Analysis of Mitochondrial Respiration:

A Guide for the Analysis and Visualization of Respirometry Data Using R

Stephen Decker, MS, ACSM-CEP
Department of Kinesiology
University of Massachusetts Amherst

Last Updated January 2021
Built with R version 4.0.3 and RStudio

Contents





Disclaimer

This is a template for the code to analyze mitochondrial respiration rates collected from the Oroboros Oxygraph O2K. I have created this file for public use, but please consider that I did this out of my free time and will continue to update and change this document as time goes on.

I ***do not*** intend to restrict access to this document. However, my only request is that if others use this document, appropriate referencing and citation to this project is greatly appreciated when necessary. Furthermore, I fully intend to update this document as my skills in R improve and more methods of analysis become available. For comments, questions, or concerns, please email me at stdecker@umass.edu or contact me via any of the links at the bottom of each page. Also, much of my knowledge has come from a vast source of publicly available sources – books, Stack Exchange, Data Novia, and others. If you wish to have any of this information, please contact me and I will gladly share my resources. I will also include links to them at the end of this document.

Please keep in mind that I am a physiologist by training. I try to do my best with statistics and code, but neither of those are something I would consider my trade. Thus, I am very open to hearing comments and critiques of this document and suggestion for improvements are much appreciated. My ultimate goal is to produce scientific discovery in the correct way, and I will gladly take any feedback on how to achieve that goal.



1. Basics of this Document

I would like to outline some of the code I think is critical to understand for this document:

1.1. Packages

```
library(readxl)
library(dplyr)
library(ggplot2)
library(ggpubr)
...
```

You'll notice that I use several packages in this document. This is simply because the analyses that I have below will call upon many functions that are spread across several packages.You may not need all of these packages to run this code, depending on what you are trying to do with this analysis. However, I will include all of these in my documentation for my use. Feel free to discard anything you think may be unused. More importantly, I find that resources, such as RDocumentation and the CRAN package search are very useful guides to understanding the capabilities and layout of each package.

To install a package, you'll simply use the `install.packages()` function with the name of the package in parentheses. Then, you must load the package from your R library using the `library()` function. To install and load a package like the `'praise'` package, do the following:

```
install.packages("praise")
```

```
library('praise')
```

Now we can begin using the functions inside this package. The 'praise' package only has one function. If we use this function in our code, we should see an output that gives us a compliment, like so:

```
praise()
```

```
## [1] "You are legendary!"
```

1.2. Taking Notes

Taking notes is a particularly useful skill to utilize in R, especially when running long lines of code (like you will probably do in graphing). To make a note, simply put a `'#'` followed with text, then start a new line to continue the code. This note will not be analyzed by R, but will remain in the interface. For an example:

```
#The 'praise' package gives me praise when I use it
praise()
```

```
## [1] "You are pioneering!"
```



```
#2 x 3 always equals 6
2*3
```

```
## [1] 6
```

1.3. Pulling data from Excel

```
df.name <- read_excel("c\\Users\\complete_file_path\\file_name.xlsx",
      sheet = "Sheet Name", col_names = TRUE)
```

This line of code will pull your data from an Excel sheet. I have provided an Excel template file on my personal GitHub page. Please feel free to use this as needed with the same stipulations mentioned earlier. The code I have made below is made for the setup in this file, so keep that in mind as this data becomes analyzed.

This script is fairly intuitive to understand — you will name your data frame (by changing "df.name") and your full file path (including the path directory) goes into the first set of quotations. If you have specific sheet names you wish to call upon, it goes into the next set of quotation marks. Lastly, if you have column names, keep this as TRUE.

1.4. Statistical Analyses

You will notice that I have several different methods of analyzing the data (t-tests, ANOVA) in this document. I have tried to include several different types of analyses and ways to streamline these analyses.

One **major** thing to consider in this process is the layout of the data being analyzed. In order to run a proper ANOVA, you ***must*** have the data organized in long format, not wide format. There are ways to do that and I cover it later in the document, but I will note that my excel files come in both wide (in the t-test tab) and long format (in the ANOVA tab) so that the analysis can be done without that conversion. However, it may be useful to explore that, especially if you want to add other variables of your own in this analysis.

1.4.1. T-tests

Performing t-tests in these types of analyses are, in my opinion, controversial at best. I've heard the argument that the addition of different substrates to the respiration medium changes the state in which the sample exists, and therefore these are independent conditions. However, I think it could easily be argued that the respiration of a sample is dependent on the individual sample and therefore the respiration of a sample in any given state is still dependent on the other states. Furthermore, there is always the issue of a family-wise error in the p-value. Therefore, I still recommend performing an ANOVA for all analyses, but it is the choice of each lab to make that decision.

```
ttest_values <- lapply(df.name[,6:16], function(x)
      t.test(x~ df.name$data_range, na.rm = TRUE))
ttest_pvalues <- data.frame(p.value =
```



```
      sapply(ttest_values, getElement, name = "p.value"))
```

For example, I have designed this code to perform several t-tests (10 to be exact — from column 6 to 16) with one line of code, then it will output a data frame with all of the p-values using the second line of code. Again, we run into the issue of family-wise p-values in this type of analysis, and I have not yet accounted for that in this analysis. I will explain this in more detail later.

1.4.2. ANOVA

The ANOVA code is pretty straightforward. With this code, you can perform a one-way or a two-way ANOVA extremely easily. I would rely more heavily on these results compared to the t-tests, following the code for a one-way ANOVA:

```
ANOVA_model <- aov(df.name$dependent.variable ~
      df.name$factor1, data = df.name)
anova(ANOVA_model)
```

Or a two-way and/or factorial ANOVA:

```
ANOVA_model <- aov(df.name$dependent.variable ~
      df.name$factor1 * df.name$factor2, data = df.name)
anova(ANOVA_model)
```

I find this process to be pretty easy to understand. The first line creates a linear model of the data using the `'aov'` function. This calls the 'dependent variable' and the grouping and condition (independent) variables. Then, the `'anova'` function will run a Type I ANOVA, completing sum of squares (Sum Sq), degrees of freedom (Df), F values, probability (Pr), and denoting significance with asterisks. Overall, this should give you everything you need for a fundamental analysis of the ANOVA. Alternatively, you can use the `Anova` function for a Type II and Type III ANOVA. For more information on this, see this description. For example, the output for a two-way ANOVA will be a table similar to the one as follows:

```
Analysis of Variance Table

Response: df.name$dependent.variable
                                        Sum Sq  Df F value  Pr(>F)
df.name$condition                        335.9   1  3.4005 0.06729 .
df.name$group                          29767.4   9 33.4822 < 2e-16 ***
Interaction                              338.0   9  0.3802 0.94299
Residuals                              13829.7 140
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This format is fairly simple to understand the overall significance of the ANOVA. For further analyses (such as post-hoc tests), there is a way to perform a Tukey HSD or pairwise t-tests. Additionally, you can analyze the linear model on these data.

1.5. Graphing

I will also include code to create plots, for example, a bar plot:

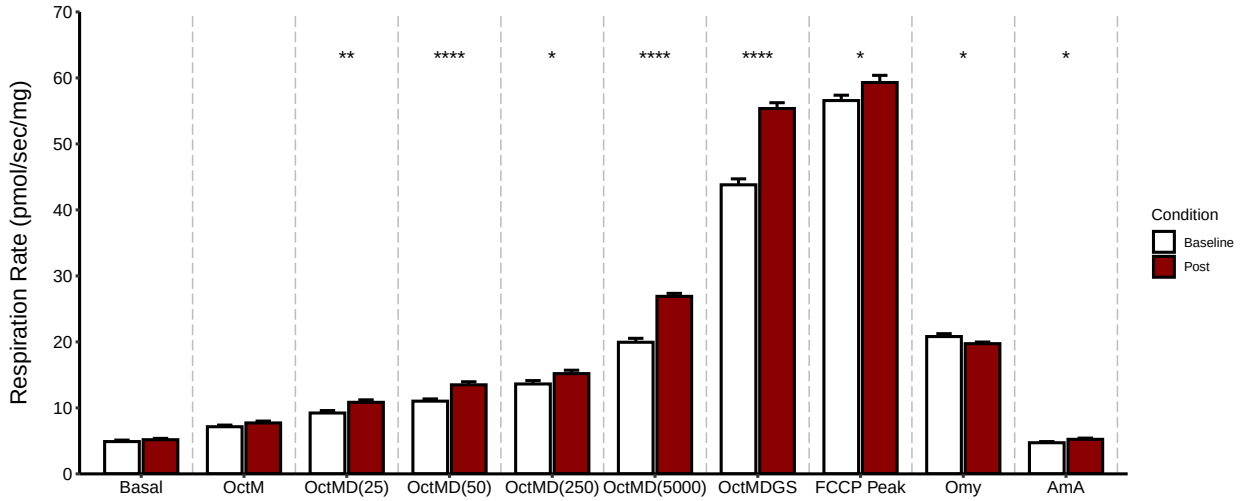


Figure 1: Sample representation of graphing respirometry data with bar graphs.

Or, we can make box plots like so:

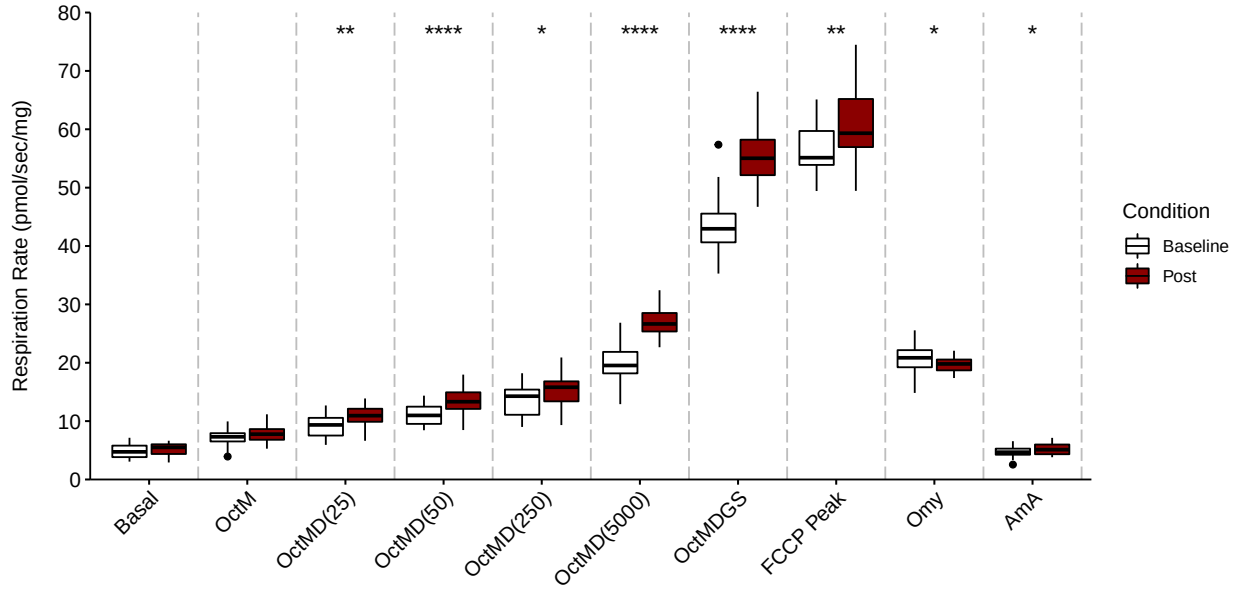


Figure 2: Sample representation of graphing respirometry data with boxplots.

My aim is to cover most of the basics of these functions and give out details to my code so that others can easily create graphical representations and reports of their own. However, as I am not quite an expert with all of this and I am learning as I go, I will not be able to cover everything. Hopefully, though, I have enough covered in this document to help guide most people through the process of analyzing and visualizing respirometry data

---



2. Preparing the Data

Overall, this process of running an analysis is pretty simple and straightforward. As I mentioned earlier, it is best to avoid running t-tests on data sets such as these because of the family-wise error problem. I'm not saying you shouldn't ever run t-tests, but it certainly makes sense to run an ANOVA (or better yet, a planned comparisons analysis) and then t-tests with appropriate post-hoc corrections.

2.1. Pulling the Data

I mentioned the steps for this process in the Pulling data from Excel section of this document. Since I mostly use Excel files for my processing and I have a fairly useful template file, I will stick with pulling the data from Excel. If you have other means of processing the data, feel free to skip this section. Last, since the data in the ANOVA tab is usually the only data that needs to be cleaned up, I will focus on this data. The data in the t-test tab usually doesn't need any cleanup, and any cleanup it needs will be done using the methods outlined in this section.

For this step, we will need to load the `'readxl'` package from our library by using the command below.[1]

```
library(readxl)
```

This simply tells R that you will be calling some of the functions within this package, which we will certainly do. Next, we run the code to extract the data we need from the excel sheet using the `'read_excel'` function. Notice that it important to use the full file directory when extracting the Excel file. We should use the following:

```
df.name <- read_excel("c\\complete_file_path\\file_name.xlsx",
      sheet = "Sheet Name", col_names = TRUE)
```

Or, something more realistic might be:

```
data_example_long <- read_excel(
  "C:\\Users\\Your_name\\Desktop\\Example_FFA_O2K_Analysis.xlsx",
  sheet = "Final Data ANOVA", col_names = TRUE)
```

This function will give us a data frame from the sheet we have selected (in this example the 'Final Data ANOVA' Sheet). If you have different names for sheets, you can change this by editing the `function sheet =` line. Simply replace the name in quotations with your own. However, it is important to maintain the quotations around the sheet name.

If we now begin to look to see what the data frame looks like by typing the name of the data frame (in this case `'data_example_long'`) we will get something that matches the data table we have from Excel. If we look at the first few rows:

[1] *If you haven't installed this or other packages, do so by using the `'install.packages()'` command; for example, `install.packages('readxl')` would be used to install this package.*

Table 1: First 10 rows of the data set.

Subject	Condition	Leg	Respiration State	Respiration Rate (pmol/sec/mg)
1	Baseline	Right	Basal	5.68
1	Baseline	Right	OctM	7.79
1	Baseline	Right	OctMD(25)	12.13
1	Baseline	Right	OctMD(50)	8.44
1	Baseline	Right	OctMD(250)	11.02
1	Baseline	Right	OctMD(5000)	23.79
1	Baseline	Right	OctMDGS	46.88
1	Baseline	Right	CytC	44.36
1	Baseline	Right	FCCP Peak	61.65
1	Baseline	Right	Omy	20.64

You'll notice in this data frame the columns have been identified and transferred into the data frame. This can be modified by the `col_names=` line – if you don't want to have the column names, simply change `TRUE` to `FALSE`. Furthermore, the data columns in this document all have random values. Other than that that, any other cells that have any sort of value in them will only appear as `"NA"`. Once missing numbers have been filled in, you will see those numbers in their respective column. We will get into some more of the details of the data frame later. If we want to use the t-test tab, simply change the code to:

```
data_example_wide <- read_excel(
  "C:\\Users\\Stephen\\Desktop\\O2K Analysis Files\\Example_FFA_O2K_Analysis.xlsx",
  sheet = "Final Data t-test", col_names = TRUE)
```

That should have all of the same information, just in a different layout, which we will discuss next.

2.1.1. Wide & Long Formatting

Something that may be useful to have in your arsenal is being able to change your data from wide format to long (otherwise known as narrow or vertical) format. ***Long format will be very useful when we graph our data, which is why I will give you ways to calculate all of your variables in long format as you will need to have the code in long format eventually.*** This is extremely critical, and some of the functions are just more intuitive to perform in long format so I don't always have the code done in wide format.[2] I define these terms as follows

- **Wide format** is the arrangement of data such that each variable is in its own column, and each row is assigned to only one subject.

[2]There are many different methods of converting between wide and long format, which I do not cover in this section. For more, see section 9.6

- **Long format**, on the other hand, is the arrangement of data such that one column contains all of the values of a variable and another column contains the context of this given value.

If we refer to our Excel sheet that I have provided, you will notice that the tab titled "Final Data t-test" is in wide format, where if I take the first 9 columns we should see the following:

Table 2: Example of Wide Format

Subject	Condition	Mass	Leg	Basal	OctM	OctMD(25)	OctMD(50)	OctMD(250)
1	Baseline	1.46	Right	5.68	7.79	12.13	8.44	11.02
2	Baseline	1.68	Left	7.15	6.65	7.07	12.23	14.85
3	Baseline	1.71	Right	4.73	6.66	10.76	12.41	12.63
4	Baseline	1.71	Left	4.63	6.20	8.91	9.07	14.49
5	Baseline	1.52	Right	5.19	8.10	7.50	12.59	17.51
6	Baseline	1.60	Left	5.79	6.47	10.06	10.98	11.16
7	Baseline	1.42	Right	5.30	7.93	10.51	10.24	14.36
8	Baseline	1.45	Left	4.59	7.57	10.44	13.12	14.84
9	Baseline	1.43	Right	3.87	6.16	7.54	10.12	14.26
10	Baseline	1.57	Left	3.65	6.90	5.93	10.19	9.00

Long format would look like the data we get in the ANOVA tab, where we have each column represented by some variable (such as 'Respiration State') and a value (such as 'Respiration Rate')

Table 3: Example of Long Format

Subject	Condition	Leg	Respiration State	Respiration Rate (pmol/sec/mg)
1	Baseline	Right	Basal	5.68
1	Baseline	Right	OctM	7.79
1	Baseline	Right	OctMD(25)	12.13
1	Baseline	Right	OctMD(50)	8.44
1	Baseline	Right	OctMD(250)	11.02
1	Baseline	Right	OctMD(5000)	23.79
1	Baseline	Right	OctMDGS	46.88
1	Baseline	Right	CytC	44.36
1	Baseline	Right	FCCP Peak	61.65
1	Baseline	Right	Omy	20.64
1	Baseline	Right	AmA	2.55
2	Baseline	Left	Basal	7.15
2	Baseline	Left	OctM	6.65
2	Baseline	Left	OctMD(25)	7.07
2	Baseline	Left	OctMD(50)	12.23



Sometimes it is more useful to look at the data in wide format, then convert it to long format for further analysis. For example, it would be fairly difficult to get a general idea of what the respiration rates are during Basal respiration if the data are in *long* format. However, *wide* format makes it much easier for us to see all of the data that we would want to compare. Furthermore, some analyses (such as t-tests) need to be done with data in wide format, while others (such as ANOVAs) need to be done in long format.

Although I have already created a basic Excel template that has a tab with the data in long format (which was quite tedious to do in Excel), it may be useful to understand this process in R as it is very simple to do and will save quite a bit of time. The process to go from wide to long format is as simple as:

```
Wide_to_Long <- gather(Wide, "Respiration State",
                           "Respiration Rate (pmol/sec/mg)",
                           Basal:AmA, factor_key = TRUE)
```

Gather is now considered out of date and is no longer updated by the Tidyverse. A more up to date function, `pivot_longer()` will also work:

```
Wide %>%
  pivot_longer(-c(Subject:Leg), names_to = "Respiration State", values_to = "Respiration
```

Both should give us an output of something like:

Table 4: Our New Table

Subject	Condition	Mass	Leg	Respiration State	Respiration Rate (pmol/sec/mg)
1	Baseline	1.46	Right	Basal	5.68
2	Baseline	1.68	Left	Basal	7.15
3	Baseline	1.71	Right	Basal	4.73
4	Baseline	1.71	Left	Basal	4.63
5	Baseline	1.52	Right	Basal	5.19
6	Baseline	1.60	Left	Basal	5.79
7	Baseline	1.42	Right	Basal	5.30
8	Baseline	1.45	Left	Basal	4.59
9	Baseline	1.43	Right	Basal	3.87
10	Baseline	1.57	Left	Basal	3.65
11	Baseline	1.58	Right	Basal	5.82
12	Baseline	1.50	Left	Basal	4.73
13	Baseline	1.67	Right	Basal	6.00
14	Baseline	1.34	Left	Basal	3.06
15	Baseline	1.47	Right	Basal	5.07

2.2. Cleaning the Data

Now that we have our data in R, we need to clean the data. This includes removing all 'NA' values and Cytochrome C (because we usually do not plot these data and only use it



as a quality control variable), and recoding the data into their proper data variable. These processes are much simpler to do than they sound..

2.2.1. Removing unwanted data

To remove the unwanted data in long format,[3] we will need to load the 'Tidyverse' package using the command `'library(tidyverse)'`. To remove all of the Cytochrome C and 'NA' values, we will use a couple of different commands. I find these commands to be fairly intuitive to follow. With the first command, we will just tell R to remove the Cytochrome C data from the document. Again, we only do this because we rarely report them in the actual data, but if you need them you can ignore that line of code. The second command is very important, as it will remove all 'NA' values from this sheet. This is extremely critical as you cannot run any statistical analyses with 'NA' values in the data frame.

```
#remove cytc and rows with NA (select number of rows to include)#

## Delete every 11th row starting from 8 to remove CytC
data_example_long <- data_example_long %>% dplyr::filter(
  row_number() %% 11 != 8)

##Remove all NA values##
data_example_long <- na.omit(data_example_long)
```

The first command uses `'dplyr'` to remove every nth row starting at row x. If we look at the way I have organized the Excel sheet, we will notice that a value for Thus, in this case, we have chosen to remove every 11th row starting at row number 8. If you have different rows for the cytochrome c values, you can modify this accordingly.
The second piece of code here uses the `'na.omit()'` function to remove all `'NA'` values in the data frame. If we run the code now, we should get:

Table 5: First Few Rows of the Data with CytC Removed

Subject	Condition	Leg	Respiration State	Respiration Rate (pmol/sec/mg)
1	Baseline	Right	Basal	5.679875
1	Baseline	Right	OctM	7.785628
1	Baseline	Right	OctMD(25)	12.126744
1	Baseline	Right	OctMD(50)	8.437867
1	Baseline	Right	OctMD(250)	11.016722
1	Baseline	Right	OctMD(5000)	23.786752

This new line now has all of the CytC and NA values removed, just like we wanted!

[3] *I recommend doing this step in long format, as the na.omit or na.rm functions will remove entire rows of data, which is not very desirable in most situations. A partial data set is sometimes better than a null data set.*

2.2.2. Recoding the data as correct data variables

Now that we have all of our "unimportant" data removed from our data frame, we need to gather basic information about our data, namely the type of data variables we have — such as characters (chr), numeric (num), logicals(logi), integers (int) and factors. These are important to identify as each of these describes how R will handle the data during an analysis. For example, we can make groups by identifying some data as 'Factors'. When we pull our data from Excel into R, R cannot identify these without us specifically identifying each type of data. We would not be able to run any analysis without first doing this because R would label them all as characters or something that wouldn't be of use to us. Thus, To show that our data are, in fact, not identified as correct variables, we can use the ' `glimpse()'` function (or we can similarly use the `'str()'` function) as below:

```
#Take a look at the data
glimpse(data_example_long)
```

```
## Rows: 540
## Columns: 5
## $ Subject                          <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2...
## $ Condition                        <chr> "Baseline", "Baseline", "Baseline"...
## $ Leg                              <chr> "Right", "Right", "Right", "Right"...
## $ `Respiration State`              <chr> "Basal", "OctM", "OctMD(25)", "Oct...
## $ `Respiration Rate (pmol/sec/mg)` <dbl> 5.679875, 7.785628, 12.126744, 8.4...
```

It's clear here that R does not identify these variables as the correct data variables. For example, we would ideally want the variables "Subject", "Condition", "Leg", and "Respiration State" as 'Factors' because these are variables we use to identify certain characteristics such as the specific subject, the baseline or post condition, which leg we used, and the type of state the sample is placed in. Likewise, "Respiration Rate" should be a numeric variable (which it is already, but I will run through the code anyway). Therefore, we should probably attempt to change these into usable variables for our later analyses. Basically, we will simply recode these using functions such as `'as.factor'` and `'as.numeric'`.

```
#Recode Subjects as factors#
data_example_long$Subject <- as.factor(data_example_long$Subject)

#Recode Condition as factors#
data_example_long$Condition <- as.factor(data_example_long$Condition)

#Recode Leg as factors#
data_example_long$Leg <- factor(data_example_long$Leg)

#Recode Respiration States as factors#
data_example_long$`Respiration State` <- factor(
  data_example_long$`Respiration State`, levels =
    c("Basal", "OctM", "OctMD(25)", "OctMD(50)", "OctMD(250)",
      "OctMD(5000)", "OctMDGS", "FCCP Peak", "Omy", "AmA"))
```

```
#Recode Respiration Rate as numeric#
data_example_long$`Respiration Rate (pmol/sec/mg)` <- as.numeric(
        data_example_long$`Respiration Rate (pmol/sec/mg)`)
```

Notice that with the 'Respiration State' factor, we must identify the order of levels so that we can properly order these data in our graphs. To check if these data are properly identified, we will rerun the 'glimpse()' function:

```
#Take a look at the data
glimpse(data_example_long)
```

```
## Rows: 540
## Columns: 5
## $ Subject                          <fct> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2...
## $ Condition                        <fct> Baseline, Baseline, Baseline, Base...
## $ Leg                              <fct> Right, Right, Right, Right, Right,...
## $ `Respiration State`              <fct> Basal, OctM, OctMD(25), OctMD(50),...
## $ `Respiration Rate (pmol/sec/mg)` <dbl> 5.679875, 7.785628, 12.126744, 8.4...
```

2.2.3. Renaming variables

If you ever need to rename variables, which happens from time to time, you can do so quite simply. If we wanted to rename 'FCCP Peak' to just 'FCCP' in wide format, we would do:

```
#Rename "FCCP Peak" to "FCCP"
names(data_example_wide)[names(data_example_wide)=="FCCP Peak"] <- "FCCP"
```

Or with the long format:

```
#Rename FCCP Peak to FCCP
gsub("FCCP Peak", "FCCP", data_example_long$`Respiration State`)
```

Now that it looks like all of the data are properly structured, we can continue with the rest of the analysis.



3. Glancing at Descriptive Data

First, I think it would be good to discuss how to get descriptive data, since they are always reported in manuscripts. Overall, this is going to be an easy process, and I will detail how to get the descriptive data in both wide ad long format.

3.1. Descriptives in Wide Format (for t-tests)

First, we must load the data in wide format from the 'Final Data t-test' tab in the Excel file, like so:

```
#load readxl
library(readxl)

# Put the excel file into R
data_example_wide <- read_excel(
  "C:\\Users\\Stephen\\Desktop\\O2K Analysis Files\\Example_FFA_O2K_Analysis.xlsx",
  sheet = "Final Data t-test", col_names = TRUE)
```

And we should see something like this:

Table 6: The First 10 Rows and the First 8 Columns of the T-test Data (Wide Format)

Subject	Condition	Mass	Leg	Basal	OctM	OctMD(25)	OctMD(50)
1	Baseline	1.455888	Right	5.679875	7.785628	12.126744	8.437867
2	Baseline	1.676108	Left	7.146072	6.646580	7.072031	12.233790
3	Baseline	1.713945	Right	4.732141	6.660655	10.756599	12.407065
4	Baseline	1.706729	Left	4.631992	6.200051	8.911402	9.069393
5	Baseline	1.520053	Right	5.188824	8.104829	7.499916	12.585101
6	Baseline	1.599761	Left	5.788582	6.472204	10.059021	10.975784
7	Baseline	1.421302	Right	5.298100	7.933290	10.511174	10.244510
8	Baseline	1.447499	Left	4.593373	7.574408	10.441080	13.121400
9	Baseline	1.434053	Right	3.872630	6.164325	7.544475	10.123786
10	Baseline	1.574042	Left	3.648194	6.895149	5.931801	10.186585

Next, we will use the `'describe.by'` function in the `'psych'` package.

```
#Load the psych package
library(psych)

#Calculate descriptive data
stat_summary_ttest <- describe.by(data_example_wide[5:15],
                        data_example_wide$Condition)
```

This will actually give us two (or more, if you have more groups) tables. To extract the tables



individually to look at clean data, we only need to call each condition we have. For example, if I want to look at the baseline condition, I only need to type `'stat_summary_ttest$Baseline'` and the data will be shown as follows:

```
#Separate baseline and post data
baseline_desc <- stat_summary_ttest$Baseline
post_desc <- stat_summary_ttest$Post
```

When we look at the data, we should see a lot of information. Mean, SD, skew, SE, etc. which should be about all of the descriptive information we need. Below is an abbreviated table (only mean, SD, median, and SE) with the baseline information:

Table 7: Baseline Descriptives

	mean	sd	median	se
Basal	4.893607	1.1531528	4.733414	0.2219244
OctM	7.144976	1.2789729	7.313233	0.2461384
OctMD(25)	9.223119	1.9157650	9.348294	0.3686891
OctMD(50)	11.020198	1.7404544	10.975784	0.3349506
OctMD(250)	13.623735	2.6520177	14.270068	0.5103811
OctMD(5000)	19.945661	3.0842134	19.533835	0.5935571
OctMDGS	43.803213	4.6850011	42.953471	0.9016289
CytC	44.623806	5.2023702	44.290524	1.0011966
FCCP Peak	56.572665	4.2149754	55.134923	0.8111724
Omy	20.808519	2.2755662	20.848535	0.4379329
AmA	4.718785	0.8961109	4.680589	0.1724566

And the post intervention information:

Table 8: Post Intervention Descriptives

	mean	sd	median	se
Basal	5.176722	1.0458081	5.505306	0.2012659
OctM	7.719175	1.4596106	7.763718	0.2809022
OctMD(25)	10.844400	1.9333035	10.935271	0.3720644
OctMD(50)	13.494246	2.4060032	13.322002	0.4630355
OctMD(250)	15.192369	2.6764675	15.805796	0.5150864
OctMD(5000)	26.898550	2.3593788	26.648326	0.4540627
OctMDGS	55.364371	4.5692324	55.032976	0.8793492
CytC	55.629002	5.8478894	55.849814	1.1254268
FCCP Peak	60.820730	6.6443178	59.324883	1.2786996
Omy	19.731467	1.2764339	19.797602	0.2456498
AmA	5.235792	0.9457705	5.108662	0.1820136

3.2. Descriptives in Long Format (for ANOVAs)

Just like I did with the t-test, I will first go over the process to get descriptive data. Now, for this step your data should be in long format, which you will need for the main part of the ANOVA anyway (this data format is the same that I used in the preparing section). We can get all of the same descriptive information as we did in the previous section by doing the following:

```
#Calculate descriptive data
by(data_example_long$`Respiration Rate (pmol/sec/mg)`,
   list(data_example_long$`Respiration State`,
        data_example_long$Condition), stat.desc, basic = FALSE)
```

This output is very large, so I won't include it in this document, but it does contain all of the same information as the descriptive data we did in the t-test section, so you can choose whatever method is easiest (I prefer the nice tables, so I normally use the other way).



4. Testing for the Assumptions of Normality & Homogeneity of Variance

Now, if we remember from our statistics class (or possibly not), we should recognize that the t-test and ANOVA both have certain assumptions in order to be properly run. These are:

- Independence of cases
- Normality
- Homogeneity of Variance (or in the case of an ANOVA, sphericity)

If you want to read more on this, you can do so here. If either of the last two of these are violated, we must use a non-parametric test such as a Wilcoxon test (paired samples t-test), Mann-Whitney test (for independent sample t-tests), Kruskal-Wallis test (for one-way or two-way ANOVA), or Friedman test (for repeated-measures ANOVA) instead(more on this in the t-test section). As I did in the Preparing the Data section, I will use the ANOVA table for simplicity. It seems like running these tests in long format is smoother than running it in wide format, however, you can use wide format if you want to use the `'aggregate'` and `'merge'` functions.

4.1. Testing for Normality

In order to determine the Normality via the Shapiro-Wilk test, we will need to run something like this:

```
#Shapiro Test for Normality#

#Load the data.table package
library(data.table)

#place the data frame into the data.table package format
DT <- data.table(data_example_long)

#perform the Shapiro Test with the new data
shapiro_results <- DT[,
   .(Statistic = shapiro.test(
     `Respiration Rate (pmol/sec/mg)`)$statistic,
     P.value = shapiro.test(
       `Respiration Rate (pmol/sec/mg)`)$p.value),
   by = .(`Respiration State`)]
```

Table 9: Normality Test Results

Respiration State	Statistic	P.value
Basal	0.9649362	0.1147109
OctM	0.9870648	0.8249723
OctMD(25)	0.9700893	0.1945292

Respiration State	Statistic	P.value
OctMD(50)	0.9604061	0.0718500
OctMD(250)	0.9829703	0.6350766
OctMD(5000)	0.9731771	0.2650343
OctMDGS	0.9719146	0.2337858
FCCP Peak	0.9582979	0.0578264
Omy	0.9756706	0.3376538
AmA	0.9900528	0.9328270

From here we now have the table:

Factor A	Statistic	P-value
Factor A1	Statistic A1	P-value A1
Factor A2	Statistic A2	P-value A2
Factor A3	Statistic A3	P-value A3
Factor A4	Statistic A4	P-value A4
Factor A5	Statistic A5	P-value A5
...	...	...

Where the first column is the factor ('Respiration State' in our data) and the second column is the p-value of the Shapiro test. Furthermore, if we have more than one grouping factor (as we do in this case), we can simply adjust the code by adding a `'+ `Factor2`'` like below:

```
#Perform the Shapiro test
shapiro_results2 <- DT[,
   .(Statistic = shapiro.test(
     `Respiration Rate (pmol/sec/mg)`)$statistic,
     P.value = shapiro.test(
       `Respiration Rate (pmol/sec/mg)`)$p.value),
   by = .(`Respiration State`, `Condition`)]
```

Table 10: Normality Test Results by Group

Respiration State	Condition	Statistic	P.value
Basal	Baseline	0.9641379	0.4567614
OctM	Baseline	0.9594502	0.3590692
OctMD(25)	Baseline	0.9515083	0.2331959
OctMD(50)	Baseline	0.9393454	0.1173986
OctMD(250)	Baseline	0.9552108	0.2859651
OctMD(5000)	Baseline	0.9757437	0.7563448
OctMDGS	Baseline	0.9408292	0.1277240
FCCP Peak	Baseline	0.9675981	0.5397602
Omy	Baseline	0.9752476	0.7431239
AmA	Baseline	0.9890982	0.9902912



Respiration State	Condition	Statistic	P.value
Basal	Post	0.9255792	0.0538902
OctM	Post	0.9676455	0.5409519
OctMD(25)	Post	0.9633710	0.4395693
OctMD(50)	Post	0.9705432	0.6162230
OctMD(250)	Post	0.9910273	0.9969041
OctMD(5000)	Post	0.9819007	0.9025226
OctMDGS	Post	0.9704261	0.6131064
FCCP Peak	Post	0.9727920	0.6767770
Omy	Post	0.9756295	0.7533113
AmA	Post	0.9558361	0.2958650

And the output:

Factor A	Factor B	Statistic	P-value
Factor A1	Factor B1	Statistic A1B1	P-value A1B1
Factor A2	Factor B1	Statistic A2B1	P-value A2B1
...	...	...	...
Factor A1	Factor B2	Statistic A1B2	P-value A1B2
Factor A2	Factor B2	Statistic A2B2	P-value A2B2
...	...	...	...
Factor A1	Factor B3	Statistic A1B3	P-value A1B3
Factor A2	Factor B3	Statistic A2B3	P-value A2B3
...	...	...	...
...	...	...	...

Where the first column is the first factor ('Condition' in our data), the second column is the second factor ('Respiration State' in our data), and the third column is the p-value of the Shapiro test. And so on.

Now, once we have this data, we can determine if the normality assumption has been violated by looking at the p-values. If any of the p-values are significant (as we see in this data), we must assume that the assumption of normality has been violated and you cannot use an ANOVA for this test (instead use a non-parametric test).

I will also note that the following function may be used to get the same results. You can choose either one, but I found the previous code to also work with the tests for Variance below. I have only included it for reference.

```
#Load rstatix
library(rstatix)

shapiro_aggregate <- aggregate(formula = `Respiration Rate (pmol/sec/mg)` ~
            `Respiration State` + `Condition`,
          data = data_example_long,
          FUN = function(x) {y <- shapiro.test(x); c(y$p.value)})
```

Table 11: Normality Test Results Using the 'aggregate()'

Respiration State	Condition	Respiration Rate (pmol/sec/mg)
Basal	Baseline	0.4567614
OctM	Baseline	0.3590692
OctMD(25)	Baseline	0.2331959
OctMD(50)	Baseline	0.1173986
OctMD(250)	Baseline	0.2859651
OctMD(5000)	Baseline	0.7563448
OctMDGS	Baseline	0.1277240
FCCP Peak	Baseline	0.5397602
Omy	Baseline	0.7431239
AmA	Baseline	0.9902912
Basal	Post	0.0538902
OctM	Post	0.5409519
OctMD(25)	Post	0.4395693
OctMD(50)	Post	0.6162230
OctMD(250)	Post	0.9969041
OctMD(5000)	Post	0.9025226
OctMDGS	Post	0.6131064
FCCP Peak	Post	0.6767770
Omy	Post	0.7533113
AmA	Post	0.2958650

4.2. Testing for Variance

For testing the homogeneity of variance among the samples using the F-test, we will use this line of code similar to the one above:

```
#Perform the F-test
variance_results <- DT[,
   .(Statistic = var.test(
     `Respiration Rate (pmol/sec/mg)` ~ `Condition`)$statistic,
     P.value = var.test(
       `Respiration Rate (pmol/sec/mg)`~ `Condition`)$p.value),
   by = .(`Respiration State`)]
```

Table 12: Variance Test Results

Respiration State	Statistic	P.value
Basal	1.2158213	0.6219630
OctM	0.7678010	0.5052567
OctMD(25)	0.9819387	0.9632916
OctMD(50)	0.5232785	0.1049008

Respiration State	Statistic	P.value
OctMD(250)	0.9818132	0.9630344
OctMD(5000)	1.7088089	0.1786045
OctMDGS	1.0513151	0.8994509
FCCP Peak	0.4024290	0.0237654
Omy	3.1782079	0.0044433
AmA	0.8977431	0.7853739

Or we can use Levene's test[4], which is more robust to deviations of normality, as follows:

```
#Load rstatix
library(rstatix)

#Perform Levene's test
levene_results <- DT %>%
  group_by(`Respiration State`) %>%
  levene_test(`Respiration Rate (pmol/sec/mg)` ~ Condition)
```

Table 13: Levene's Test Results

Respiration State	df1	df2	statistic	p
Basal	1	52	0.2233489	0.6384776
OctM	1	52	0.3766363	0.5420844
OctMD(25)	1	52	0.1030712	0.7494616
OctMD(50)	1	52	1.4899482	0.2277308
OctMD(250)	1	52	0.0075615	0.9310396
OctMD(5000)	1	52	1.4272821	0.2376293
OctMDGS	1	52	0.0583857	0.8100163
FCCP Peak	1	52	2.8493122	0.0974028
Omy	1	52	4.2163552	0.0450815
AmA	1	52	0.4006953	0.5295045

We only need to run this code to compare the two conditions (Baseline vs Post) within each Respiration State because the goal of the F-statistic is to make sure that the variance between two variables is equal. Since, in this case, the two variables we would compare would be the Baseline and Post values, we only need to compare those.

Ideally, we should not see any significant values in our tests we have run above. Though, as you can see, we have a few variables that have kicked back a couple of significant p-values. There are a few options, in my opinion, that can be done at this point:

- You can perform a non-parametric test. This can be done, however, you will most likely

[4] *I'm not sure why I can't get all of these functions to work under one package, which is why I have all of these options to make the same code. Use what works best for you.*

lose some power in your further analyses.
- You can ignore the significant values. There are several reasons for coming to this conclusion based on these data, but the biggest reason is that we have performed several comparisons, and thus have accumulated a greater chance of a type-I error, or, a false positive. If we consider that the probability of a variable becoming significant due to random chance increases as we perform more comparisons, then we must consider that some of these p-values must be subject to the same randomness. Therefore, if we apply a correction for this randomness (such as a Bonferroni or Tukey correction), our values are clearly no longer significant.

I would argue for the second case in these random data I have created for the sake of this manuscript (and that the data that have been identified are not of extreme importance to these data, but that itself is a weak argument). However, I strongly encourage each individual data set be properly analyzed for these parameters.

4.3. Further Considerations

It is important to acknowledge the limitations above. First, I am assuming you will only be comparing 2 factors (Baseline vs Post), rather than multiple factors. If you wish to compare more than 2 factors (e.g. Young, Middle-Age, and Elderly), you can perform Levene's test like above (though I will add I haven't tested the code yet, but it may be done in one of my projects soon).

4.3.1. Borwn-Forsythe Test

Furthermore, if you wish to change the center of the Levene Test from 'mean' (the default parameter) to 'median' (also known as a Brown-Forsythe Test), simply add `center = median` to the code as follows:

```
library(rstatix)

BrownForsythe_results <- DT %>%
  group_by(`Respiration State`) %>%
  levene_test(`Respiration Rate (pmol/sec/mg)` ~ Condition, center = median)
```

Table 14: Brown-Forsythe Test Results

Respiration State	df1	df2	statistic	p
Basal	1	52	0.2233489	0.6384776
OctM	1	52	0.3766363	0.5420844
OctMD(25)	1	52	0.1030712	0.7494616
OctMD(50)	1	52	1.4899482	0.2277308
OctMD(250)	1	52	0.0075615	0.9310396
OctMD(5000)	1	52	1.4272821	0.2376293
OctMDGS	1	52	0.0583857	0.8100163
FCCP Peak	1	52	2.8493122	0.0974028



Respiration State	df1	df2	statistic	p
Omy	1	52	4.2163552	0.0450815
AmA	1	52	0.4006953	0.5295045

4.3.2. Mauchly's Test for Sphericity

Similarly, if you are running a repeated-measures ANOVA (discussed later), you should use Mauchly's test in the as follows:

```
#Load the package ez
library('ez')

#Set factor levels for Respiration State
DT$State<- factor(
  data_example_long$`Respiration State`, levels =
    c("Basal", "OctM", "OctMD(25)", "OctMD(50)", "OctMD(250)",
      "OctMD(5000)", "OctMDGS", "FCCP Peak", "Omy", "AmA"))

#Recode Respiration Rate as numeric#
DT$Rate <- as.numeric(
        data_example_long$`Respiration Rate (pmol/sec/mg)`)

#Run the ANOVA to get Mauchly's Test
ezANOVA(data = DT, dv = .(Rate), wid = .(Subject), within = .(State),
        between = .(Condition), detailed = TRUE, type = 3)
```

Please noice that in order to get this done, I had to recode the variables into single word variables (`State` and `Rate` from `Respiration State` and `Respiration Rate (pmol/sec/mg)`, respectively). I don't know why the code won't recognize the longer strings, but this is the fix.

4.3.3. Fligner-Killeen Test

Lastly, if your data are not normally distributed, you cannot perform an F-test or Levene's test; instead you must perform the Fligner-Killeen test as follows:

```
FlingerKilleen_results <- DT[,
   .(Statistic = fligner.test(
     `Respiration Rate (pmol/sec/mg)` ~ `Condition`)$statistic,
     P.value = fligner.test(
       `Respiration Rate (pmol/sec/mg)`~ `Condition`)$p.value),
   by = .(`Respiration State`)]
```

Table 15: Fligner-Killeen Test Results

Respiration State	Statistic	P.value
Basal	0.1613565	0.6879107
OctM	0.5288571	0.4670878
OctMD(25)	0.0740582	0.7855172
OctMD(50)	1.2885977	0.2563065
OctMD(250)	0.0267515	0.8700784
OctMD(5000)	1.2009659	0.2731287
OctMDGS	0.2079152	0.6484064
FCCP Peak	2.6304685	0.1048308
Omy	3.3377431	0.0677074
AmA	0.2611531	0.6093291

This should be about all of the tests you will need to properly test assumptions for your data. Once you determine the proper test, we can move on to further analysis.



5. Hypothesis Testing

Now that we have looked into the assumptions, we can move on to the next step: running comparisons. These should all be fairly straightforward, but can get tricky depending on what test you want to perform. Overall, doing a basic t-test and ANOVA will be simple, but things will get more complex if you wish to do a planned comparison ANOVA or something a little more complex than the basic tests (I will hopefully try to cover that at some point). For now, let's dive into the t-test.

5.1. Hypothesis Testing with Two Variables

5.1.1. The t-test

The t-test analysis is fairly easy to run.[5] Lucky for us, R comes with a t-test function and an output that is easy to understand:

```
t.test(y ~ x)
```

Where y is a numeric value and x is a binary (group) value. Pretty simple, right? You can also do:

```
t.test(y1,y2)
```

Where each y is a numeric value. For paired samples (data with the same subjects tested twice, as in a pre- & post-measurement), you simply need to add a `'paired = TRUE'` line like this:

```
t.test(y ~ x, paired = TRUE)
```

Now, since we are going to run multiple paired t-tests (remember this is paired data) to analyze each respiration state, we can simply use the `'lapply'` and `'sapply'` functions in the `'tidyr'` package, and we will place this information in new data frames so we can have a much simpler time looking at the data as so:

```
#load dplyr
library(dplyr)

# Run multiple paired t-tests
FFA_ttest_values <- lapply(data_example_wide[,5:15],
  function(x) t.test(x ~ data_example_wide$Condition,
  paired = TRUE, na.rm = TRUE))

# Place p-values into columns
FFA_ttest_pvalues <- data.frame("P.value" = sapply(
  FFA_ttest_values, getElement, name = "p.value"))
```

[5] *It's best to make sure that all of your variables are identified properly (e.g. as integers, characters, etc.) before running the t-test, but isn't always necessary since R should identify them properly. Still, you will need to do this for the ANOVA.*

The 'lapply' function serves as a sort of loop that takes the specified columns (in this case, columns 5-15) and runs the function of choice (in this case, the t-test). The '~' symbol, when using it in a function of, can be thought to mean "as a function of." Essentially, we are taking columns 5-15 of our data, and applying a t-test to the data in every column 'x' as a function of the Condition. These are paired samples, and we want to remove all 'NA' data points. Furthermore, we have placed the analysis in a new data frame 'FFA_ttest_values,' which is slightly messy. To clean this up, we will take the data frame 'FFA_ttest_values' and extract the p-values using 'sapply' and place them in the new data frame 'FFA_ttest_pvalues.'[6] The output will give us the new data frame 'FFA_ttest_pvalues' which should look something like:

Table 16: A Table of the P-Values from the T-test

	P.value
Basal	0.3515074
OctM	0.1078474
OctMD(25)	0.0074443
OctMD(50)	0.0006206
OctMD(250)	0.0406295
OctMD(5000)	0.0000000
OctMDGS	0.0000000
CytC	0.0000000
FCCP Peak	0.0067060
Omy	0.0480655
AmA	0.0384915

Short and sweet, if you ask me. Really, you don't need to create a data frame with only the p-values, but I find that this really helps keep the data organized and put into place. If you want to do a correction for multiple comparisons, the use the following:

```
#Correct for multiple comparisons
FFA_ttest_pvalues$`Adjusted p` <- p.adjust(FFA_ttest_pvalues$P.value,
  method = "holm")
```

I have chosen to use the Holm adjustment, which will adjust for the p value and give us:

Table 17: Adjusted P-Values Using the Holm Method

	P.value	Adjusted p
Basal	0.3515074	0.3515074
OctM	0.1078474	0.2156948
OctMD(25)	0.0074443	0.0469418

[6] *If you want to extract other variables with 'sapply' simply change the two "p.value" characters to the column name you want to extract (such as confidence interval, standard error, etc.).*

	P.value	Adjusted p
OctMD(50)	0.0006206	0.0049645
OctMD(250)	0.0406295	0.1924574
OctMD(5000)	0.0000000	0.0000001
OctMDGS	0.0000000	0.0000000
CytC	0.0000000	0.0000000
FCCP Peak	0.0067060	0.0469418
Omy	0.0480655	0.1924574
AmA	0.0384915	0.1924574

For more information on this code or for other adjustments that can be made, check out the R Companion book (also in the book section below).

5.1.2. Non-parametric tests for two variables

Below are non-parametric tests that can be used to analyze data that violate the assumptions discussed prior.

Mann-Whitney U Tests

The Mann-Whitney U test (also known as the Wilcoxon rank-sum test, not to be confused with the Wilcoxon signed-rank test discussed next) is the non-parametric equivalent of an independent samples t-test. This test follows the basic code outline of the t-test, instead you will use:

```
wilcox.test()
```

So, following our example above, we would do:

```
# Run multiple Mann-Whitney tests
FFA_mannwhit_values <- lapply(data_example_wide[,5:15],
  function(x) wilcox.test(x ~ data_example_wide$Condition,
  na.rm = TRUE))

# Place p-values into columns
FFA_mannwhit_pvalues <- data.frame("P value" = sapply(
  FFA_mannwhit_values, getElement, name = "p.value"))
```

And here, we see:

Table 18: P-Values from the Mann-Whitney Test

	P.value
Basal	0.3019111
OctM	0.1471977
OctMD(25)	0.0039626

	P.value
OctMD(50)	0.0002192
OctMD(250)	0.0460350
OctMD(5000)	0.0000000
OctMDGS	0.0000000
CytC	0.0000000
FCCP Peak	0.0215278
Omy	0.0259570
AmA	0.0868286

Wilcoxon Signed-Rank Test

The Wilcoxon Signed-Rank Test is the non-parametric equivalent of the paired t-test. As we did with the t-test, all we need to do to make this happen is add `'paired = TRUE'` to the Mann-Whitney code above, like so:

```
# Run multiple Mann-Whitney tests
FFA_wilcoxon_values <- lapply(data_example_wide[,5:15],
  function(x) wilcox.test(x ~ data_example_wide$Condition,
  paired = TRUE, na.rm = TRUE))

# Place p-values into columns
FFA_wilcoxon_pvalues <- data.frame("P value" = sapply(
  FFA_wilcoxon_values, getElement, name = "p.value"))
```

And here, we see:

Table 19: P-Values from the Wilcoxon Test

	P.value
Basal	0.3483276
OctM	0.1775200
OctMD(25)	0.0088885
OctMD(50)	0.0006678
OctMD(250)	0.0299050
OctMD(5000)	0.0000003
OctMDGS	0.0000000
CytC	0.0000003
FCCP Peak	0.0140057
Omy	0.0130108
AmA	0.0462626



5.2. Hypothesis Testing with More Than Two Variables

Now, as I mentioned before, I think the ANOVA is the most appropriate test to run for these data. To me, it is pretty clear you would want an F-value to control for Type I (false positive) errors by performing too many tests. As shown above, we would be performing 10 different t-tests. If we assume that $FWE \leq 1 - (1 - \alpha_{IT})^c$, where α_{IT} is the alpha level of a given test and c is the number of comparisons, we would get the following $FWE \leq 1 - (1 - 0.05)^{10}$ which gives us a final value of 0.401. **This means that the probability of a type I error is 0.401, or just over 40%.** I think it would go without saying that this would be a very high chance of a type I error in these data, hence the need for an ANOVA in our analysis if we want to maintain confidence in our results.

5.2.1. The ANOVA

I'm going to preface this analysis by saying that, under normal circumstances, the ANOVA should be performed using independent samples; meaning the ANOVA is an extension of the independent samples t-test and the subjects being compared should not be the same in any two groups (this is covered in the next section). The point of this statement is to mention that the example data I will use in this section should be analyzed via a repeated-measures ANOVA,[7] not the traditional ANOVA. Therefore, this section is meant to merely demonstrate the basics of how to run a traditional ANOVA.

If you remember, I briefly covered the code in a previous section, but I will go into more details here. Simply put, the ANOVA analysis has two steps:

1. Create a linear model of the data
2. Run the ANOVA on the linear model

While there is the second step, it isn't hard to implement in this process and is extremely straightforward to code. First, though, your data ***must be in long format.*** I cannot stress this enough, simply because this is how most other statistical software does ANOVAs and R is no different in that respect. Fortunately, the data in the ANOVA tab within the Excel file I have created is in long format. However, if you are using your own set of data and you need to do this or you need more explanation on what I mean by long and wide format, you can read more about it here.

By now, you've hopefully identified all of your variables properly in their proper columns. If you haven't, please do so first before running the ANOVA. Once we have that, we should have a table that looks something like this:

[7]*I am starting off with the repeated-measures ANOVA instead of a one-way ANOVA because the code is essentially the same, as I explained earlier and gives the same result. Furthermore, it is more likely that a factorial ANOVA, rather than a one-way ANOVA, will be run on data like these.



Table 20: The First 15 Rows of Data in Long Format for the ANOVA

Subject	Condition	Leg	Respiration State	Respiration Rate (pmol/sec/mg)
1	Baseline	Right	Basal	5.679875
1	Baseline	Right	OctM	7.785628
1	Baseline	Right	OctMD(25)	12.126744
1	Baseline	Right	OctMD(50)	8.437867
1	Baseline	Right	OctMD(250)	11.016722
1	Baseline	Right	OctMD(5000)	23.786752
1	Baseline	Right	OctMDGS	46.881066
1	Baseline	Right	FCCP Peak	61.645046
1	Baseline	Right	Omy	20.643978
1	Baseline	Right	AmA	2.554508
2	Baseline	Left	Basal	7.146072
2	Baseline	Left	OctM	6.646580
2	Baseline	Left	OctMD(25)	7.072031
2	Baseline	Left	OctMD(50)	12.233790
2	Baseline	Left	OctMD(250)	14.851221

Now, as I mentioned before, we need to first create a linear model of the data using the `'aov()'` function, then we can analyze the linear model using the `'anova()'` function. Let's start by performing an ANOVA examining the effect of the Condition (baseline vs post intervention) and Respiration State on Respiration Rate,[8] as so:

```
#Create the linear model
ANOVA_model<- aov(`Respiration Rate (pmol/sec/mg)` ~
        Condition + `Respiration State`, data = data_example_long, type = "II")

#Analyze the linear model
ANOVA_table <- as.data.frame(anova(ANOVA_model))
```

For context, We should see an output similar to something like this:

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Condition	1	1113.79	1113.79	97.37	0
`Respiration State`	9	171655.49	19072.83	1667.32	0
Residuals	529	6051.36	11.44	NA	NA

This table is fairly intuitive to grasp.[9] In the columns, you have your degrees of freedom,

[8] *This is just an example for reference. Normally, since we are comparing the same subjects over time, we should be using a repeated-measures or a mixed design ANOVA, which will be covered in the next section.*

[9] *This is the way to analyze a two-way (or factorial) ANOVA, which is what we need to do for these data.*

sum of squares, mean squares, F values, and lastly p-values. The rows represent the values for Factor A, Factor B, (in this case, Condition and Respiration State, respectively)„ and the Residuals. You'll notice that in my example, the p-values are labeled as 0. This is because the p values are so low (probably due to the way I have arranged the random data I created from thin air) that the tables just rounded them to 0. If I inspect the column by itself, the p-values are actually:

```
##                          Pr(>F)
## Condition              < 2.2e-16 ***
## `Respiration State`    < 2.2e-16 ***
## Residuals
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Based on this analysis, we can see that our results are indeed very significant. It appears that there is a 'Condition' effect, meaning that the post-intervention had some effect on our subjects. Personally, I don't care much about the p-value from the Respiration State effect, since we would expect respiration rates to be different when we add different substrates. I didn't include an interaction in this analysis because it wouldn't add any meaning to the data, but if you wanted to do so, simply change the `'+'` to a `'*'` in the equation like so:

```
aov(`Respiration Rate (pmol/sec/mg)` ~
        Condition * `Respiration State`, data = data_example_long, type = "II")
```

Lastly, I should mention the ability to run different types of sum of squares analyses in these functions. There are 3 types of sum of squares analyses (more on that here):

1. **Type I Sum of Squares** performs the sum of squares analysis in sequential order, assigning a maximum value to Factor A, then the remaining variation to Factor B, then the interaction (if present), then the residual. In other words, this test gradually adds in factors with each analysis. Annotated, it looks like this:
   SS(A) for Factor A.
   SS(B | A) for Factor B.
   SS(AB | A, B) for interaction AB.
   In this case, the ordering of the model makes a big difference in the result. In this case, it is unwise to use this unless the main effects are all completely independent of each other, which is not the case for our data.

2. **Type II Sum of Squares** is not sequential like the Type I Sum of Squares, but it also does not take the interaction into effect and tests each main effect after the other. Annotated, it looks like:
   SS(A | B) for Factor A.
   SS(B | A) for Factor B.
   This type or analysis is beneficial if you care most about the main effects (which we do here) rather than the interaction effects, or if there is no interaction effect.

If you want to do a one-way ANOVA for any reason, simply exclude the `'+ FactorB'` part (in this case "`'+Respiration State`").

3. **Type III Sum of Squares** are not sequential, like Type II Sum of Squares, but *do* take into account the interaction effects. The annotation looks like:
   SS(A | B, AB) for Factor A.
   SS(B | A, AB) for Factor B.
   This type is useful if you don't want an ordering effect, and expect an interaction effect. Also, this type is ***the only one that should be used if you have unequal sample sizes.*** Types I and II should not be used if your groups have unequal sample sizes.

To change the type of Sum of Squares analysis, simply change the `'type = '` line to the desired test. Overall, this process is all fairly easy to comprehend.

5.2.2. Repeated-Measures and Mixed Design ANOVAs

The Repeated-Measures and Mixed Design ANOVAs are slightly more complicated to understand and implement. The best way I try to remember this is:

- **ANOVAs** (factorial or not) are used with *independent samples,* and are extensions of *independent t-tests.*
- **Repeated-Measures ANOVAs** are used with *paired samples,* and are extensions of *paired t-tests.*
- **Mixed Design ANOVAs** are for *any combination of independent and dependent samples.*

In essence, each ANOVA is designed to answer a different question, just like the independent and paired samples t-tests. In the last section, the data we would analyze would be considered to be a part of a measurement of independent samples. Now, with the repeated-measures ANOVA, we will analyze data from the same subjects across conditions, like time. To do this, I found that we first need to actually change the name of some of our variables so that the variable names are a single word[10], like so:

```
#Set the rate as a numeric and the state as a factor variable
data_example_long$Rate <- as.numeric(
  data_example_long$`Respiration Rate (pmol/sec/mg)`)

data_example_long$State <- as.factor(
  data_example_long$`Respiration State`)
```

I have changed the names of the 'Respiration Rate' and 'Respiration State' variables to 'Rate' and 'State' respectively. Now that we have that, we can use the `'ezANOVA'` function in the `'ez'` package as so:

```
#Load the ez package
library(ez)
```

[10] *It appears that the package we use to run is test,* `'ez'`*, doesn't like it when I have variable names containing more than one word. I'm not sure why that's the case, but I found a way around it all. This is still the easiest way to go about performing the repeated-measures ANOVA.*

```
#Run the repeated measures ANOVA
repeated_measures <- ezANOVA(data = data_example_long, dv = .(Rate), wid = .(Subject),
        within = .(Condition, State), type = "III", detailed = TRUE)
```

This function is a little different than the others. Essentially, we are choosing the data frame 'data_example_long', our dependent variable (dv) is 'Rate,' the within-subjects identifier (wid) is 'Subject,' and the two within-subjects conditions are 'Condition' and 'State.' ***It is important to have the '.()' in the code with each parameter.*** The code will not work if those are missing. Lastly, I have chosen a type III sum of squares analysis and a detailed output. When we run this, we will see an output like so:

Table 22: Repeated-Measures ANOVA Results

Effect	DFn	DFd	SSn	SSd	F	p	p<.05	ges
(Intercept)	1	26	229412.883	196.838	30302.774	0	*	0.982
Condition	1	26	1113.791	241.606	119.859	0	*	0.206
State	9	234	171655.492	2110.314	2114.871	0	*	0.976
Condition:State	9	234	1763.010	1739.587	26.350	0	*	0.291

Table 23: Sphericity Results (Repeated-Measures)

	Effect	W	p	p<.05
3	State	0.001	0	*
4	Condition:State	0.003	0	*

Table 24: Sphericity Corrections (Repeated-Measures)

	Effect	GGe	p[GG]	p[GG]<.05	HFe	p[HF]	p[HF]<.05
3	State	0.377	0	*	0.440	0	*
4	Condition:State	0.390	0	*	0.458	0	*

Now, we have a much different layout than we did before, but it follows the same pattern. We first have the degrees of freedom values, sums of squares, F-value, p-value, a column with asterisks representing significance, and finally we have a general eta-squared (more on this in the Effect Size section). The next set of rows are the results of Mauchly's Test for Sphericity, followed by the corrections if sphericity has been violated. In this case, if sphericity has been violated (indicated by significance in Mauchly's test), then it is best to use the Greenhouse-Geisser correction (GGe column) and the p-values from the p[GG] column. Here, we see that there was a significant difference between baseline and post-intervention, a difference somewhere in the respiration state, and an interaction effect.



Overall this is fairly straightforward, but it is different than the past tests we did. Luckily, the `'ezANOVA'` function is fairly robust and we don't need to change much if we want to do a mixed designs ANOVA. All we would have to add is a between factor, as so:

```
#Run the mixed design ANOVA
mixed_design <- ezANOVA(data = data_example_long, dv = .(Rate), wid = .(Subject),
        between = .(Leg), within = .(Condition, State),
        type = "III", detailed = TRUE)
```

Table 25: Mixed Design ANOVA Results

	Effect	DFn	DFd	SSn	SSd	F	p	p<.05	ges
1	(Intercept)	1	25	229234.209	182.124	31466.772	0.000	*	0.982
2	Leg	1	25	14.714	182.124	2.020	0.168		0.004
3	Condition	1	25	1114.806	240.548	115.861	0.000	*	0.214
5	State	9	225	171611.627	2041.300	2101.744	0.000	*	0.977
4	Leg:Condition	1	25	1.058	240.548	0.110	0.743		0.000
6	Leg:State	9	225	69.014	2041.300	0.845	0.575		0.017
7	Condition:State	9	225	1750.513	1642.660	26.641	0.000	*	0.299
8	Leg:Condition:State	9	225	96.927	1642.660	1.475	0.158		0.023

Table 26: Sphericity Results (Mixed Design)

	Effect	W	p	p<.05
5	State	0.001	0	*
6	Leg:State	0.001	0	*
7	Condition:State	0.003	0	*
8	Leg:Condition:State	0.003	0	*

Table 27: Sphericity Corrections (Mixed Design)

	Effect	GGe	p[GG]	p[GG]<.05	HFe	p[HF]	p[HF]<.05
5	State	0.374	0.000	*	0.440	0.000	*
6	Leg:State	0.374	0.484		0.440	0.499	
7	Condition:State	0.393	0.000	*	0.466	0.000	*
8	Leg:Condition:State	0.393	0.221		0.466	0.213	

Of course, here we will have a much larger set of tables because we have introduced another factor (Leg) to our analysis, but everything is the same as the repeated-measures ANOVA. Here, our results show the same outcomes as before (significant differences in condition and respiration), but the leg that the sample came from seemed to have no impact on respiration

rate.

5.2.3. Non-parametric tests for more than two samples

When performing non-parametric equivalents for data with multiple groups, there are two tests we can do: the Kruskal-Wallis test (the equivalent to a one-way ANOVA). **These two tests do not perform two-way tests. For two-way non-parametric analyses, there are some tests available in the `WRS2` package, but I am much less familiar with these.** The overall processes are very similar to what we did for the parametric tests. For the Kruskal-Wallis test, we will simply use the following:

```
kruskal.test(data = data_example_long, `Respiration Rate (pmol/sec/mg)` ~ `Condition`)
```

```
##
##  Kruskal-Wallis rank sum test
##
## data:  Respiration Rate (pmol/sec/mg) by Condition
## Kruskal-Wallis chi-squared = 3.5298, df = 1, p-value = 0.06028
```

5.2.4. *Post-hoc* Methods

Once you're done with the ANOVA analysis, you'll probably want to perform a *post hoc* analysis to determine **where** exactly the significant differences are (remember, ANOVA only tells you that there is a good chance that a significant difference exists in the data, but does not tell you which values are significant). Fortunately, it's also pretty simple to run all of these.

Pairwise Tests

To perform Tukey's HSD for the factorial ANOVA, all we need to do is take the ANOVA model we made previously and put it into the `'lsmeans'` function from the `'lsmeans'` package, as so:

```
#Load the package lsmeans
library(lsmeans)

#Calculate the Tukey corrections
tukey_results<- lsmeans(ANOVA_model, pairwise ~ `Respiration State`:Condition,
        adjust = "tukey", paired = TRUE)

#Show the first 10 results of each section
head(tukey_results$lsmeans)
```

```
##  Respiration State Condition lsmean    SE  df lower.CL upper.CL
##  Basal             Baseline     3.6 0.483 529     2.65     4.55
##
## Confidence level used: 0.95
```



```
head(tukey_results$contrasts)
```

```
##  contrast                          estimate    SE  df t.ratio p.value
##  Basal Baseline - OctM Baseline      -2.4 0.651 529 -3.682  0.0003
```

This table is extremely long, especially considering the huge number of contrasts we have performed. Therefore, I will not include the output, but it should be fairly self-explanatory if you've ever done a *post-hoc* analysis in the past. You can also do other adjustments, such as the Scheffe or Sidak adjustments by typing in `"scheffe"` or `"sidak"` instead of `"tukey"`. The nice thing about this method is that it gives you the lower and upper confidence levels. However, it is slightly more difficult to interpret. If you don't care about the confidence levels and only want the adjusted p-values, you can use the following for both factorial and repeated measures ANOVAs:

```
#Calculate the Holm correction
holm_results <- pairwise.t.test(x = data_example_long$`Rate`,
      g = data_example_long$State:data_example_long$Condition,
      paired = TRUE, p.adjust.method = "holm")
```

As with the other adjustment method, this test can use the Holm, Hochberg, Hommel, Bonferroni, Benjamini-Hochberg or its alias FDR, or the Benjamini-Yekutieili adjustment by using `"holm"`, `"hochberg"`, `"hommel"`, `"bonferroni"`, `"BH"`, `"FDR"`, or `"BY"` respectively. This code can also be done for Wilcoxon tests as so:

```
#Wilcoxon Comparison
wilcoxon_results <- pairwise.wilcox.test(x = data_example_long$`Rate`,
      g = data_example_long$State:data_example_long$Condition,
      paired = TRUE, p.adjust.method = "holm")
```

5.2.5. Planned Comparisons

Now, running *post-hoc* tests can be slightly tedious and difficult, especially when there are so many data points that don't matter to our story. For example, we would probably not care about the difference between basal respiration at baseline and respiration with oligomycin post-intervention; but we would certainly care a lot about the difference in maximal respiration at baseline compared to post-intervention. In fact, it would probably be best to compare the baseline and post-intervention values for all of the respiration states (like we did with the t-tests). In all probability, it is probably best to just run the t-tests with a p-value adjustment, but I will try to show contrasts here. Be warned: they can be complicated.



6. Effect Sizes

Effect sizes are often useful counterparts to your p-values for when you want to add more information to your data. Since effect sizes are best done in long format after doing t-tests, I'll probably just write about those for now and add in something about ANOVAs later. The reason for that is that effect sizes from ANOVAs usually require contrasts, which I haven't yet developed.

To calculate Cohen's *d* from the t-test data, simply use the `'effectsize'` package as so:

```
#Load the package effectsize
library(effectsize)

#Calculate ES
effect_sizes <- data.frame(t(sapply(data_example_wide[,5:15],
        function(x) effectsize::cohens_d(x, y = "Condition",
        data = data_example_wide, pooled = TRUE))))
```

This is essentially the same function used to do the t-tests, just with a different function applied to the loop.[11] This gives us the output as follows:

	Cohens_d	CI	CI_low	CI_high
Basal	-0.2571926	0.95	-0.791678	0.2797396
OctM	-0.4184311	0.95	-0.9559002	0.1229663
OctMD(25)	-0.842419	0.95	-1.396054	-0.2813549
OctMD(50)	-1.178249	0.95	-1.752716	-0.5941294
OctMD(250)	-0.5887667	0.95	-1.131306	-0.04081189
OctMD(5000)	-2.532173	0.95	-3.246538	-1.804081
OctMDGS	-2.498371	0.95	-3.208407	-1.774637
CytC	-1.988454	0.95	-2.637654	-1.326368
FCCP Peak	-0.763511	0.95	-1.31325	-0.2069403
Omy	0.5837923	0.95	0.03605506	1.126156
AmA	-0.5611865	0.95	-1.102773	-0.01441883

The only thing I have with these results is that the effect size values are the negative values of what they should be. If you need to change this, just do the following:

```
effect_sizes$Cohens_d <- as.numeric(effect_sizes$Cohens_d)

effect_sizes$Cohens_d_new <- effect_sizes$Cohens_d[] * -1
```

Or if you want to have the absolute values:

[11] *I added the t() function which transposes the data, only to make it easier to read.*

```
effect_sizes$Cohens_d_all_pos <- abs(effect_sizes$Cohens_d)
```

If you care to do the effect sizes as a pairwise comparison, you can create contrasts fairly simply and run a quick analysis to get the effect sizes for each pairwise comparison like so:

```
#Create a function to calculate contrasts
rcontrast <- function(t, df)
{r <- sqrt(t^2/(t^2 + df))
    paste("r = ", r)}

#Create the data frame of variables to be analyzed
tukey_contrasts <- data.frame(tukey_results$contrasts)

#Calculate pairwise effect sizes using the created contrasts function above
tukey_contrasts$`Effect Size` <- t(data.frame(lapply(
  tukey_contrasts$t.ratio, function(x) rcontrast(x, 529))))
```

contrast	estimate	SE	df	t.ratio	p.value	Effect Size
Basal Baseline - OctM Baseline	-2.3969109	0.6509035	529	-3.6824367	0.033683	r = 0.158092501548327
Basal Baseline - OctMD(25) Baseline	-4.9985949	0.6509035	529	-7.6794716	0.000000	r = 0.316702996813579
Basal Baseline - OctMD(50) Baseline	-7.2220573	0.6509035	529	-11.0954349	0.000000	r = 0.434494558554741
Basal Baseline - OctMD(250) Baseline	-9.3728871	0.6509035	529	-14.3998107	0.000000	r = 0.530656265685931
Basal Baseline - OctMD(5000) Baseline	-18.3869404	0.6509035	529	-28.2483358	0.000000	r = 0.775465649229539
Basal Baseline - OctMDGS Baseline	-44.5486270	0.6509035	529	-68.4412170	0.000000	r = 0.947906546157429
Basal Baseline - FCCP Peak Baseline	-53.6615325	0.6509035	529	-82.4416113	0.000000	r = 0.963217356044235
Basal Baseline - Omy Baseline	-15.2348279	0.6509035	529	-23.4056633	0.000000	r = 0.713260827489314
Basal Baseline - AmA Baseline	0.0578763	0.6509035	529	0.0889169	1.000000	r = 0.00386592368367269
Basal Baseline - Basal Post	-2.8723344	0.2910929	529	-9.8674153	0.000000	r = 0.394266030215752

7. Visualizing the Data: The Simple Guide to ggplot2

7.1. The Basics

7.1.1. The Importance of Visualization

I think the importance of data visualization goes without saying: seeing the data is much more telling than reporting the numbers. Thus, painting an accurate picture of the data at hand is imperative for the communication of scientific data. Not only do we want readers to visualize our results in a meaningful way, we also want to provide an explaination of our results in a manner that compliments our manuscript. In this section, I till try to detail a few different plots (mostly using the packages `'ggplot2'` and `'ggpubr'`) that can be very helpful for the interpretation and communication of respirometry data.

7.1.2. Understanding Plotting in R

Plotting in R can be fairly simple, once the environment of R (and ggplot2) is well understood. Essentially, graphing in R works by creating a base graph with the data you will be using, then creating layers upon layers of the data you want to visualize. The process is analogous to building a cake: you start with the inner cake layers (data), then add icing (data points), some color, a message like "Happy Birthday!" (titles, legends, etc.), maybe a few candles (significance asterisks), and *Voila*! Your graph is made.

Each layer we place to our code will add another detail to the graph. However, while this is a very powerful feature in R, we must always remember that the simplest graphs are often the best graphs. It is important to not get carried away when graphing, such as adding too many points or details that confuse the reader, rather than compliment the story. When making graphs, it is best to always be efficient with your space and colors — guiding the reader to the most important information using proper colors and visuals while having sufficient negative space.

7.1.3. ggplot2 and ggpubr: The two graphing packages of choice

The two most popular graphing tools that are used in R are `'ggplot2'` and `'ggpubr'`, both of which are important tools to create great visualizations. The ggplot2 package is the most well-known visualization package used in R. It was developed by Hadley Wickham (the same person who developed the Tidyverse) based on the Grammar of Graphics. Logically,`'ggplot2'` is based on the idea that you can build layer upon layer to get the final graphical outcome of choice.

Similarly, `'ggpubr'` is founded on the same principles, but is created to "generate publication-ready graphics." Both packages are relatively similar in how they operate, so choose the package you like best. Generally, I think `'ggpubr'` is easier to work with, but `'ggplot2'` has the advantage of being more flexible in many aspects. Another thing to note is that `'ggpubr'` *does not* work well with variable names that have more than one string of characters in it (i.e. more than one word), meaning you can only use factors that contain a single word in `'ggpubr'`. It isn't the greatest, but it still works fine — you can also just simply rename the factor.

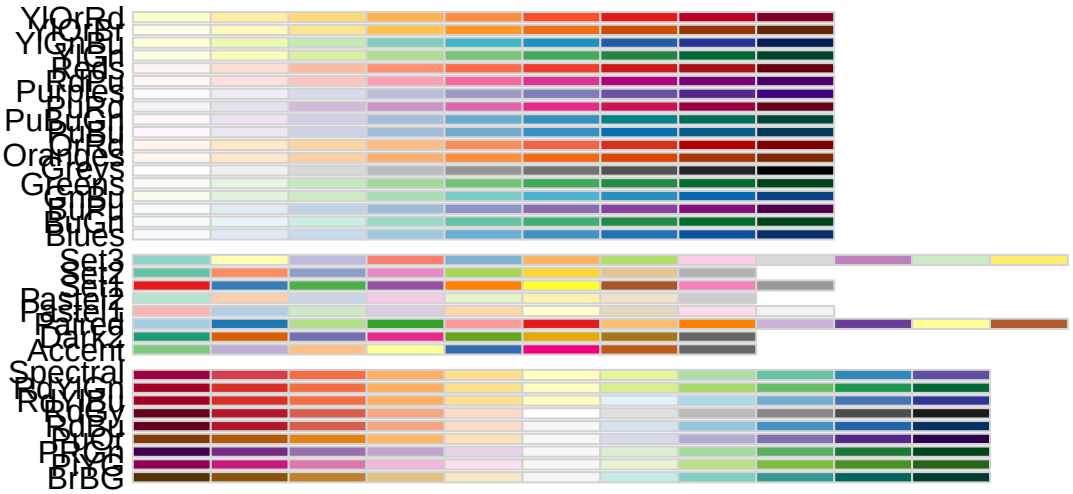
7.1.4. Color Palettes

Choosing color is an important aspect of graphing. Not only does one need to consider the audience and message (colors are associated with feelings, but are largely dependent on sociocultural norms). Furthermore, a fair portion of the population is colorblind. Therefore, to convey an appropriate message, one must consider strongly the audience and message of the manuscript. That said, color is not something one wants to mess up when preparing figures. R is great in that you can create many different color combinations to make awesome graphs. A useful cheatsheet to this can be found here. However, choosing colors on your own can still be a tricky task.

Luckily for us, there are several packages that contain color palettes (and a more comprehensive list can be found here). Three common packages are `'RColorBrewer'`, `'ggsci'`, and `'wesanderson'` (yes, like the film director). Let's take a look at these, starting with `'RColorBrewer'`:

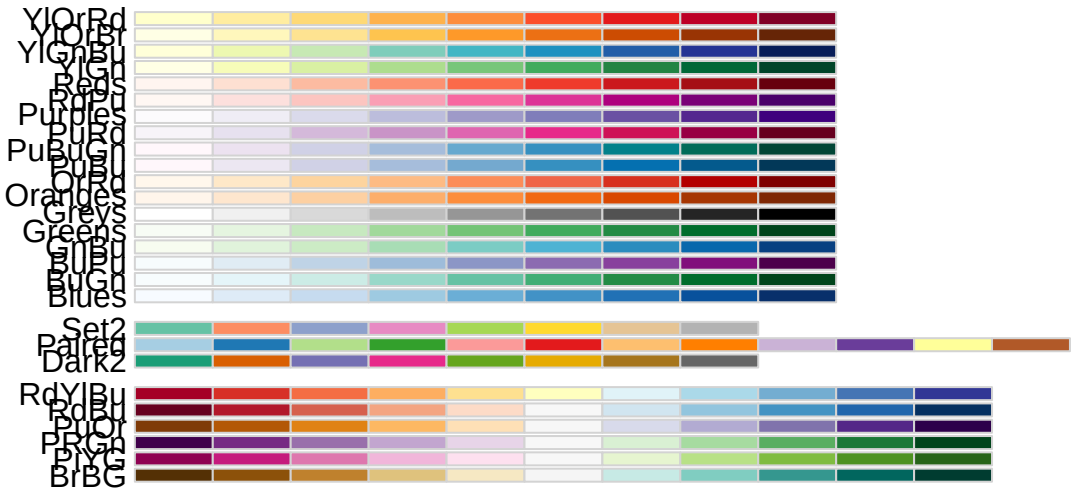
```
#Load RColorBrewer
library(RColorBrewer)

#Show palettes
display.brewer.all()
```



The nice thing about `'RColorBrewer'` is that there are specific palettes that are colorblind, which we can identify by:

```
display.brewer.all(colorblindFriendly = TRUE)
```



| The `'ggsci'` package contains several palettes that are used in major journals (Nature, Lancet, others). We can only view these online for now.

For the `'wesanderson'` palette, we can't see all of the palettes unless you go to the GitHub site, but all of the palette names can be listed by:

```
library(wesanderson)

names(wes_palettes)
```

```
##  [1] "BottleRocket1"   "BottleRocket2"   "Rushmore1"       "Rushmore"
##  [5] "Royal1"          "Royal2"          "Zissou1"         "Darjeeling1"
##  [9] "Darjeeling2"     "Chevalier1"      "FantasticFox1"   "Moonrise1"
## [13] "Moonrise2"       "Moonrise3"       "Cavalcanti1"     "GrandBudapest1"
## [17] "GrandBudapest2"  "IsleofDogs1"     "IsleofDogs2"
```

Which you can then view each palette individually by:

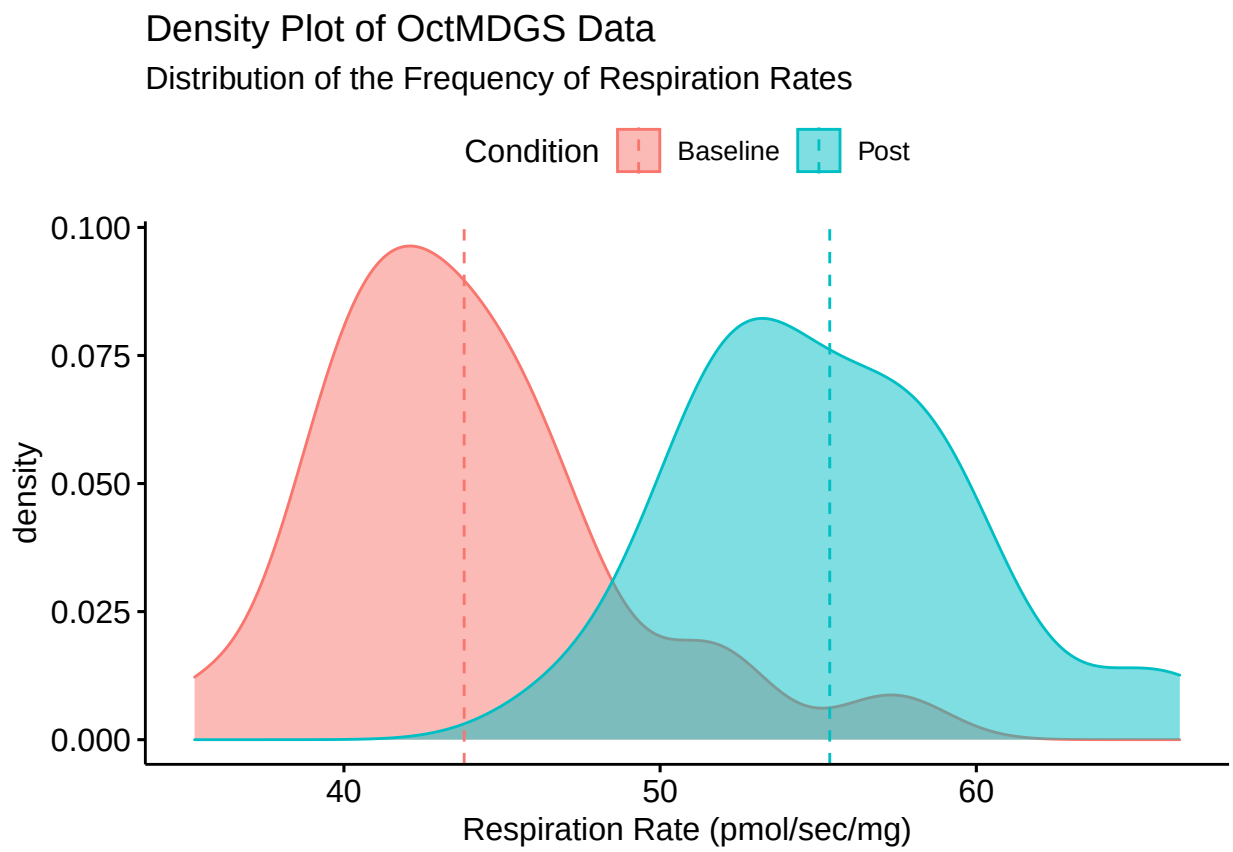
```
wes_palette("FantasticFox1")
```

7.2. Density Plots

Density plots are useful to understand the distribution of the data, and to visualize the spread of data points. In my experience, I have found that `'ggpubr'` is much easier to work with in creating simple density plots compared to `'ggplot'`, but this is mostly because of my personal preference. Now, when we use `'ggpubr'` for this, we need to remember that we must input all of the variables we need into the function. So, when we call `ggdensity()` to create the plot, we need to call the data, call the `x` variable, and input whatever else we need. Functions like `color` and `fill` will indicate how to separate the color of the lines and the fill color of the variables you have. If we want to show where the mean is, we can also do so by `'add = "mean"'`. So, if I want to look at the distribution of State III respiration data (OctMGDS), I could do something like:

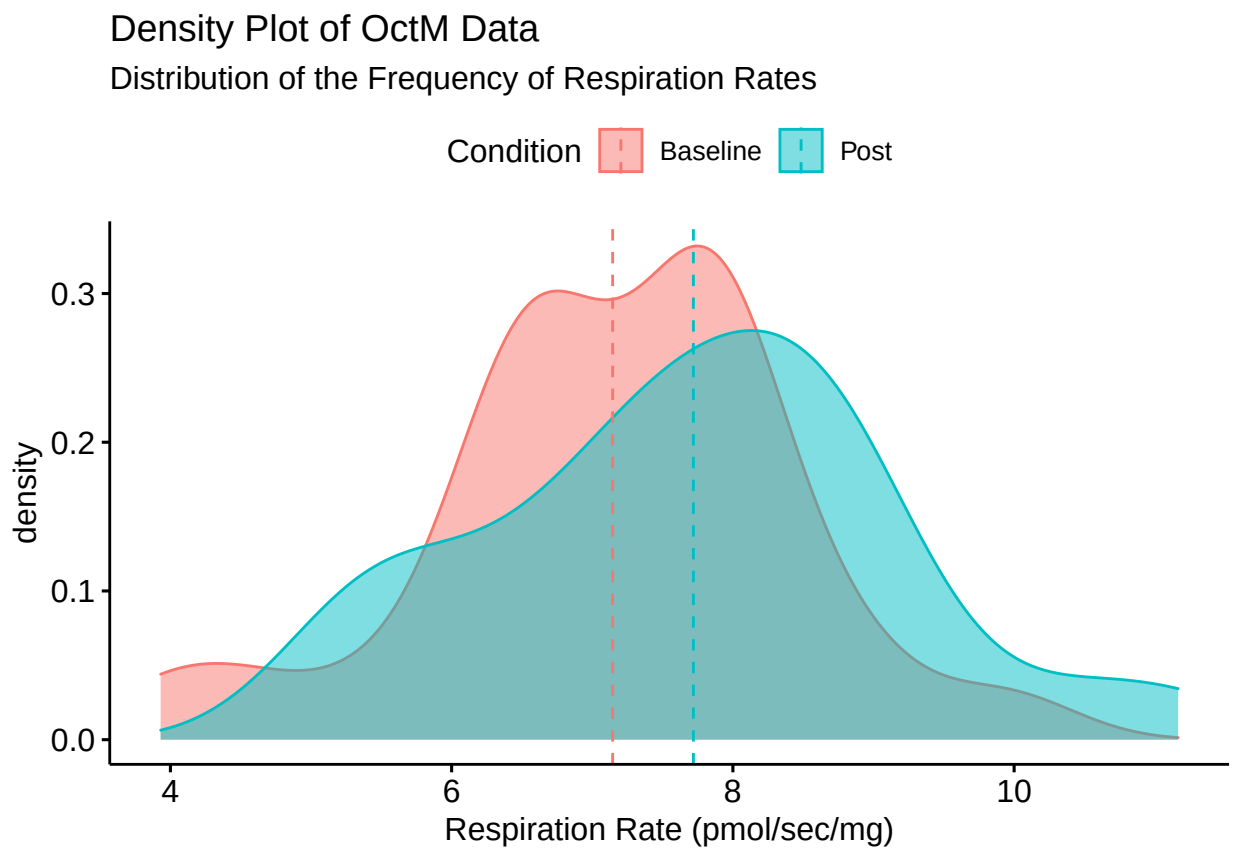
```
#Load ggpubr
library(ggpubr)

ggdensity(data = data_example_wide, x = "OctMDGS", color = "Condition",
          fill = "Condition", add = "mean",
          xlab = "Respiration Rate (pmol/sec/mg)",
          title = "Density Plot of OctMDGS Data", subtitle = "Distribution of the Freque
```



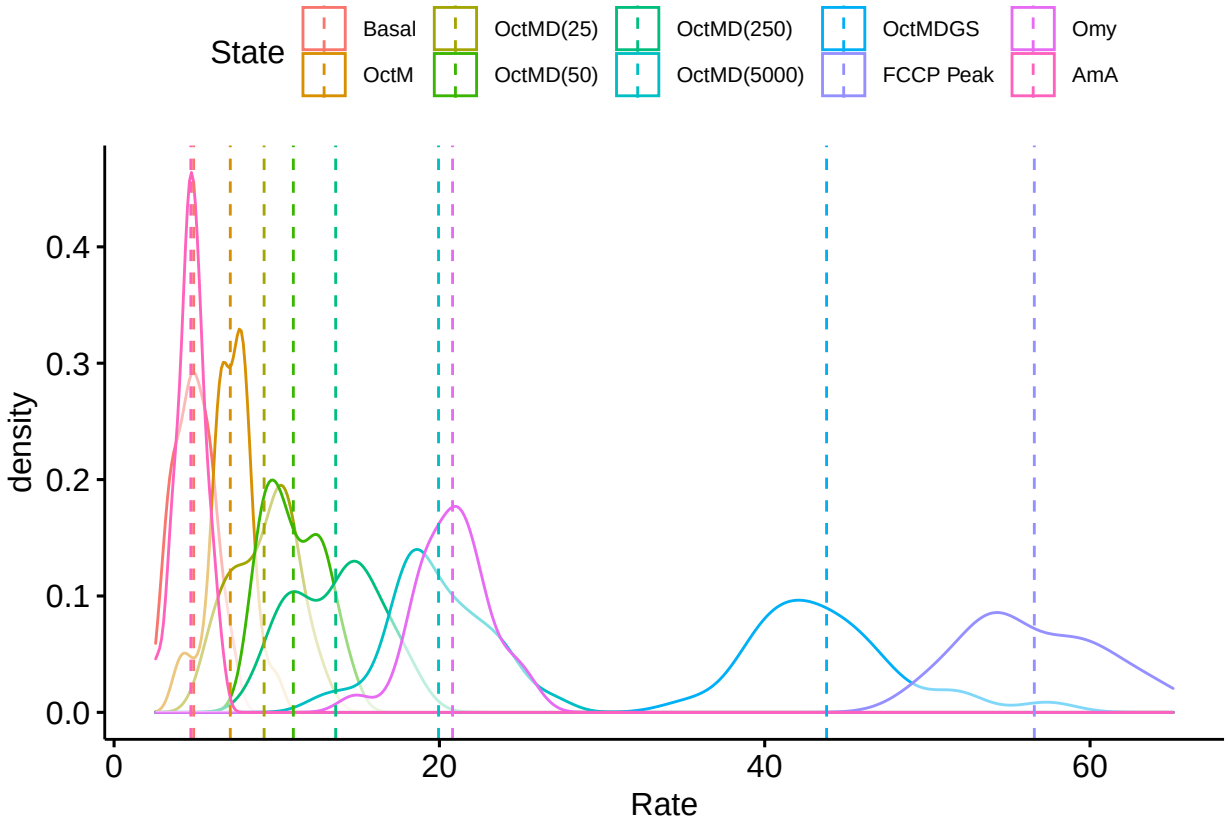
Likewise, if I wanted to look at leak respiration (OctM), I could also do:

```
ggdensity(data = data_example_wide, x = "OctM", color = "Condition",
          fill = "Condition", add = "mean",
          xlab = "Respiration Rate (pmol/sec/mg)",
          title = "Density Plot of OctM Data", subtitle = "Distribution of the Frequency
```



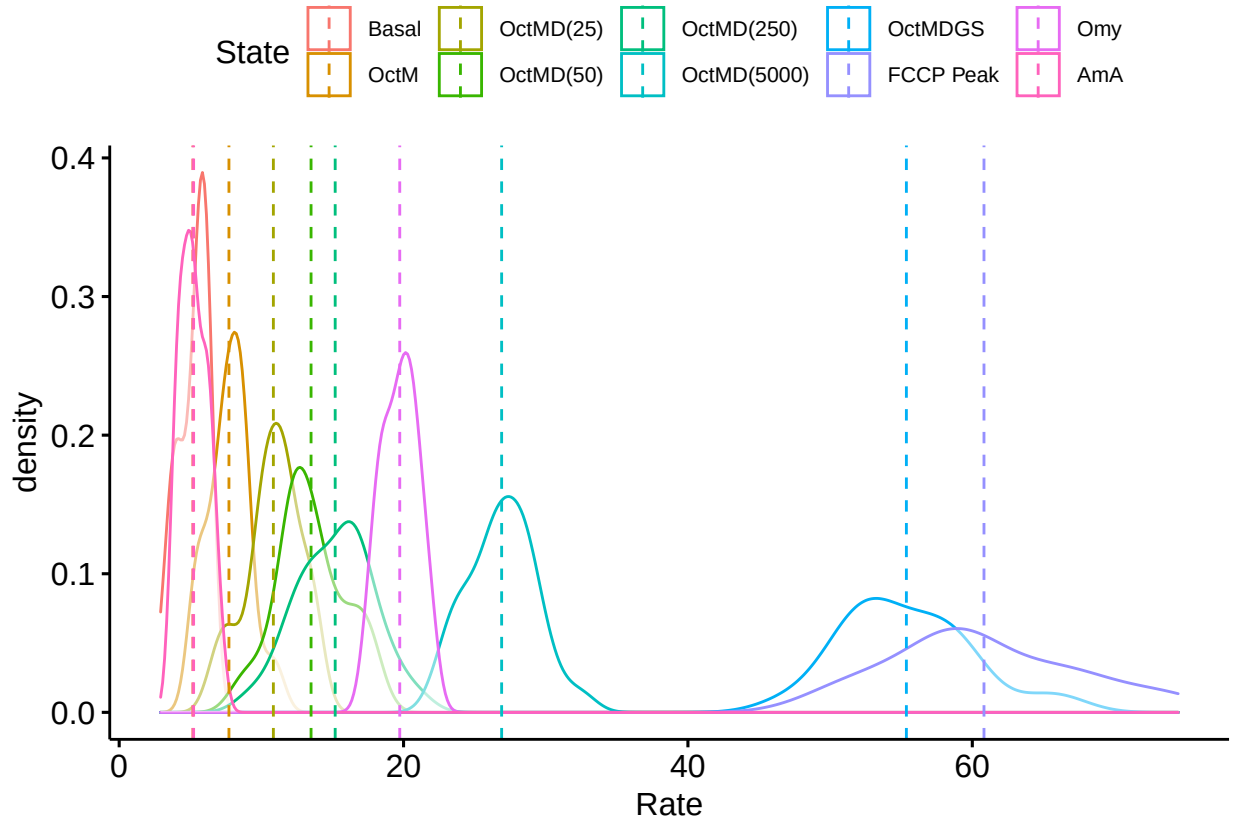
Similarly, we can show the distribution of all of the Baseline data by using the `'subset'` and `'select()'` functions, and the `'%in%'` operator to select only the baseline respiration data like so:

```
ggdensity(data  = subset(data_example_long, Condition %in% c("Baseline")),
          x = "Rate", color = "State", add = "mean") +
  theme(legend.text = element_text(size = 8))
```



And we can run it on the post-interverntion:

```
ggdensity(subset(data_example_long, Condition %in% c("Post")),
          x = "Rate", color = "State", add = "mean") +
  theme(legend.text = element_text(size = 8))
```

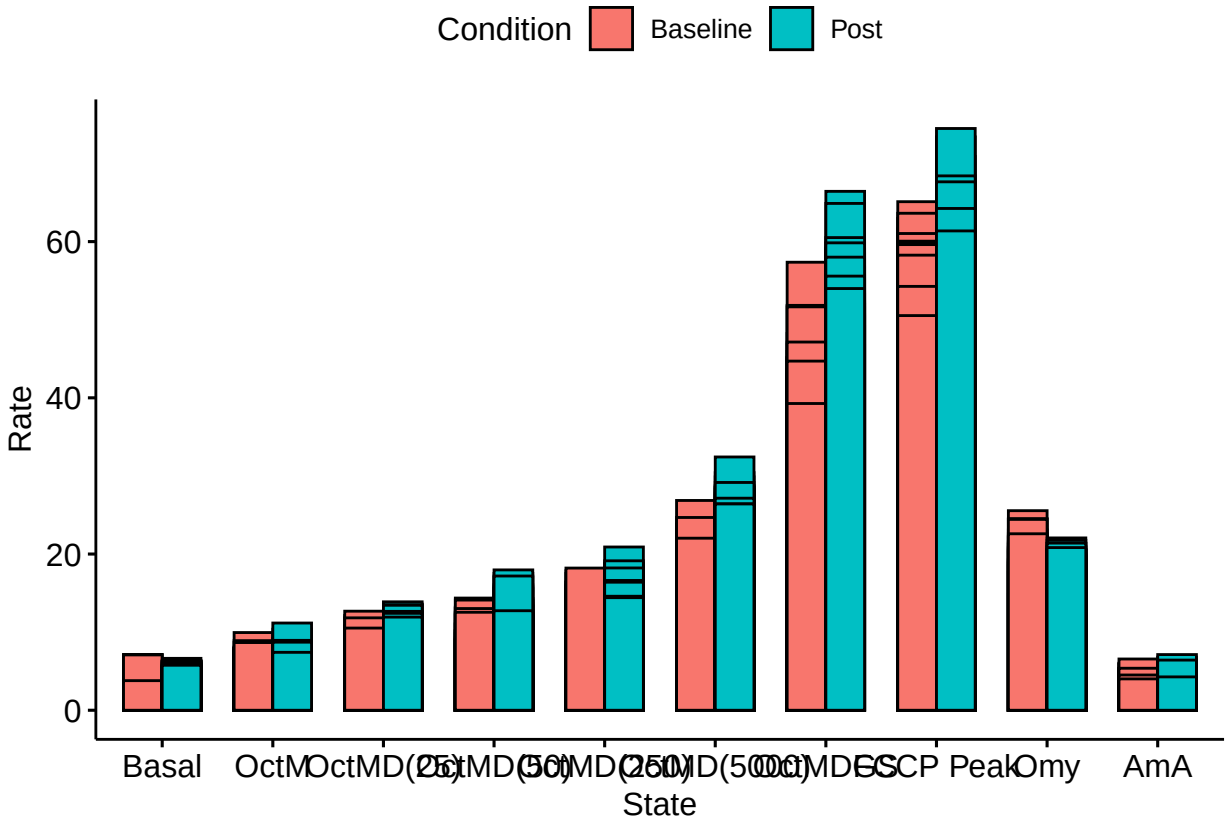


7.3. Bar Plots

Bar plots are perhaps the most commonly used type of graph in the literature, so it only seems appropriate to cover them in this document. This is really the part of the section where layers become important. Furthermore, I will try to outline as much code as I can, as well as explain any notes. As before, `'ggpubr'` is the simplest function to use. If we use `'ggpubr'`, we can make a simple plot by the following code:

```
#Create the basic plot
ggpubr_bar <- ggbarplot(data = data_example_long, x = "State", y = "Rate", fill = "Condi
                        position = position_dodge(0.7))

ggpubr_bar
```



Each time we create a plot, we must, at the very least, tell R the dataset we want to use (via `'data = '`), and the x and y variables. If you have a grouping variable (as we do with the 'Condition'), you can designate that with the `'fill = '` line — however, you also need to have the position of the bars (via `'position = position_dodge(0.7)'`) specifically stated as well or else you will be given a stacked bar chart. Now, as you can probably tell, our graph is very messy and is missing a lot of components. To begin cleaning it up, we will add a mean and error bars, like so:

```
ggpubr_bar <- ggbarplot(data = data_example_long, x = "State", y = "Rate", fill = "Condi
                        position = position_dodge(0.7), add = "mean_se", error.plot = "u


ggpubr_bar
```

We can also change the color by inputting a `'palette ='` line like so (this is where you can use a palette or your own color choices, as I have):

```
ggpubr_bar <- ggbarplot(data = data_example_long, x = "State", y = "Rate",
                        fill = "Condition",
                        position = position_dodge(0.7),
                        add = "mean_se", error.plot = "upper_errorbar",
                        palette = c("white", "dark red"))
```

Much better. Now, we should probably begin working out the aesthetics with this by rotating

the text and starting the y axis at 0, which we will do by adding layers (via '+') using `rotate_x_test()` and `scale_y_continuous()`, like so

```
ggpubr_bar <- ggbarplot(data = data_example_long, x = "State", y = "Rate",
                        fill = "Condition",
                        position = position_dodge(0.7),
                        add = "mean_se", error.plot = "upper_errorbar",
                        palette = c("white", "dark red")) +
  rotate_x_text(45) +
  scale_y_continuous(expand = c(0,0))
```

We can add a title and change the labels, by using the `ggtitle()` and `labs()` functions

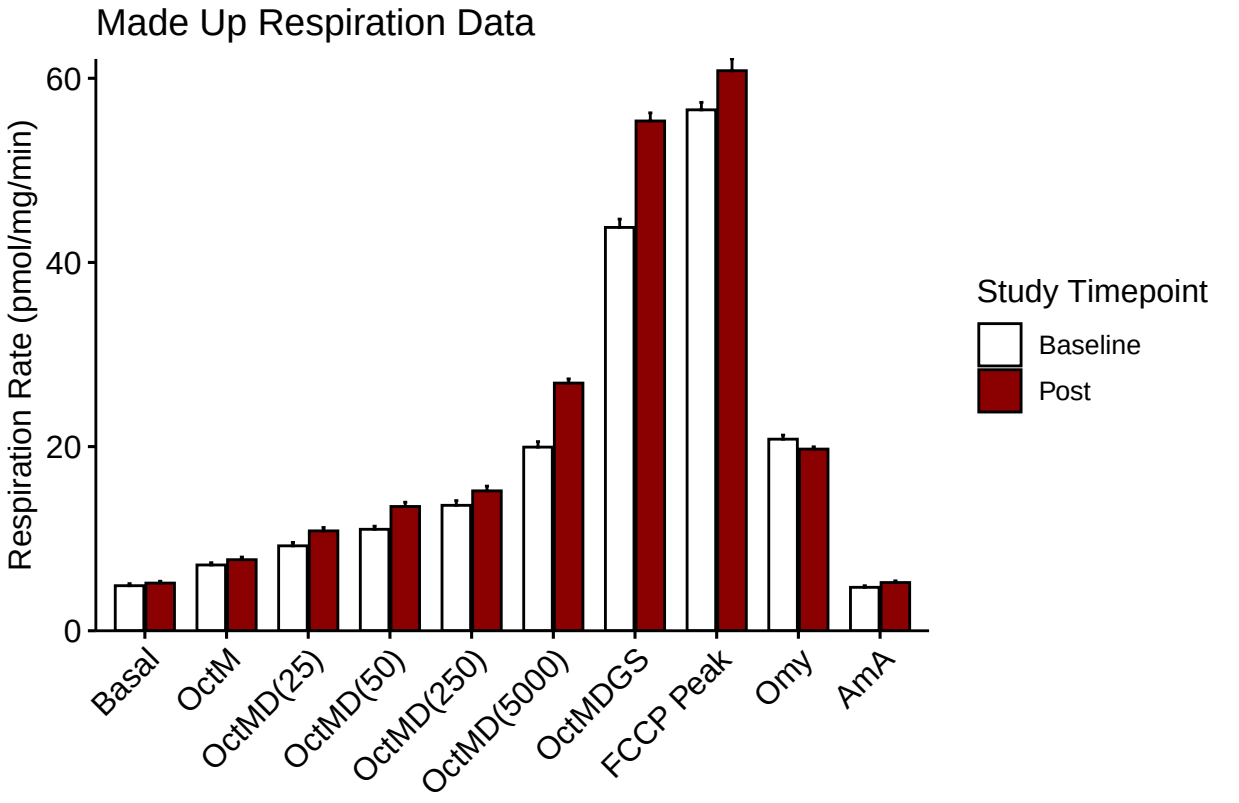
```
ggpubr_bar <- ggbarplot(data = data_example_long, x = "State", y = "Rate",
                        fill = "Condition",
                        position = position_dodge(0.7),
                        add = "mean_se", error.plot = "upper_errorbar",
                        palette = c("white", "dark red")) +
  rotate_x_text(45) +
  scale_y_continuous(expand = c(0,0)) +
  ggtitle("Made Up Respiration Data") +
  labs(x = "", y = "Respiration Rate (pmol/mg/min)",
       fill = "Study Timepoint")
```

And we can change many things in the `theme()` function, which alters the basic layout such as background color, line/text thickness, and legend position

```
#Set up basic plot (data, colors, layout)
ggpubr_bar <- ggbarplot(data = data_example_long,
                        x = "State", y = "Rate",
                        fill = "Condition",
                        position = position_dodge(0.75),
                        add = "mean_se", error.plot = "upper_errorbar",
                        palette = c("white", "dark red")) +
  #Rotate text
  rotate_x_text(45) +
  #Adjust axis position
  scale_y_continuous(expand = c(0,0)) +
  #Add title
  ggtitle("Made Up Respiration Data") +
  #Change axis labels
  labs(x = "", y = "Respiration Rate (pmol/mg/min)",
       fill = "Study Timepoint") +
  #Move legend
  theme(legend.position = "right")
```



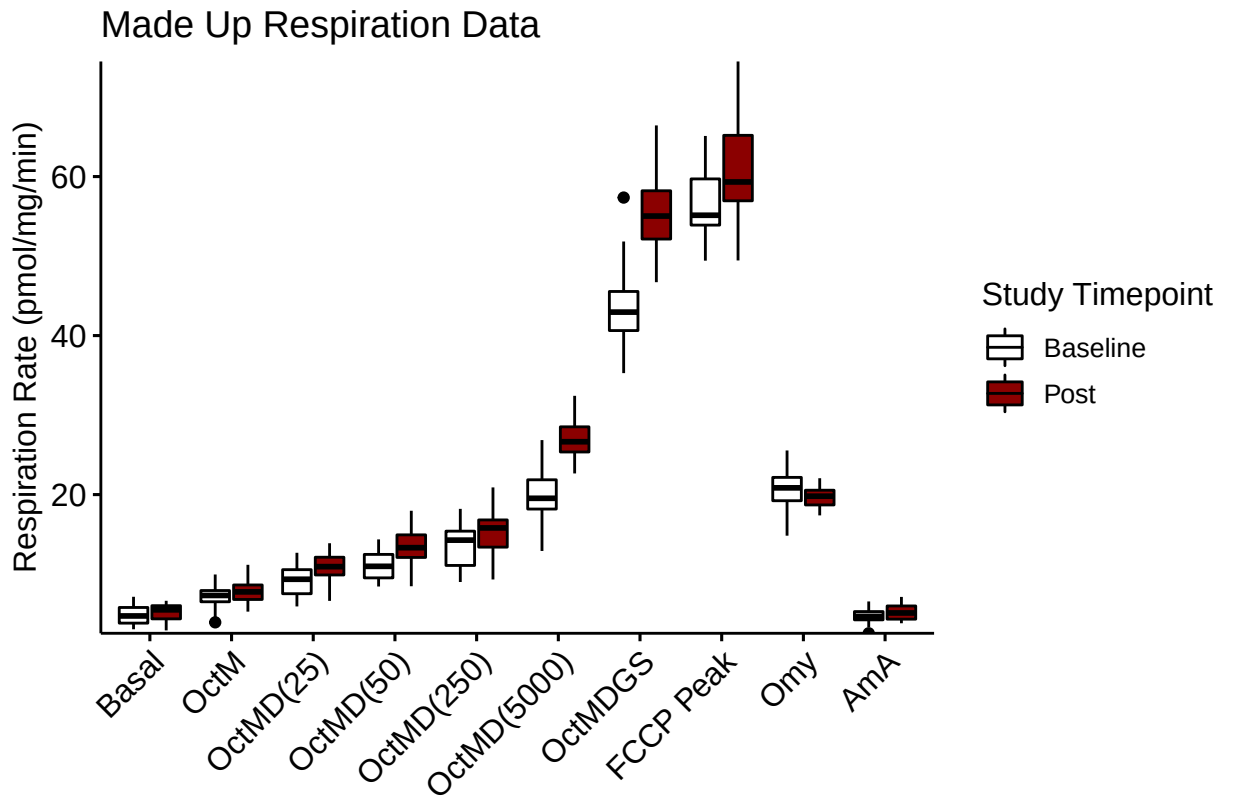
```
ggpubr_bar
```



7.4. Box Plots {#BoxPlots} Box plots follow a similar pattern, but we will use the `ggboxplot` function and remove some of the lines of code that are specific to barplots (such as the positioning and mean/SE), like this:

```
#Set up basic plot (data, colors, layout)
ggpubr_box <- ggboxplot(data = data_example_long,
                        x = "State", y = "Rate",
                        fill = "Condition",
                        palette = c("white", "dark red")) +
  #Rotate text
  rotate_x_text(45) +
  #Adjust axis position
  scale_y_continuous(expand = c(0,0)) +
  #Add title
  ggtitle("Made Up Respiration Data") +
  #Change axis labels
  labs(x = "", y = "Respiration Rate (pmol/mg/min)",
       fill = "Study Timepoint") +
  #Move legend
  theme(legend.position = "right")
```

```
ggpubr_box
```



7.5. Adding Significance

Significance in R can be pretty tricky. To be frank, I don't think there is a great way to do this in R. You can place in significance symbols manually, if needed, but that takes a lot of time. You can use ggsignif as well, but you can't really change much about the output of the plot. I will show a way to do both.

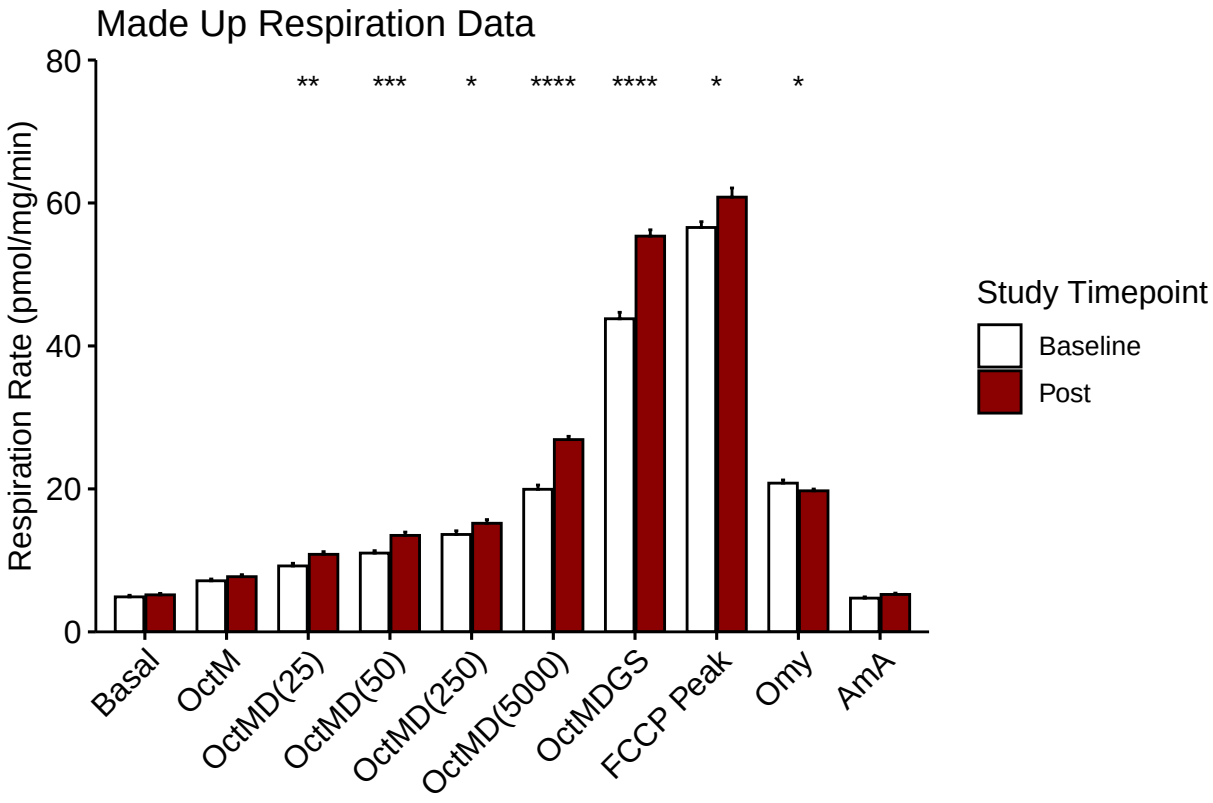
7.5.1. Significance with ggsignif

GGsignif is by far the easiest method to display significance in graphs, however, you will only get one output in the graphs (you cannot customize the asterisks).

If we take our pervious graph (either the barplot or the boxplot), all we would need to do is load the `ggsignif` package and run the `stat_compare_means`. Doing this, we will also specify that we want asterisks by using `label = "p.signif"` and hiding non-significant valuies by stating `hide.ns = TRUE` function like so:

```
ggpubr_bar + stat_compare_means(aes(group = `Condition`),
                                label = "p.signif",
```

```
                                hide.ns = TRUE,
                                label.y = 75) +
  scale_y_continuous(expand = c(0,0),
                     limits = c(0,80))
```



Note that you may have to change the scaling, as I did, by using the `limits` command `scale_y_continuous`.

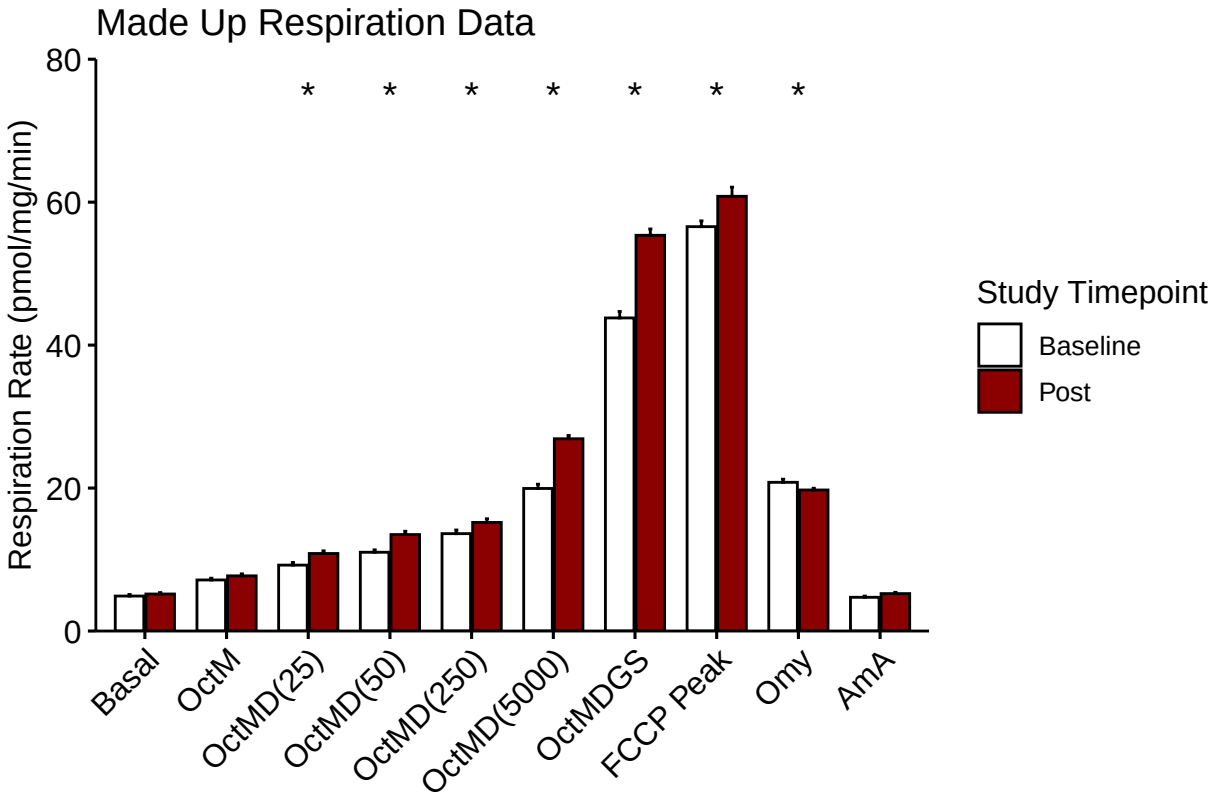
7.5.2. Manual Significance

Adding significance manually can be a pain, but I think it looks better than the `stat_compare_means()` function because you can place special asterisks or other marks wherever you need them to go. `stat_compare_means()` only places the asterisks in their default placement, to my knowledge. This may change in the future, though.

Fortunately, these significance markers aren't too challenging to place, but it does require *some* effort. To add special characters (or even simple letters) to the graph, we will simply use the `annotate()` function in ggplot like so:

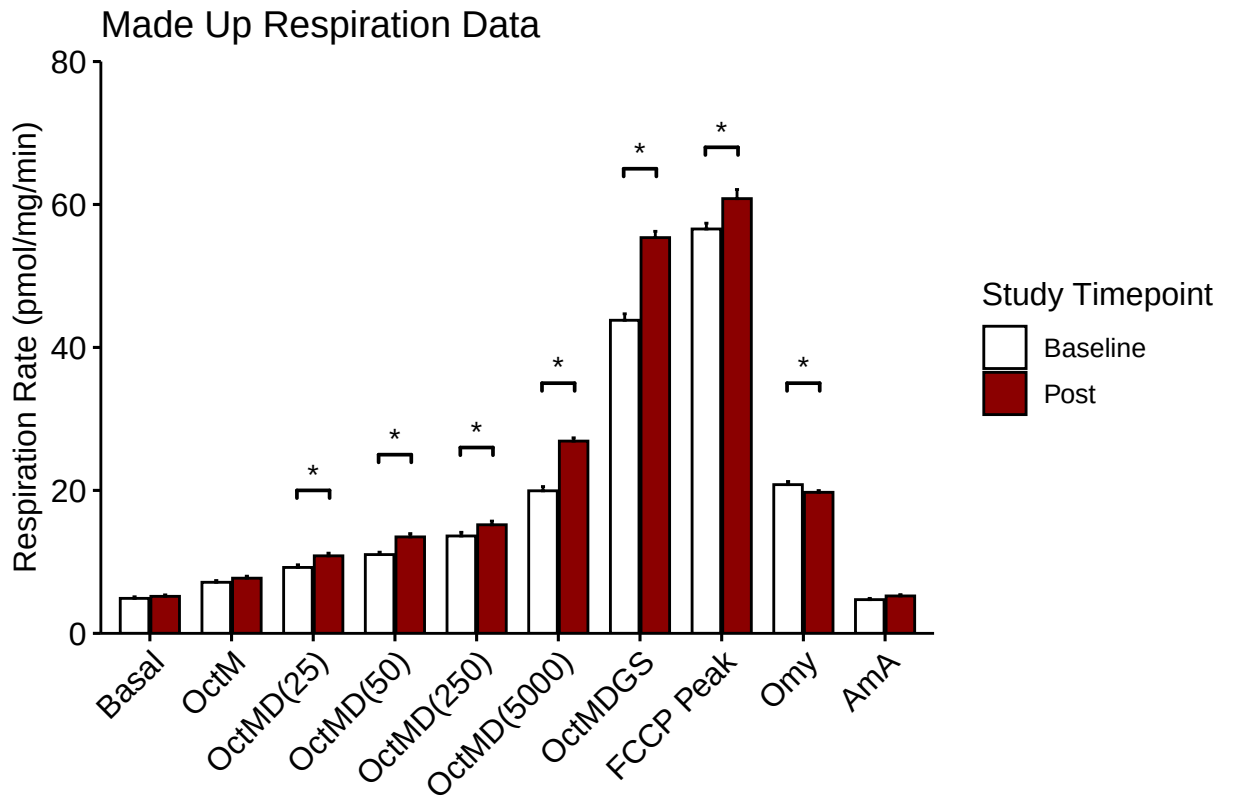
```
ggpubr_bar +
  scale_y_continuous(expand = c(0,0),
                     limits = c(0,80)) +
  #Add stars
```

```
  annotate('text', x = c(3,4,5,6,7,8,9), y = 75,
           label = "*", size = 5)
```



To add brackets, we will use `geom_bracket()` like so:

```
ggpubr_bar +
  scale_y_continuous(expand = c(0,0),
                     limits = c(0,80)) +
 #Add brackets
  geom_bracket(inherit.aes = TRUE,
               #Set left edge for each bracket
               xmin = c(2.8, 3.8, 4.8, 5.8, 6.8, 7.8, 8.8),
               #set right edge for each bracket
               xmax = c(3.2, 4.2, 5.2, 6.2, 7.2, 8.2, 9.2),
               #position for each bracket
               y.position = c(20, 25, 26, 35, 65, 68, 35),
               #adjust tip length
               tip.length = 0.015,
               #Place asterisk on each bracket
               label = "*",
               size = .65)
```

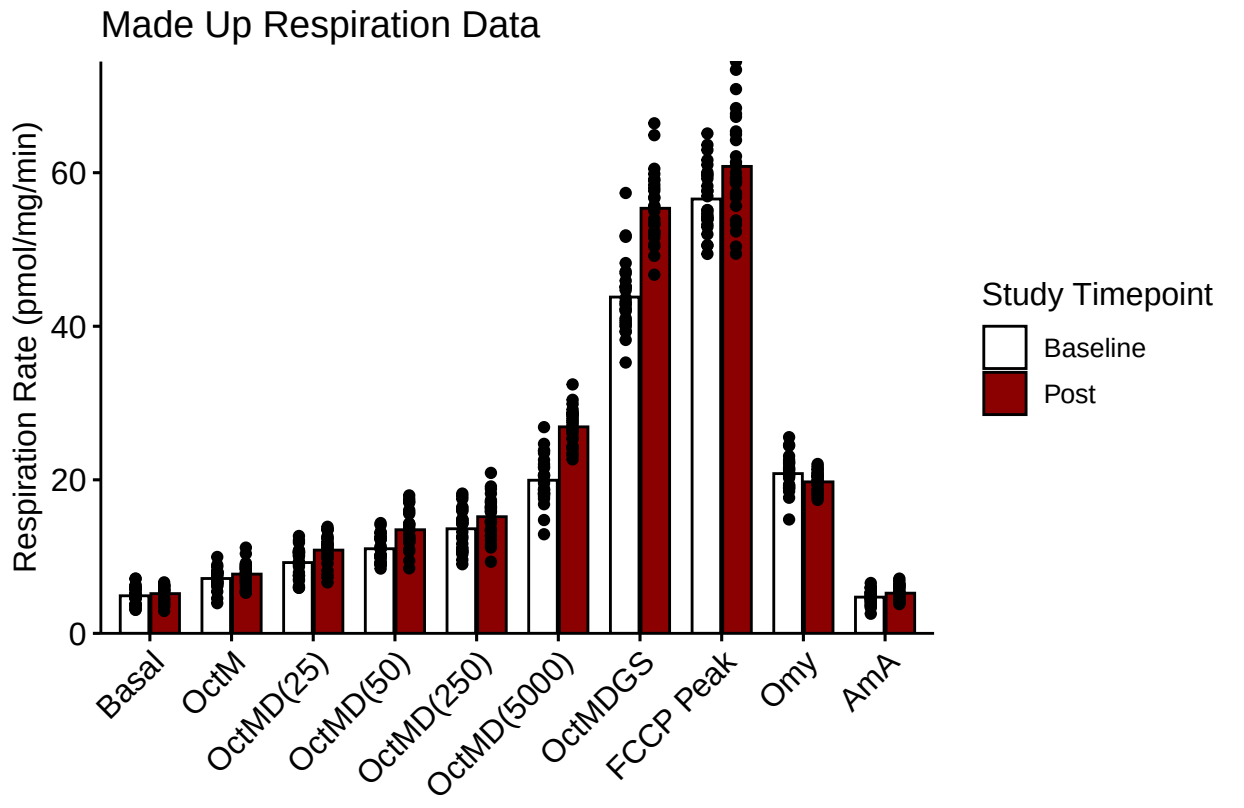


7.6. Adding Individual Data

It may be useful to place individual data points on a plot. Depending on how the data look, though, it can make the graphs appear messy. To do so is very simple, though. We will simply add another layer to our plot by using the `geom_point()` or `geom_jitter()`[12] function to make something like this

```
ggpubr_bar +
  geom_point(aes(x = State, y = Rate, fill = Condition),
             binaxis = 'y', stackdir = "center", dotsize = .5,
             position = position_dodge(0.7), show.legend = F)
```

[12]The two functions essentially perform the same task. However, `geom_jitter()` automatically offsets (jitters) the points so they don't overlay each other. You can do this manually in `geom_points()` by using the `position = "jitter"`. Furthermore, you can change the jitter position with the `height =` and `width =` commands in `geom_jitter()`



You can also use the `geom_dotplot()` function when you have fewer points (not shown), like so:

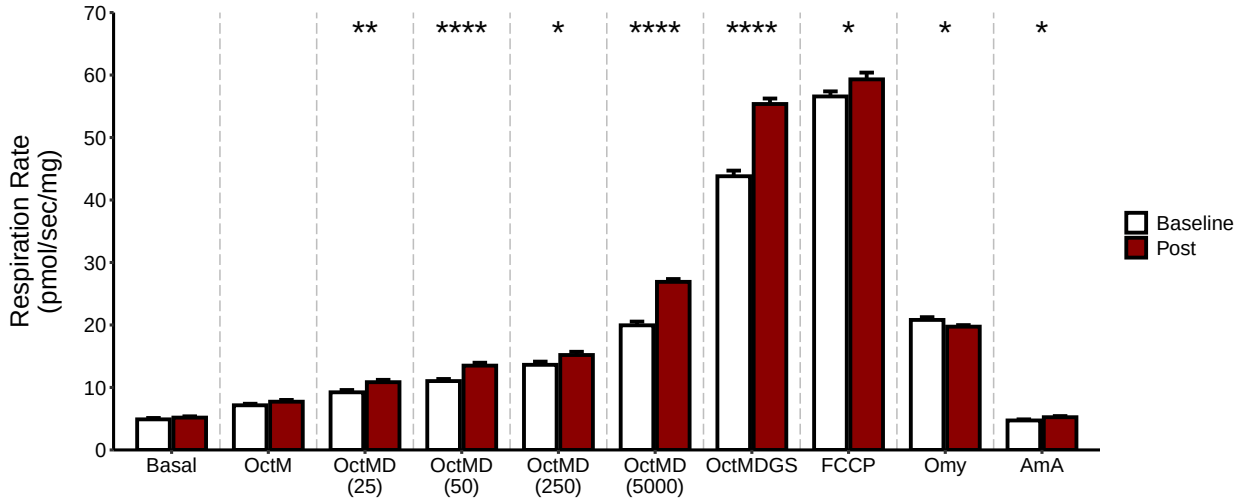
```
ggpubr_bar +
  geom_dotplot(aes(x = State, y = Rate, fill = Condition),
               binaxis = 'y', stackdir = "center", dotsize = .5,
               position = position_dodge(0.7))
```

7.7. My Graphs

You'll notice that I have added some details to my original graphs that are not present in the graphs above. I've included the raw code below to list those changes:

```
ggplot(data_example_long, aes(x=`Respiration State`,
                              y=`Respiration Rate (pmol/sec/mg)`,
                              fill = `Condition`)) +
  #Adds standard error bars#
  stat_summary(fun.data = mean_se, geom = "errorbar",
               position = position_dodge(width=0.75),
               width = 0.3, size = 1.25) +
  #Adds Average bars#
  stat_summary(fun = mean, geom = "bar", width = 0.75,
```

```
              position = position_dodge2(width = 5),
              colour = "black", size = 1.25) +
#add significance
stat_compare_means(aes(group = `Condition`),
                   label = "p.signif", label.y = 65, hide.ns = TRUE,
                   method = "t.test", size = 8) +
#Graph labels (\n creates a new line)#
labs(x = "",
     y = "Respiration Rate\n(pmol/sec/mg)", fill = "Condition") +
#Removes Data point from legend#
guides(fill = guide_legend(override.aes = list(shape = NA))) +
#removes gridlines and top/left borders#
theme_classic() +
#Make y-axis start at 0, set limits#
scale_y_continuous(expand = c(0, 0),
                   limits = c(0,70), breaks=seq(0,70,10)) +
#Make bars different colors#
scale_fill_manual(values = c("white", "dark red")) +
#Adjust fonts#
theme(axis.line = element_line(size = 1),
      axis.ticks = element_line(size = 1),
      axis.text.x = element_text(size = 14,
                                 color = "black"),
      axis.text.y = element_text(color = "black",
                                 size = 14),
      axis.title.y = element_text(size = 18,
                                  margin = margin(t = 0, r = 15,
                                                  b = 0, l = 0)),
      legend.text = element_text(size = 14),
      legend.title = element_blank()) +
#Create dashed lines between Respiration States#
geom_vline(xintercept = c(1.5, 2.5, 3.5, 4.5, 5.5, 6.5, 7.5, 8.5, 9.5),
           color = "grey", linetype = "longdash") +
#Change names#
scale_x_discrete(labels = c("OctMD(25)" = "OctMD\n(25)",
                            "OctMD(50)" = "OctMD\n(50)",
                            "OctMD(250)" = "OctMD\n(250)",
                            "OctMD(5000)" = "OctMD\n(5000)",
                            "FCCP Peak" = "FCCP"))
```



7.8. Saving Graphs

Plots are easily saved in many formats including .png and .svg formats using the `ggsave()` function. To do this, you need to save the plot as an object and signify the file name, and you can customize the size and the background color, too. Here's an example:

```
#Save as a .png
ggsave("RespBoxplot.png", ggpubr_box)

#Save as a .svg with transparent background
ggsave("RespBar.svg", ggpubr_bar, bg = "transparent", width = 4, height = 3)
```

8. Helpful Resources

Below are the resources I have used to compile this document

8.1. Books

8.1.1. For Statistics

- **Discovering Statistics Using R** by Andy Field, Jeremy Miles, and Zoë Field
- **R for Data Science** by Garett Grolemund & Hadley Wickham
- **R Cookbook, 2nd Edition** by JD Long & Paul Teetor
- **An R Companion for the Handbook of Biological Statistics** by Salvatore S. Mangiafico
- **Handbook of Biological Statistics** by John H. McDonald
- **Learning Statistics with R: A tutorial for psychology students and other beginners** by Danielle Navarro
- **R Programming for Data Science** by Roger D. Peng
- **Advanced R** by Hadley Wickham

8.1.2. For Graphing

- **R Graphics Cookbook, 2nd edition** by Winston Chang
- **R Cookbook, 2nd Edition** by JD Long & Paul Teetor
- **ggplot2: Elegant Graphics for Data Analysis** by Hadley Wickham
- **ggplot2 Cheat Sheet**

8.2. Blogs

- **Towards Data Science**

8.3. Forums

- **Data Novia**
- **Stack Exchange**
- **Stack Overflow**
- **STHDA**

8.4. Videos

- **University of Texas Statistics Online Support (SOS)**

8.5. Resources for Packages

- **RDocumentation**
- **CRAN package search**

9. Afterthoughts & Updates

This section is dedicated to other lines of code that may be useful for the exploration of data outside of the Excel files that I have created and mentioned in the first section of the document. Some of these may be more useful if you have other ways of analyzing the data than what I have outlined above. Though I think that the above sections are useful as well.

9.1. Categorizing Continuous Variables

Sometimes it is very useful to take numeric data and group them into ranges or categories. We can easily do this using the `data.table` package. For example, say we have a variable with age ranges, and we want to divy them up into 'Young', 'Middle Age', and 'Elderly'. For this, we need the `'data.table'` package, and we will run this code:

```
setDT(data_example_wide)[ , "Age Group" := cut(Age,
                                    breaks = c(0,45,65,500),
                                    right = FALSE,
                                    labels = c("Young","Middle Age", "Elderly"))]
```

This line takes our data frame ('data_example_wide') and creates a new column titled 'Age Group' based on values in the 'Age' column with the ranges of (0-45, 45-65, 65+), and categorizes them into 'Young', 'Middle Age', and 'Elderly' age groups. Furthermore, we can do something similar if we wish to have more details in our grouping categories, such as combining physical activity and age to get a category such as 'Young Active' by doing the following;

```
data_example_wide <- data_example_wide %>%
  mutate(ActGroup = case_when((Age < 45 & Steps < 5000) ~ "Young Sedentary",
                              (Age < 45 & Steps > 5000) ~ "Young Active",
          (Age > 45 & Steps < 5000) ~ "Elderly Sedentary",
(Age > 45 & Steps > 5000) ~ "Elderly Active"))
```

9.2. Calculations

Sometimes it is easier to perform calculations (such as RCR and sensitivity calculations) directly in R compared to doing it in Excel. I will include some examples that I think are common and useful for that

9.2.1. Respiratory Control Ratio (RCR)

RCR is probably one of the more common calculations used in respirometry analysis. Classicaly defined by Chance, RCR is the ratio of State 3 to State 4 respiration; or, the ratio of maximal respiration to leak respiration. The purpose of the RCR is to determine the "efficiency" of the mitochondria, and how much respiration is "wasted" by proton leak. To create this column, we will simply create a new column based of a mathematical function:

```
data_example_wide$RCR <- (data_example_wide$`OctMD(5000)`/data_example_wide$OctM)
```

The same steps can be used to calculate substrate sensitivity, if you have that data available.

9.3. Enzyme Kinetics

We can also calculate the enzyme kinetics of oxidative phosphorylation, assuming you have performed some sort of titration with difference concentration values of some substrate (such as ADP, like I have in this example). Now, there are a couple of ways to do this. You can, of course, use the Michaelis-Menten equation:

$$v = \frac{V_{max}[S]}{K_M + [S]}$$

This method can be fairly complex and you typically need a dozen or so data points, but I have found an article written by Cathy Huitema (Waterloo Centre for Microbial Research) and Geoff Horsman (Wilfrid Laurier University) that explains how to do this in great detail. Because this resource is conveniently available and is, quite frankly, something I don't have time to do into detail about at the time, I will skip this and go to the much simpler Lineweaver-Burke Method.

9.3.1. Lineweaver-Burke Plot

To perform the Lineweaver-Burk Method, we will create a new data frame with the data that we want to calculate the enzyme kinetics from. Don't forget to also include the subject number and conditon in this

```
library(tidyverse)
kinetics <- gather(Wide[c(1,2,6:10)], "Respiration State",
                           "Respiration Rate (pmol/sec/mg)",
                           `OctMD(25)`:`OctMD(5000)`)
```

Now, we need to create a table with our substrate concentrations based on the data we have collected thus far. Essentially, we will use the `ifelse` function to recode the values 'OctMD(x)' to simply 'x', like so:

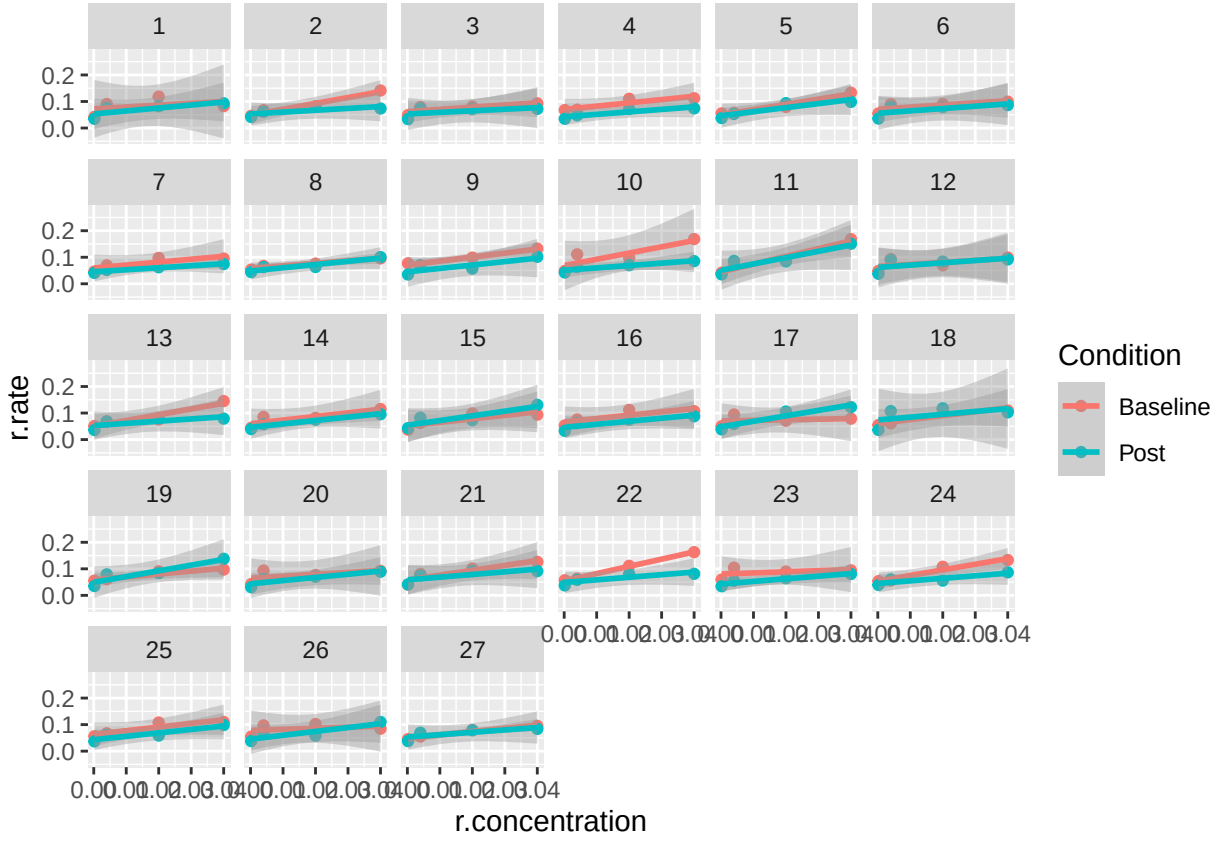
```
#Create the kinetics data
kinetics$Concentration <- ifelse((
  kinetics$`Respiration State` == "OctM"), 0,
  ifelse((
    kinetics$`Respiration State` == "OctMD(25)"), 25,
    ifelse((
     kinetics$`Respiration State` == "OctMD(50)"), 50,
      ifelse((
        kinetics$`Respiration State` == "OctMD(250)"), 250,
        5000))))
```

Now that we have the concentration, we will take the reciprocal of the substrate concentration and the respiration rates

```
#Calculate the reciprocals
kinetics$r.concentration <- 1/kinetics$Concentration
kinetics$r.rate <- 1/kinetics$`Respiration Rate (pmol/sec/mg)`
```

We can plot the resulting data just to visually inspect the data points

```
#Visual inspection
ggplot(kinetics, aes(r.concentration, r.rate, color = Condition)) +
  geom_point() + facet_wrap(~Subject) + stat_smooth(method='lm')
```



Next, we need to perform the linear model using the `lm()` function. The result will give us the linear model for each subject in each condition. Keep in mind that the results we get from this model only gives us the y-intercept and the slope. The y-intercept will be used to calculate the V_{max}, but the Km will need to be calculated by dividing the V_{max} (or, y-intercept) by the slope.

```
#Perform linear modeling on the regression (calculates y-intercept and slope)
lb_model <- (ddply(kinetics, .(Subject, Condition)
                   ,summarise,r.Vmax=lm(r.rate~r.concentration)$coef[1],
                   slope=lm(r.rate~r.concentration)$coef[2]))
```

```
#Calculate x-intercept
lb_model$r.Km <- -(lb_model$r.Vmax/lb_model$slope)
```

And this should give us results that look like this:

Table 30: First Few Rows of lb_model

Subject	Condition	r.Vmax	slope	r.Km
1	Baseline	0.07233	0.6929	-0.1044
1	Post	0.053	1.123	-0.04718
2	Baseline	0.04885	2.199	-0.02221
2	Post	0.053	0.6977	-0.07596
3	Baseline	0.06205	0.83	-0.07476
3	Post	0.05291	0.5967	-0.08866

We will then get the reciprocal of the values once again.

```
lb_model$Vmax <- (1/lb_model$r.Vmax)
lb_model$Km <- -(1/lb_model$r.Km)
```

And then we can recode some of the values and perform our statistical tests:

```
#Recode Condition to Factor

lb_model$Condition <- factor(lb_model$Condition,
                             levels = c("Baseline", "Post"))

#Calculate means
lb_means <- by(list(lb_model$Vmax, lb_model$Km),
        lb_model$Condition, stat.desc, basic = TRUE)


#T-test
vmax_ttest <- t.test(lb_model$Vmax ~ lb_model$Condition, paired = TRUE)

km_ttest <- t.test(lb_model$Km ~ lb_model$Condition, paired = TRUE)
```

9.4. Using Loops to Make Many Graphs

Loops can come in handy if the goal is to create many different graphs that all look alike, but you don't want to go through the hassle of changing every graph individually. In order to do this, it is probably best to create a `for` loop, like so:

```
#Create loop parameters
for(y in names(Wide[5:15])){
```



```
#Create plot here
plot <- ggplot(data = Wide, aes(x = Condition,
                                y = Wide[[y]], fill = Condition)) +
  stat_summary(fun.data = "mean_se", geom = "errorbar",
               position = position_dodge(width = 0.9), width = 0.3,
               na.rm = TRUE, size = 1, color = "black")+
  stat_summary(fun = mean, geom = "bar",
               position = position_dodge(width = 0.9),
               size = 1.1, color = "black") +
  stat_compare_means(aes(group = `Condition`),
                     label = "p.signif", label.y = 65, hide.ns = TRUE,
                     method = "t.test", size = 10, label.x = 1.5) +
  theme_classic2() +
  scale_fill_manual(values = c("white", "dark red")) +
  scale_y_continuous(expand = c(0,0), limits = c(0,80),
                     breaks = seq(0,70,10)) +
  theme(axis.line = element_line(size = 1.1),
        axis.ticks = element_line(size = 1.1, color = "black"),
        axis.title.x = element_blank(),
        axis.title.y = element_text(size = 16,
                                    margin = margin(t = 0, r = 20, b = 0, l = 1),
                                    colour = "black"),
        axis.ticks.x = element_blank(),
        axis.text.y = element_text(size = 11, color = "black"),
        text = element_text(family = "Arial"),
        legend.background = element_rect("NA"),
        plot.background = element_rect(fill = "transparent", color = NA),
        panel.background = element_rect(fill = "transparent", colour = NA),
        plot.margin = unit(c(.5,2.5,.5,.1), "cm"),
        plot.title = element_text(hjust = 0.5, size = 16)
  ) +
  ggtitle(paste(names(Wide[y]))) +
  ylab("Respiration Rate")

#Shows the plots
print(plot)
}
```

RMarkdown doesn't let me show all of the graphs, so you will just have to take my word on the efficacy of this loop.

9.5. Tips for Analysis with Many Factors

In the case of more than two condition variables, the analysis can be pretty tricky – especially when it comes to visualizing the variables. For example, let's say I also want to



know if there are differences between the right or left legs of these participants. The analysis for all of this can be very tricky to navigate, and there isn't a clearly optimal way to do all of this. A few tips that I have used are this:

- Use color when possible. I haven't found a simple way to create plots with textures (i.e. diagonal lines, dots, etc.), so color tends to work best with these.
-

9.6. More Wide and Long Formatting

There are many other ways to convert between long and wide format. The functions I discussed earler, `gather()` and `pivot_longer()` are both functions from the Tidyverse. Another package, reshape2, has the function `melt()` which can be used in a similar manner, and can work with less information. You can also convert to wide format from long format by using the `spread()` or `pivot_wider()` functions in the Tidyverse, or by using the `dcast()` function in reshape2.



10. Functions

After completing the bulk of this document, I think I will try to create functions (and possibly a package) to fulfill many of these analyses automated and much easier to perform than using individual lines of code.