups variance.

Now for the comparison of the variances that we’ve been talking about all along. The F value is the ratio of the Mean Sq of the variable over the Residuals. So the larger the F value the more the variance between the predictor levels dominates the variance within predictor levels (i.e. the between treatment variance is greater than the sampling variance). So for our experiment a gene with an F value much greater than one indicates that the variance among genotypes is much greater than the variance within genotypes. The Pr(>F), is the probability that the data are consistent with the null hypothesis, which in this case is that the variance between genotypes of gene is not different from the variance within the genotypes. This is referred to as the p-value. So if this p-value is less than the confidence threshold you have set for your experiment (frequently 0.05), you can reject the null hypothesis and conclude that the gene has a significant effect on the phenotype.

However, **we cannot accept the null hypothesis** for those independent variables with p-values greater than our confidence threshold!!! We can only fail to reject the null, meaning we can only say that we don’t have enough data to detect an effect on the phenotype.

Chances are with this first small experiment you don’t have enough data to say anything significant about the interaction between gene1 and gene2 (i.e. we don’t have enough evidence to reject the null hypothesis).

Let’s repeat the experiment!

## Experiment 2

WT <- rnorm(3, mean=100, sd=10)  
M1 <- rnorm(3, mean=85, sd=10)  
M2 <- rnorm(3, mean=85, sd=10)  
DM <- rnorm(3, mean=70, sd=10)  
Exp <- c("2", "2", "2", "2", "2", "2", "2", "2", "2", "2", "2", "2")   
Phen <- c(WT, M1, M2, DM)  
  
Rep2 <- data.frame(Exp,M1G,M2G,Phen)   
Rep2$M1G <- factor(Rep2$M1G)   
Rep2$M2G <- factor(Rep2$M2G)   
Rep2$Exp <- factor(Rep2$Exp)

ANOVA analysis for trial 2

Exp2 <- aov(Phen ~ M1G + M2G + M1G:M2G , data=Rep2)   
summary(Exp2)

#> Df Sum Sq Mean Sq F value Pr(>F)   
#> M1G 1 205.1 205.1 2.769 0.1347   
#> M2G 1 665.1 665.1 8.978 0.0172 \*  
#> M1G:M2G 1 73.0 73.0 0.986 0.3499   
#> Residuals 8 592.6 74.1   
#> ---  
#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##### ???Question???

What can you conclude from this experiment? *answer here*

Because all good things come in threes, and the conclusions likely differ between experiments, let’s repeat the experiment a third time.

## Experiment 3

WT <- rnorm(3, mean=100, sd=10)  
M1 <- rnorm(3, mean=85, sd=10)  
M2 <- rnorm(3, mean=85, sd=10)  
DM <- rnorm(3, mean=70, sd=10)  
Exp <- c("3", "3", "3", "3", "3", "3", "3", "3", "3", "3", "3", "3")   
Phen <- c(WT, M1, M2, DM)  
  
Rep3 <- data.frame(Exp,M1G,M2G,Phen)   
Rep3$M1G <- factor(Rep3$M1G)   
Rep3$M2G <- factor(Rep3$M2G)   
Rep3$Exp <- factor(Rep3$Exp)

ANOVA analysis for trial 3

Exp3 <- aov(Phen ~ M1G + M2G + M1G:M2G , data=Rep3)   
summary(Exp3)

#> Df Sum Sq Mean Sq F value Pr(>F)   
#> M1G 1 886.5 886.5 10.660 0.0114 \*  
#> M2G 1 793.9 793.9 9.548 0.0149 \*  
#> M1G:M2G 1 7.0 7.0 0.084 0.7796   
#> Residuals 8 665.2 83.2   
#> ---  
#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

## Combined ANOVA

Now we can combine the experiments and control control for trial number effects.

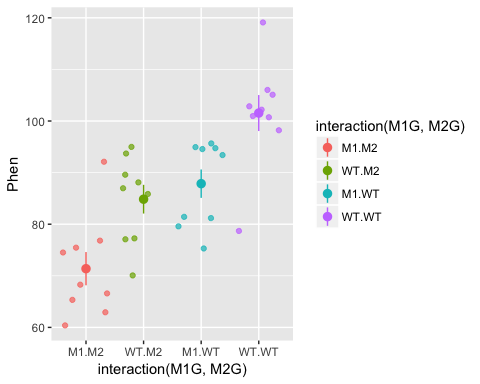
AllExp <- rbind(Rep1, Rep2, Rep3)  
# Notice how we add an Exp variable to the model  
Full <- aov(Phen ~ M1G + M2G + M1G:M2G + Exp , data=AllExp)   
summary(Full)

#> Df Sum Sq Mean Sq F value Pr(>F)   
#> M1G 1 1659.2 1659.2 19.063 0.000138 \*\*\*  
#> M2G 1 2475.9 2475.9 28.447 9.08e-06 \*\*\*  
#> Exp 2 115.5 57.7 0.663 0.522507   
#> M1G:M2G 1 0.1 0.1 0.001 0.972304   
#> Residuals 30 2611.1 87.0   
#> ---  
#> Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

### Advanced plotting with ggplot2

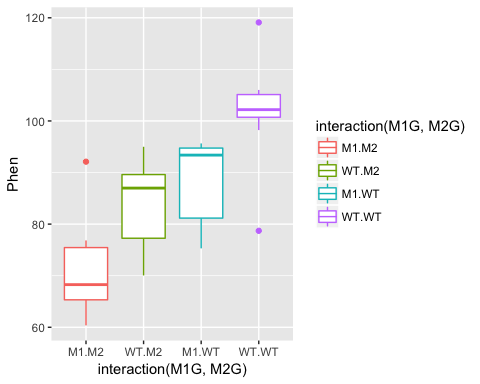
Let’s plot the full data set and see if the ANOVA results match our visual expectations. This time instead of using qplot we’ll use the more flexible and powerful ggplot function. To do this we create a ggplot element that specifies the data and aesthetic mapping. This sets up which columns of the dataset will be mapped to the coordinate axes and colors, shapes, linetypes or other aesthetic element of the graph. Then we can add geometries to this ggplot object to layer the data onto the plot. We can also add the summaries of the data onto the plot as before.

ggplot(data = AllExp, mapping = aes(x = interaction(M1G, M2G), y = Phen, color = interaction(M1G, M2G))) + geom\_point(alpha = 0.7, position = "jitter") + stat\_summary(fun.data = "mean\_se", alpha = 1)



Notice how we can alter the aesthetics of different geoms. Let’s try plotting this as a boxplot as well. All we have to do is specify a a different geom.

ggplot(data = AllExp, aes(x = interaction(M1G, M2G), y = Phen, color = interaction(M1G, M2G))) + geom\_boxplot()



In the end you will likely be able to conclude that both gene1 and gene2 have significant effects on the phenotype, but that there is not evidence to suggest that there is a significant interaction between the two genes. Also, the experimental trial most likely does not have an effect. This is all dependent on the mean and variance of the distributions we set in the beginning.

# Draw 3 measurements from a random normal distribution for each plant line  
WT <- rnorm(n = 3, mean=100, sd=10) # WT genotype  
M1 <- rnorm(n = 3, mean=85, sd=10) # Single mutant in Gene 1  
M2 <- rnorm(n = 3, mean=85, sd=10) # Single mutant in Gene 2  
DM <- rnorm(n = 3, mean=70, sd=10) # Double mutant in Gene 1 and Gene 2

##### ???Question???

What value of the mean of the double mutant distribution would be likely to result in a significant interaction between the two genes? *answer here*

## Post hoc testing

While the ANOVA summary tells us which independent variables explain a significant amount of variance in the experiment, it doesn’t allow us to compare the levels of the independent variables. How can we tell if the mutant in gene 1 has a different phenotype from the mutant in gene 2? What if the experimental trial did have an effect? How could we figure out which trial is different from the others?

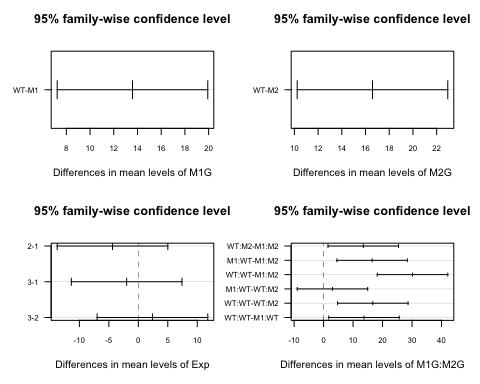
To compare group or level means and variances and answer these questions we will need to do what is called a *post hoc* test, following our ANOVA analysis. One of the most commonly used post hoc tests is Tukey’s Honest Significant Difference test. This compares all of the group means in a pairwise manner and corrects the confidence threshold since we are making multiple comparisons. We do this test on our ANOVA object Full from above.

tuk <- TukeyHSD(Full)  
tuk

#> Tukey multiple comparisons of means  
#> 95% family-wise confidence level  
#>   
#> Fit: aov(formula = Phen ~ M1G + M2G + M1G:M2G + Exp, data = AllExp)  
#>   
#> $M1G  
#> diff lwr upr p adj  
#> WT-M1 13.57771 7.226701 19.92871 0.0001384  
#>   
#> $M2G  
#> diff lwr upr p adj  
#> WT-M2 16.58615 10.23515 22.93716 9.1e-06  
#>   
#> $Exp  
#> diff lwr upr p adj  
#> 2-1 -4.380456 -13.769884 5.008973 0.4915590  
#> 3-1 -1.982952 -11.372381 7.406476 0.8618984  
#> 3-2 2.397504 -6.991925 11.786932 0.8051714  
#>   
#> $`M1G:M2G`  
#> diff lwr upr p adj  
#> WT:M2-M1:M2 13.468835 1.510507 25.42716 0.0225840  
#> M1:WT-M1:M2 16.477283 4.518954 28.43561 0.0040257  
#> WT:WT-M1:M2 30.163860 18.205531 42.12219 0.0000008  
#> M1:WT-WT:M2 3.008447 -8.949882 14.96678 0.9023482  
#> WT:WT-WT:M2 16.695025 4.736696 28.65335 0.0035338  
#> WT:WT-M1:WT 13.686578 1.728249 25.64491 0.0200483

So more than likely, you will see for your full experiment with 3 replications that WT is significantly different from M1 and M2. This is shown in the first two sections of the TukeyHSD summary which compare the levels (wildtype vs mutant) of the factors M1G and M2G. The p adj is the p-value (adjusted for the multiple comparisons) resulting from the equivalent of a *t*-test comparing the two groups. So if the p adj value is less than the accepted confidence level (typically 0.05), then the two compared groups are significantly different from one another. Hopefully, each replication of the experiment will not be significantly different from the others, i.e. the adjusted p-values will be greater than 0.05. We can visualize the differences in the means by plotting the TukeyHSD results. This uses base R plotting which is much less user-friendly than ggplot. Don’t get bogged down in the code here, just pay attention to the graphs.

par(mfrow = c(2,2), cex = 0.65)  
plot(tuk, las = 1, cex.axis = 0.75)



This plots the confidence intervals around the difference between the means for each pair of groups. If these confidence intervals intersect zero we cannot conclude that the two groups being compared are significantly different.

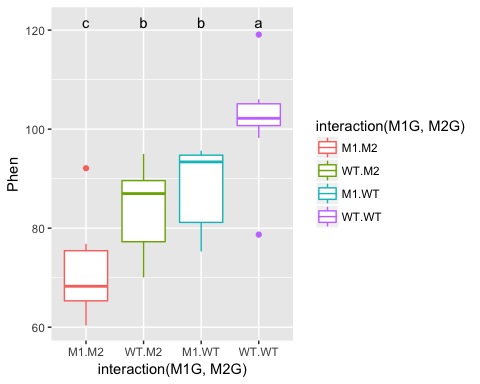
We could also use the HSD.test function from the agricolae package, which puts the groups into groups that are not significantly different from each other.

library("agricolae")  
HSD.test(y = Full, trt = c("M1G","M2G"), console = TRUE)

#>   
#> Study: Full ~ c("M1G", "M2G")  
#>   
#> HSD Test for Phen   
#>   
#> Mean Square Error: 87.03624   
#>   
#> M1G:M2G, means  
#>   
#> Phen std r Min Max  
#> M1:M2 71.38054 9.665252 9 60.36359 92.10125  
#> M1:WT 87.85782 8.256381 9 75.29847 95.64224  
#> WT:M2 84.84937 8.347418 9 70.04983 94.99391  
#> WT:WT 101.54440 10.466894 9 78.70690 119.09597  
#>   
#> Alpha: 0.05 ; DF Error: 30   
#> Critical Value of Studentized Range: 3.845401   
#>   
#> Minimun Significant Difference: 11.95833   
#>   
#> Treatments with the same letter are not significantly different.  
#>   
#> Phen groups  
#> WT:WT 101.54440 a  
#> M1:WT 87.85782 b  
#> WT:M2 84.84937 b  
#> M1:M2 71.38054 c

Finally we can add these groups to our graph and make the titles a bit more understandable. We’ll add a text geometry to add the post hoc comparisons. Another tricky point is that we have to make sure that the x-axis labels match between our data and our HSD test. To do this we use gsub to swap out the colons for periods.

HSD.groups <- HSD.test(y = Full, trt = c("M1G","M2G"))$groups  
HSD.groups$trt <- row.names(HSD.groups)  
ggplot(data = AllExp, aes(x = interaction(M1G, M2G), y = Phen, color = interaction(M1G, M2G))) + geom\_boxplot() + geom\_text(data = HSD.groups, mapping = aes(x = gsub(":", ".", trt), y = max(Phen) + 20, label = groups), color = "black")

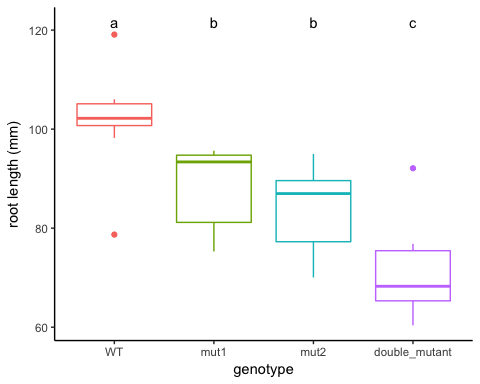


Now let’s clean this up a bit by changing the axis titles and labels. We’ll use forcats functions like before but this time with pipes to make the process a little more readable. We’ll also use a new theme to clean up the plot a bit and get rid of the legend.

# First we will use mutate from `dplyr` to add a column for the full genotype  
AllExp %<>% mutate(genotype = interaction(M1G, M2G))

#> Warning: package 'bindrcpp' was built under R version 3.4.4

# %<>% is a two way pipe that passes AllExp as the first argument to mutate and then assigns the result back to AllExp  
  
# Now to recode genotype we will unfurl the nested forcats function using pipes  
AllExp$genotype %<>%  
 fct\_rev() %>%  
 fct\_recode(double\_mutant = "M1.M2", mut2 = "WT.M2",   
 mut1 = "M1.WT", WT = "WT.WT")  
  
HSD.groups$trt %<>%  
 fct\_rev() %>%  
 fct\_recode(double\_mutant = "M1:M2", mut2 = "WT:M2",   
 mut1 = "M1:WT", WT = "WT:WT")  
  
ggplot(data = AllExp, aes(x = genotype, y = Phen, color = genotype)) + geom\_boxplot() + geom\_text(data = HSD.groups, mapping = aes(x = trt, y = max(Phen) + 20, label = groups), color = "black") + xlab("genotype") + ylab("root length (mm)") + theme\_classic() + theme(legend.position="none")



## Assumptions of ANOVA

Before making any final conclusions we should check and make sure that our data satisfies the assumptions of the F-statistic, and really the assumptions of nearly any statistical test. These are normality (that the groups are normally distributed), *homogeneity of variance* (that the variances are similar for each group) and independence (that each observation is independent of the others, e.g. each measurement was of a different plant). It’s not a deal-breaker if our data defies one of these assumptions. There are variations of ANOVA procedures that are robust to certain violations, but these will have to wait for another class.

How do you test these assumptions?

### Normality

We can visualize deviations from normality for each group of measurements using qqnorm. The tricky thing is that we need to split up our dataframe by groups. To do this we will use the plyr package.

library(plyr)

#> -------------------------------------------------------------------------

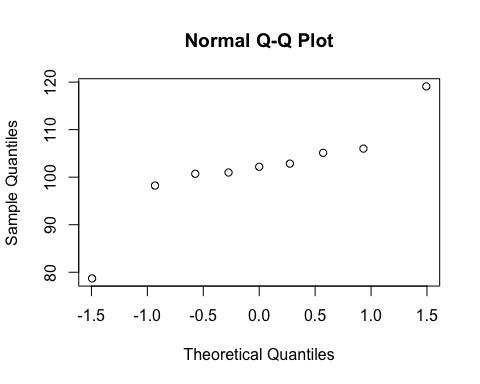
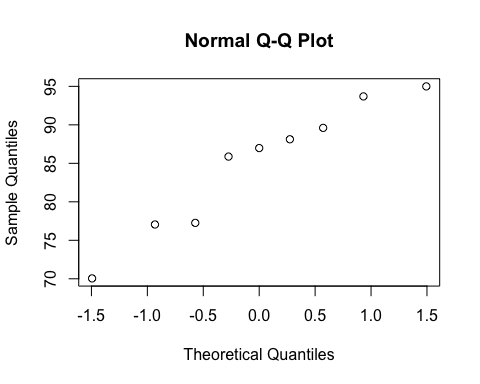
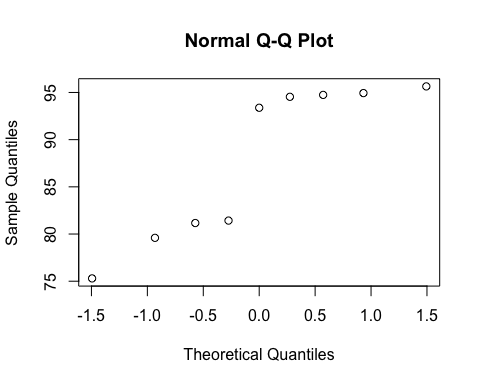
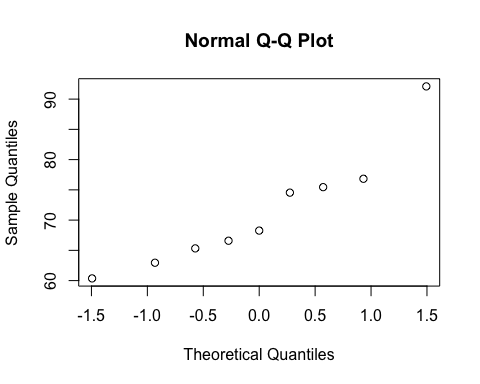
#> You have loaded plyr after dplyr - this is likely to cause problems.  
#> If you need functions from both plyr and dplyr, please load plyr first, then dplyr:  
#> library(plyr); library(dplyr)

#> -------------------------------------------------------------------------

#>   
#> Attaching package: 'plyr'

#> The following objects are masked from 'package:dplyr':  
#>   
#> arrange, count, desc, failwith, id, mutate, rename, summarise,  
#> summarize

d\_ply(.data = AllExp, .variables = .(M1G, M2G), .fun = summarise, qqnorm(Phen))



# If there is strong deviation from a straight line then there may be a violation of normality

We can run the Shapiro-Wilk test to test for deviations from normality.

AllExp %>% group\_by(genotype) %>% summarise(p\_value = shapiro.test(Phen)$p.value)

#> p\_value  
#> 1 0.5997075

A significant test indicates that we can reject the null hypothesis that the data is normally distributed.

### Homogeneity of variance

The test for homogeneity of variance is called Levene’s Test. This tests whether the variance of each group is different from the others, so a significant test would mean that the assumption of homogeneity is violated.

library('car')  
leveneTest(AllExp$Phen, interaction(AllExp$M1G, AllExp$M2G), center = median)

#> Levene's Test for Homogeneity of Variance (center = median)  
#> Df F value Pr(>F)  
#> group 3 0.0469 0.9863  
#> 32

### Independence

To test for independence we have to think “is there any reason that any measurements might be dependent on another measurement?” Typically this is due to a *repeated-measures design*. This might be that you have measured the same phenotype of different leaves from the same plant, or measured the same leaf at two different times.

Hopefully with this introduction you can now you can proceed to analyze the data you have collected in lab. Follow the same steps of formulating your hypothesis as a mathematical model, graphing the data, and performing an ANOVA and post hoc test to test your hypothesis.