

# **The book of seahtrue**

Vincent de Boer

10/4/23

# Table of contents

<b>Preface</b>	<b>5</b>
Resources . . . . .	5
Free books . . . . .	5
Webr/WASM . . . . .	6
Other info . . . . .	6
Movies: . . . . .	7
R fun . . . . .	7
<b>I   Jump into the water</b>	<b>8</b>
<b>1   Ditching point-and-click and diving into R</b>	<b>10</b>
<b>2   Jumping essentials</b>	<b>12</b>
2.0.1 Find your info online and in documentation . . . . .	12
2.0.2 R and tidyverse documentation . . . . .	12
2.0.3 Style and layout . . . . .	13
2.1 Basic R semantics . . . . .	14
2.1.1 Assignment . . . . .	14
2.1.2 Vectors and lists . . . . .	14
2.1.3 Common semantics . . . . .	16
2.1.4 ~ (the “tilde”) . . . . .	17
2.1.5 + (the plus) . . . . .	17
2.1.6 %>% (the pipe) . . . . .	17
2.1.7 == (equal to) . . . . .	18
2.1.8 aes (aesthetics in ggplot) . . . . .	18
2.1.9 %in% (match operator) . . . . .	19
2.2 Practical tips . . . . .	19
2.2.1 Running your code . . . . .	19
2.2.2 Simple troubleshooting your pipelines and ggplots . . . . .	19
2.2.3 Building your data visualisation step by step . . . . .	20
<b>3   Plotting cars</b>	<b>23</b>
3.1 Exercises . . . . .	28
3.1.1 Adding layers and changing the MTCARS plot . . . . .	28

3.1.2	Fixing common errors . . . . .	33
3.2	Basic R semantics . . . . .	41
3.2.1	Assignment . . . . .	41
3.2.2	Vectors and lists . . . . .	41
3.2.3	Common semantics . . . . .	43
3.2.4	~ (the “tilde”) . . . . .	44
3.2.5	+ (the plus) . . . . .	44
3.2.6	%>% (the pipe) . . . . .	44
3.2.7	== (equal to) . . . . .	45
3.2.8	aes (aesthetics in ggplot) . . . . .	45
3.2.9	%in% (match operator) . . . . .	46
<b>4</b>	<b>Practical tips</b>	<b>47</b>
4.1	Running your code . . . . .	47
4.2	Simple troubleshooting your pipelines and ggplots . . . . .	47
4.3	Building your data visualisation step by step . . . . .	47
4.3.1	Find your info online and in documentation . . . . .	50
4.3.2	R and tidyverse documentation . . . . .	51
4.3.3	Style and layout . . . . .	51
<b>5</b>	<b>Plotting seahorse</b>	<b>53</b>
<b>6</b>	<b>Summary</b>	<b>75</b>
6.0.1	Ditching . . . . .	75
6.0.2	Cars . . . . .	75
6.0.3	Seahorse . . . . .	75
6.1	What we did not learn? . . . . .	76
<b>II</b>	<b>Swim underwater</b>	<b>77</b>
<b>7</b>	<b>Seahtrue functions</b>	<b>79</b>
7.1	Seahtrue read data function . . . . .	81
7.2	Seahtrue preprocess data function . . . . .	84
<b>8</b>	<b>Seahtrue outputs</b>	<b>87</b>
8.1	The purrr map function . . . . .	88
8.2	The seahtrue ouput . . . . .	90
<b>9</b>	<b>Summary</b>	<b>93</b>
9.0.1	Functions . . . . .	93
9.0.2	Outputs . . . . .	93
9.1	What we did not learn? . . . . .	93

<b>III Diving deeper</b>	<b>94</b>
<b>10 Single experiment</b>	<b>96</b>
10.1 Background . . . . .	99
10.2 Low signals . . . . .	106
10.3 Plotting basal and maximal respiration . . . . .	113
<b>11 Multiple experiments</b>	<b>120</b>
 <b>IV Surfing the waves</b>	 <b>126</b>
<b>12 Running Seahorse functions</b>	<b>127</b>
12.0.1 read_xf_plate() . . . . .	127
12.0.2 preprocess_xfplate() . . . . .	138
12.0.3 run_seahorse() . . . . .	142
12.1 run it . . . . .	142
12.2 Bioenergetic space plot . . . . .	145
12.2.1 Buffering power . . . . .	145
12.2.2 Calculations . . . . .	146
12.2.3 Space plot . . . . .	149
12.2.4 Gauge plots . . . . .	152

# Preface

This is the R Seahorse data analysis manual using functions from the Seahtrue package. Its purpose is to demonstrate and educate how to use R for Extracellular Flux analysis. A dedicated data analysis pipeline is used that allows for quality control and advanced plotting of the data.

The manual is designed using a webr/wasm format, so that you can run R in your browser and do not need to install any R or Rstudio software on your computer.

## Note

The manual is targeted to all levels of learning, meaning that also interested learners without any R background or programming knowledge can use this manual

## Tip

To turn your `.asyr` Seahorse Wave file into an excel `.xlsx` file, you can use the Seahorse Wave desktop software or the Seahorse analytics website from Agilent:

<https://seahorseanalytics.agilent.com>

Lots of info regarding Seahorse analysis including how it works and how to run experiments is available from the Agilent website

All things lab - How to run an assay:

<https://www.agilent.com/en/product/cell-analysis/how-to-run-an-assay>

All other Agilent Seahorse info:

<https://www.agilent.com/en/products/cell-analysis/how-seahorse-xf-analyzers-work>

## Resources

### Free books

[Telling Stories with data](#)

*Excellent very complete overview covering basic R coding, communicating science, and statistics*

Good chapters for example:

- Clean and prepare [https://tellingstorieswithdata.com/09-clean\\_and\\_prepare.html](https://tellingstorieswithdata.com/09-clean_and_prepare.html)
- appendix 1: R essentials [https://tellingstorieswithdata.com/20-r\\_essentials.html](https://tellingstorieswithdata.com/20-r_essentials.html)

## R for data Science

*The go-to book when you want to get familiar with R and tidy from Hadley Wickham and others from Posit* <https://r4ds.hadley.nz/>

Nice ones for example:

- ggplot basics <https://r4ds.hadley.nz/data-visualize>
- lubridate date and time basics <https://r4ds.hadley.nz/datetimes>

## Functional programming book

*Nice visualizations of data structures and operations*

- vectors, lists, tibbles <https://dcl-prog.stanford.edu/data-structure-basics.html>
- map and purrr <https://dcl-prog.stanford.edu/purrr-basics.html>

## Fundamentals of Data Visualization

*Everything you want to know about visualizing data using ggplot, with beautiful plots. Also, all code fully available on github*

- associations (scatter plots etc) <https://clauswilke.com/dataviz/visualizing-associations.html>
- uncertainty (error bars and distributions etc) <https://clauswilke.com/dataviz/visualizing-uncertainty.html>

## **Webr/WASM**

webr REPL - a full complete rstudio-like R environment in the browser <https://webr.r-wasm.org/latest/>

Webr/wasm presentation bob rudis - (how I got interested in webr/wasm) - <https://www.youtube.com/watch?v=inpwcTUmBDY>

Webr manual <https://docs.r-wasm.org/webr/latest/>

Webr on github <https://github.com/r-wasm/webr/>

## **Other info**

Learn to purrr, Rebecca Barter - [https://www.rebeccabarter.com/blog/2019-08-19\\_purrr](https://www.rebeccabarter.com/blog/2019-08-19_purrr)

**Movies:**

Purrr and map functions explained by Hadley - <https://www.youtube.com/watch?v=EGAs7zuRutY>

**R fun**

<https://twitter.com/rafamoral/status/1571622591219236864?s=20&t=RJWOSe30-8bbDxgLIamRUQ>

## **Part I**

# **Jump into the water**



In the section, we jump into the water. We learn how to swim in R code and get ourselves familiar with using R for data handling and plotting. We will use first a dataset from R itself and secondly work with a Seahorse data file. There are exercises, with solutions, as well.

# 1 Ditching point-and-click and diving into R

Data analysis in biological and medical sciences was (and still is) dominated by the use of point-and-click tools, like Excel, Prism, SPSS etc. The reasons for this are that it is easy, it is visual, and gets you fast to an outcome. These point-and-click tools are convenient to use and their main asset is that you have a canvas or grid in front of you for dragging-dropping, copy-pasting, typing and calculating. For plotting data, you can choose formatting options by clicking on the features you want to change, again on a canvas that is in front of you on the screen. This canvas style of working is likely what is closest to our natural way of getting things done; putting stuff together with your hands and seeing directly what happens to the stuff is quick and actionable.

The disadvantages of using point-and-click tools in biological and medical sciences are that point-and-click tools are not traceable and can be prone to mistakes. The sequence of clicks and manipulations that have been made to the data was not recorded, which makes the data wrangling process not traceable and proper version control does not exist. Also, you will always work within the limits of the tools that you are using, or how Bruno Rodrigues, the author of the free book [Building reproducible analytical pipelines with R](#), phrased it “... point and click never allow you to go beyond what vendors think you need.” [X-link](#).

Enter R! R is possibly the best, easiest and most accessible tool to use for biologist and like-minded scientists. R is part of or adopted more and more in scholarly programs at academic institutes not only for statistical use but also for other aspects of data science. Other tools like Python and Matlab are also widely used and they have similar benefits as R over point-and-click tools. Matlab however is proprietary software that needs high licence fees from institutions to be able to work with it.

Here you will jump straight to using the **tidyverse** way of working <https://www.tidyverse.org/>. A complete (but extensive) overview of R data science can be found at <https://r4ds.hadley.nz/>. The “R for data science” resource also centers around the **tidyverse** and the **tidy** concept of data handling. Often data handling (organizing the data and tidying it to be able to use it in your downstream workflow) is described as **data wrangling**. As mentioned in the **R for data science book**: “Together, tidying and transforming are called wrangling because getting your data in a form that’s natural to work with often feels like a fight!” <https://r4ds.hadley.nz/intro>. With the **tidyverse** and some level of experience your fight will become less and less over time.

Since the R community is huge, there is also an overwhelming number of resources (like books, tutorials, videos, blog posts, stack-exchange content) that all want to teach, educate and

inform you about R in one way or the other. Also, there are again collections of R resources, and even collections of collections of R resources....

These tutorials and courses have one thing in common, they start of with installing R and Rstudio and learning the software. How nice would it be if we can skip these (often) nasty installations? What if we can skip version updating and package installations and start working with your data right away? That would be amazing! And this is possible with the development of **webr** by George Stagg and colleagues <https://github.com/r-wasm/webr>.

It is R in the browser!

This is so great, because it provides the most convenient, quick and easy way to enter the R world. It is just like you having your Excel, Word and Powerpoint always immediately up and running by a click of a button. Since with R we type in our commands instead of pointing and clicking we are in the era of type-and-click to get your data science done.

This book is completely written using **quarto** and **webr** and allows you to typ in the code and run it right in the browser.

## 2 Jumping essentials

Before you jumop into the water it can be of benefit when you know more about the water. What is the temperature? Is it really cold or just nice and warm? How high is the jump? Do you need to jump first 5 meters from a diving board or can you allready feel the water with your toes. This first chapter will give some basic programming essentials that will allow you to jump easier. Also it can be used as a reference for when you need to make the jump again

### 2.0.1 Find your info online and in documentation

R has so many functions that it is impossible to know everything by heart. So documentation of functions and the internet are always your best friend.

**Stackexchange** is an excellent resource. Almost 90 to 99% of your questions related to how you should use your R and tidy functions has been asked before by others. THE nice thing is that the active coding community put those questions with reproducible code in Stackexchange. More importantly, almost all questions has been accurately answered in multiple ways.

Other resources that come up more often in my search results are either forums on **POSIT community**, **Reddit**, or **Github discussions or issues** can also be usefull, but these are more forum-like comments, with not such a good solvability structure as stackexchange.

Then there are many more resources that somehow scrape the internet and collect basic info. Most of the time the info is correct but too simplistic. Not real issues are tackled. These are sites like **geeksforgeeks**, **datanovia**, **towardsdatascience**, some have better info then others, but most of the time these have commercial activities and in the end want to sell you courses or get your clicks.

### 2.0.2 R and tidyverse documentation

All functions in R and tidyverse are accurately documented. All its arguments are described and especially the **examples** that are given are really helpful. Packages have often even more documentation called **vignettes** that explain certatin topics and contexts on how and when to use the functions.

## 2.0.3 Style and layout

Writing your code benefits from proper readability. Just like we layout our texts, manuscripts and excel data files, we also need a good layout for our code.

```
# NOT VERY READABLE (but runnable )
  ggplot(data=mtcars,   mapping=aes.   (x = mpg,y   = disp,
color = hp,shape =           as.factor(cyl))           ) +geom_point()
```

There are multiple ways to organize your code, I try to adhere to: - short lines (max 60 characters per line) - indent after first line - indent after ggplot - each next function call aligns with the above function - each argument aligns with the previous argument - each ggplot layer gets its own line - I put the x and y aesthetics for ggplot mapping on one line

Other good practices are: - use the package name before a function, like `dplyr::mutate` - use comments to annotate the code, when you put a `#` before it, it is not executed

So here is an example on what not to do and its corrections

```
#NOT GOOD
iris %>%
as_tibble() %>% janitor::clean_names() %>%
filter(species
      %in% c("setosa", "virginica")) %>%
      ggplot(aes(x = sepal_length, y = sepal_width,group = petal_length, color = petal_w
geom_point()+ geom_line() +
      theme_bw(base_size = 16)

#GOOD
iris %>%
  as_tibble() %>%
  janitor::clean_names() %>%
  filter(species %in% c("setosa", "virginica")) %>%
  ggplot(aes(x = sepal_length, y = sepal_width,
            group = petal_length,
            color = petal_width))+
  geom_point()+
  geom_line() +
  theme_bw(base_size = 16)
```

## 2.1 Basic R semantics

When starting using R and tidyverse the new language can be daunting. So here is a short primer of common semantics that are often not directly understood from code.

I took some of these example directly or indirectly from:

<https://uc-r.github.io/basics>

### 2.1.1 Assignment

The most common way of assigning in R is the `<-` symbol. Although the `=` works in the same way, it is reserved by R users for other things. I tend to use it for assigning numbers to constants, and it is used in function arguments

```
#assignment
x <- 1

#is the same as:
x = 1

#but the <- is preferred
```

### 2.1.2 Vectors and lists

A **vector** in R is a collection of items (elements) of the same kind (types). A **list** is a collection of items that can also have different types. We make a vector with `c()` and a list with `list()`. The `c` in `c()` apparently stands for **combine** [link](#)

```
#vectors
x <- c(1, 2, 3)
y <- c("aap", "noot", "mies")
x
y
```

```

#lists
x <- list(1, 2, 3)
y <- list("aap", "noot", "mies", 1, c(22, 23, 25))
x
y

```

Also, R forces that a vector is of one type. You can see that when you specify a vector with numbers and characters eg. `c(1, 2, "1", "2")`. It forces the vector to be of **character** type.

```

#other vector semantics
x <- 1:10

#is the same as
x <- c(1:10)

#is the same as
x <- c(1,2,3,4,5,6,7,8,9,10)

#you can multiply all elements of a vector at the same time
x * 3

# or:
y <- 3
x * y

# or:
x / y

# also adding y to x will add 3 to each element
x + y

# you can also extend or combine two vectors
z <- 20:25

c(x, z)

```

Lists form the basis of all other data than vectors. Dataframes are collections of related

data with rows and columns and unique columns names and row names (or row numbers). `data.frame` is actually a wrapper around the `list` method. Tibbles are the tidyverse equivalent of `dataframes` with some more handy properties over `dataframes`. A 'list' can have named items or not.

```
#a list without named items
my_list <- list(1:10, letters[1:10], LETTERS[1:10])

#a list with named items
my_list <- list(my_numbers = 1:10,
               my_lowercase = letters[1:10],
               my_uppercase = LETTERS[1:10])

#this almost looks like a table, it only is not in a matrix format

#turning the list into a dataframe generates a table
as.data.frame(my_list)

#which is similar to making it a tibble
as_tibble(my_list)

#when the columns are not of the same length the df or tibble
#cannot be generated
my_list_2 <- list(my_numbers = 1:10,
                  my_lowercase = letters[1:10],
                  my_uppercase = LETTERS[1:9])

as.data.frame(my_list_2)
```

### 2.1.3 Common semantics

R language is different from other programming languages, and when starting out learning R there are some rules and common practices.



### 2.1.4 ~ (the “tilde”)

```
#the primary use case is to separate the left hand side
#with the right hand side in a formula

y ~ a*x+ b

#the ~ is also used in the ggplot facet_wrap or facet_grid
#it can be read as "by"
# separate the ggplot "by" cyl
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point()+
    facet_wrap(~cyl)
```

### 2.1.5 + (the plus)

Apart from the simple arithmetic addition + is also used in the ggplot functions. It adds the multiple layers to each ggplot

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point()+
    geom_line()+
    geom_boxplot()+
    labs(title = "Crazy plot")
```

### 2.1.6 %>% (the pipe)

The %>% is used to forward an object to another function or expression. It was first introduced in the `magrittr` package and is now also introduced in base R as the `|>` pipe, which are now identical. See [blogpost](#) for more info.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  mutate(new_column = mpg*cyl) %>%
  filter(new_column > 130)
```

### 2.1.7 == (equal to)

The == is the equal to operator. It is different than = which is used only for assignment.

```
#the equal to is validating whether the left hand side
#is the same as the right hand side and its output is TRUE or FALSE
7 == 7
#generates TRUE wheres
6 == 7
#generates FALSE
```

### 2.1.8 aes (aesthetics in ggplot)

The **aes** is important for telling the ggplot what to plot. **aes** are the aesthetics of the plot that need to be mapped to data. So the ggplot needs **data** and **mappings**.

The **ggplot** acronym is actually coming from the **grammar of graphics**, which is a book "The grammar of graphics" by Leland Wilkinson, and was used by Hadley Wickham to make the **ggplot** package in 2005.

A **ggplot** consists of: - data - aesthetic mappings (like x, y, shape, color etc) - geometric objects (like points, lines etc) - statistical transformations (stat\_smooth) - scales - coordinate systems - themes and layouts - faceting

```
#ggplot basics with one geometric object "geom_point"
#and several aesthetics
ggplot(data = mtcars,
  mapping = aes(x = mpg,
                y = disp,
                color = hp,
                shape = as.factor(cyl))) +
  geom_point()
```

### 2.1.9 %in% (match operator)

This is handy to check and filter specific elements from a vector

```
my_groups <- c("50.000", "100.000", "150.000")

"50.000" %in% my_groups #generates TRUE

#and the other way around
my_groups %in% c("50.000", "100.000")

#this is usefull when filtering specific elements in a tibble
iris %>%
  filter(species %in% c("setosa", "virginica"))
```

## 2.2 Practical tips

### 2.2.1 Running your code

Webr code in the browser can be run as a complete code block by clicking on the **Run code** button when the webr status is **Ready!**, right above the block.

WEBR STATUS

● Ready!

Run code

```
1 library(tidyverse)
2
```

Figure 2.1: Screenshot of a code block that is ready to run

Another option is to select a line of code (or more lines) and press **command** or **ctrl enter**. This will execute only the line or lines that you have selected.

### 2.2.2 Simple troubleshooting your pipelines and ggplots

It happens that your code is not right away typed in perfectly, so you will get errors and warnings. It is good practice to break down your full code block or pipe into parts and observe after which line of code the code is not working properly.

### 2.2.3 Building your data visualisation step by step

Let's take a built-in R dataset `USArrests`. We want to visualize how the relative number of murders in the state Massachusetts relates to the other states with the highest urban population in those state. In the dataset, the `murder` column represents the number of murders per 100.000 residents

```
USArrests
```

```
head(USArrests)
```

```
glimpse(USArrests)
```

```
#please note that the states are listed as rownames. The glimpse does not show the rownames
```

#### Exercise x

Make a plot that addresses the above dataviz problem.

```
USArrests %>%  
  #.....  
  #.....  
  #.....  
  #.....  
  #ggplot.....  
  #.....  
  #etc
```

## **i** HINTS

Hints:

Do the following in your coding:

- `glimpse` at the data and look at the top5 rows using `head()`
- use `tibble::rownames_to_column()` to make a separate column called `states`
- clean the column names using `janitor::clean_names()`
- turn the datatable into a `tibble` using `'as_tibble'`
- take only the the top states by using a filter on the urban population (take it higher than 74)
- plot the data using a `geom_col`
- label the x axis and not the y-axis
- highlight the massachusetts column using a separate `geom_col` layer, were you put a filter on the original data by using in the `geom_col` a call to `'data = . %>% filter(str_detect(states, "Mass"))`. Also give this bar a red color.
- apply a nice theme so that there are only x axis grid lines and no lines for y and x axis.
- Also make sure that x-axis starts at zero

Include all these aspects step by step.

## Solution to Exercise x

```
mUSArrests %>%
  tibble::rownames_to_column(var = "states") %>%
  janitor::clean_names() %>%
  as_tibble() %>%
  filter(urban_pop > 74) %>%
  ggplot(aes( x = murder,
              y = forcats::fct_reorder(states, murder)))+
  geom_col(fill = "grey70")+
  geom_col(data = . %>%
            filter(stringr::str_detect(states, "Mass")),
            fill = "red")+
  labs(y = "",
       x = "number of murders per 100.000 residents")+
  scale_x_continuous(expand = c(0,0))+
  theme_minimal(base_size = 18)+
  theme(panel.grid.major.y = element_blank())
```

## 3 Plotting cars

First, let's make a simple scatter plot. We use a famous dataset that is used in R a lot for educational puposes. This is the `mtcars` dataset (`mtcars` = “Motor Trend Car Road Tests”). See [parameter overview](#) and [documentation](#) for info about the `mtcars` dataset. We can always call this dataset when using R, it is one of many datasets available from base R or tidyverse packages.

First, we will inspect the dataset. For this we will load the tidyverse:

```
library(tidyverse)
```

Once tidyverse is loaded via the `library` call, it is loaded in your current session in your browser, so you do not have to load it each time. Next have a look at the full dataset.

```
mtcars
```

or

```
#if you get an error here,  
# please load the library call to tidyverse  
mtcars %>% glimpse()
```

or

```
mtcars %>% head()
```

or

```
mtcars %>% tail()
```

### **i** The pipe %>% operator

The %>% operator is a huge component of the tidy way of working. In R we now also have a “native” pipe that we can also use. This is the |> operator. For a history of the pipe in R see this [blogpost](#).

```
mtcars %>% head()

#gives the same results as:
mtcars |> head()
```

Let’s select a small part of the data using **select** from the **dplyr** package:

```
mtcars %>%
  select(mpg, disp)
```

Next, make a simple plot with the miles per gallon (**mpg**) and dispension parameters (**disp**) in the **mtcars** dataset.

```
mtcars %>%
  select(mpg, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point(size = 4)
```

This is a very basic plot, without much formatting. Let’s make it prettier!

Add color and bring in a third parameter:

```
mtcars %>%
  #added cyl to the selection here
  select(mpg, disp, cyl) %>%
  ggplot(aes(x = mpg, y = disp,
    #added color to the aesthetics here
    color = cyl))+
    geom_point(size = 4)
```

Here we need to have a look at data-types. The **cyl** parameters is a numerical parameter. GGplot automatically assumes we want a continous scale for this. Instead the **cyl** is more of



a categorical data type there are either 4, 6 or 8 cylinders in each car so we can make the `cyl` parameter categorical like this:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl)))+
  geom_point(size = 4)
```

If you want to have different color you can use one of the many color palettes that are available:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl)))+
  geom_point(size = 4)+
  #I also manually changed the name of the legend here
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")
```

Apart from color you can change the shape of the datapoints:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl)))+
  geom_point(size = 4)+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
             name = "cylinders")
```

GGplot can use different themes for your plots and there are many many options to tweak your plots to the way you like

Let's change titles:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl)))+
  geom_point(size = 4)+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
              name = "cylinders")+
  labs(title = "My cool MTCARS plot",
       x = "miles per gallon",
       y = "dispension")
```

Change the plotting theme and base size of the elements:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl)))+
  geom_point(size = 4)+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
              name = "cylinders")+
  labs(title = "My cool MTCARS plot",
       x = "miles per gallon",
       y = "dispension")+
  theme_bw(base_size = 20)
```

Change the scaling of the axes. It is good practice to plot graphs from zero:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl)))+
  geom_point(size = 4)+
  scale_color_brewer(name = "cylinders",
```

```

        palette = "Set2")+
scale_shape(solid = TRUE,
            name = "cylinders")+
labs(title = "My cool MTCARS plot",
      x = "miles per gallon",
      y = "dispension")+
scale_x_continuous(limits = c(0, NA),
                  expand = c(0,NA))+
scale_y_continuous(limits = c(0, NA),
                  expand = c(0,NA))+
theme_bw(base_size = 20)

```

Now the datapoints at the maxima of the axis are not completely visible so it would be nice that we have some more space:

```

mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
            color = as.factor(cyl),
            shape = as.factor(cyl)))+
  geom_point(size = 4)+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
              name = "cylinders")+
  labs(title = "My cool MTCARS plot",
        x = "miles per gallon",
        y = "dispension")+
  scale_x_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  scale_y_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  theme_bw(base_size = 20)

```

Now we have generated a nice visualisation of our data using **ggplot**. Please note that **ggplot** uses layers and we added each time a different layer of information to the ggplot. If you want you can go wild with ggplot. Please find a nice [overview](#) of visualisations using ggplot, tidy and R from Cedric Scherer. Also the underlying R code is available for those plots.

## 3.1 Exercises

### 3.1.1 Adding layers and changing the MTCARS plot

#### Exercise 1

Give the points in the ggplot some transparency (or opacity), so that individual points are better visible. TIP: use the `alpha` argument it should be a number from 0 to 1.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl)))+
  geom_point(size = 4,
             #type your extra code here:

             )+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
              name = "cylinders")+
  labs(title = "My cool MTCARS plot",
       x = "miles per gallon",
       y = "dispension")+
  scale_x_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  scale_y_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  theme_bw(base_size = 20)
```

#### Solution to Exercise 1

Please note that the `alpha` we added is not part of an aesthetics (`aes`), meaning that the value of the `alpha` is not linked with a parameter in our data.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl)))+
  geom_point(size = 4,
             #type your extra code here:
             alpha = 0.7
             )+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
              name = "cylinders")+
  labs(title = "My cool MTCARS plot",
       x = "miles per gallon",
       y = "dispension")+
  scale_x_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  scale_y_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  theme_bw(base_size = 20)
```

### 💡 Exercise 2

Add a layer that will generate a smooth linear regression line that shows the relation between `mpg` and `disp`. Use the `stat_smooth` command for this.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point(size = 4,
               alpha = 0.7)+
  # enter code here

  labs(title = "My cool MTCARS plot",
        x = "miles per gallon",
        y = "dispension")+
  scale_x_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  scale_y_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  theme_bw(base_size = 20)
```

### Solution to Exercise 2

Please make sure that the ggplot is not separates into groups, if the data is grouped by **color** or **shape** the regression line for each group will be generated.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point(size = 4,
               alpha = 0.7)+
    # enter code here
    stat_smooth(geom = "line",
                method = "lm",
                formula = "y ~ x")+
    labs(title = "My cool MTCARS plot",
         x = "miles per gallon",
         y = "dispension")+
    scale_x_continuous(
      limits = c(0, NA),
      expand = expansion(mult = c(0, 0.1)))+
    scale_y_continuous(
      limits = c(0, NA),
      expand = expansion(mult = c(0, 0.1)))+
    theme_bw(base_size = 20)
```

### 💡 Exercise 3

Use the `facet_wrap` command to make three separate plots for each cylinder.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point(size = 4,
               alpha = 0.7)+

  labs(title = "My cool MTCARS plot",
        x = "miles per gallon",
        y = "dispension")+
  scale_x_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  scale_y_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  theme_bw(base_size = 20)
# enter code here
#(and don't forget to at a plus to the last line)
```

### Solution to Exercise 3

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point(size = 4,
               alpha = 0.7)+
  labs(title = "My cool MTCARS plot",
        x = "miles per gallon",
        y = "dispension")+
  scale_x_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  scale_y_continuous(
    limits = c(0, NA),
    expand = expansion(mult = c(0, 0.1)))+
  theme_bw(base_size = 20)+
  facet_wrap(~cyl)
```



### 3.1.2 Fixing common errors

Below is some code that is not working properly, because of coding semantics mistakes. Can you spot (and fix) the errors?

#### ! Fix error 1

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp
             color = cyl))+
  geom_point(size = 4)
```

#### 🔥 Solution to Error 1

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp, #the error was here
             color = cyl))+
  geom_point(size = 4)
```

Commas are often forgotten, but easily fixed. Within brackets arguments are separated with commas. R also generates an error that is helpful and can point you to the missing

,.

#### ! Fix error 2

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = cyl)) %>%
  geom_point(size = 4)
```

### 🔥 Solution to Error 2

```
mtcars %>%  
  select(mpg, cyl, disp) %>%  
  ggplot(aes(x = mpg, y = disp,  
             color = cyl))+ #the error was in this line  
    geom_point(size = 4)
```

Ggplot layers are added with a + not with the pipe term.

### ! Fix error 3

```
mtcars %>%  
  select(mpg, cyl, disp) %>%  
  ggplot(aes(x = mpg, y = disp  
             color = cyl)) +  
    geom_point(size = 4) +
```

### 🔥 Solution to Error 3

```
mtcars %>%  
  select(mpg, cyl, disp) %>%  
  ggplot(aes(x = mpg, y = disp,  
             color = cyl))+  
    geom_point(size = 4) #the error was in this line
```

Make sure that the end of a layer or line of code is not followed up with a + or %>%.

### ! Fix error 4

This is an error that is not apparent from an error message that R generates for you. However, the code does not give you what you want. The plot should show the `cyl` parameter in different shapes, just like there are three different colors for each level of the `cyl` parameter.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl)),
         shape = as.factor(cyl))+
  geom_point(size = 4)+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
              name = "cylinders")
```

#### Solution to Error 4

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl), #parenthesis error
             shape = as.factor(cyl)))+
  geom_point(size = 4)+
  scale_color_brewer(name = "cylinders",
                    palette = "Set2")+
  scale_shape(solid = TRUE,
              name = "cylinders")
```

The **shape** argument should be included in the aesthetics (**aes**) part of the ggplot

#### Solution to Error 4 using chatGPT

Since chatGPT can also solve basic programming problems <https://doi.org/10.1371/journal.pcbi.1011511>, I tried solving Error 4 also with chatGPT. In my chatGPT session, the following solution was given:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl))) +
  geom_point(size = 4) +
  scale_color_brewer(name = "Cylinders", palette = "Set2") +
  scale_shape_manual(
    name = "Cylinders",
    values = c(1, 2, 3), # Customize shape values as needed
    labels = c("4", "6", "8")
  )
```

chatGPT indeed put the `shape` in the `aes`, but also started making other changes. The remark that chatGPT made was: *“you should correct the syntax error and use `scale_shape_manual()` instead of `scale_shape()`.”*... *“to customize the shape values and labels.”* I didn’t really ask for that, and from there it went all downhill. Let’s call that the “chatGPT loop of death”. Let’s see what happened.

Running the “improved” code showed that there was still no fill, so it did not improve in a way that I wanted.

Next I said *“but now the shapes don’t have a fill”* and chatGPT replied:

*“I see your concern. If you want to have both color and fill for the shapes, you can achieve this by mapping fill within the `aes()` call and using `scale_fill_brewer()` for the fill scale. Here’s the modified code to achieve that:”*

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl),
             fill = as.factor(cyl))) +
  geom_point(size = 4) +
  scale_color_brewer(name = "Cylinders", palette = "Set2") +
  scale_fill_brewer(name = "Cylinders", palette = "Set2") +
  scale_shape_manual(
    name = "Cylinders",
    values = c(1, 2, 3), # Customize shape values as needed
    labels = c("4", "6", "8")
  )
```

This again did not give fill unfortunately. Asking for a different solution: *“the fill is not there, do you have another solution?”*

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl))) +
  geom_point(size = 4) +
  scale_color_brewer(name = "Cylinders", palette = "Set2") +
  scale_fill_manual(
    name = "Cylinders",
    values = c("4" = "red", "6" = "blue", "8" = "green")
  ) +
  scale_shape_manual(
    name = "Cylinders",
    values = c(1, 2, 3), # Customize shape values as needed
    labels = c("4", "6", "8")
  )
)
```

Also did not give me fill.

Another attempt by chatGPT (still not working):

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  mutate(cyl_shape_fill = as.factor(cyl)) %>%
  ggplot(aes(x = mpg, y = disp,
             shape = cyl_shape_fill,
             color = cyl_shape_fill)) +
  geom_point(size = 4) +
  scale_shape_manual(
    name = "Cylinders",
    values = c("4" = 21, "6" = 22, "8" = 23),
    labels = c("4", "6", "8")
  ) +
  scale_color_manual(
    name = "Cylinders",
    values = c("4" = "red", "6" = "blue", "8" = "green")
  )
)
```

Another attempt by chatGPT (still not working):

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             shape = as.factor(cyl),
             color = as.factor(cyl))) +
  geom_point(size = 4) +
  geom_point(aes(fill = as.factor(cyl)), size = 2,
            show.legend = FALSE) +
  scale_color_brewer(name = "Cylinders", palette = "Set2") +
  scale_fill_brewer(name = "Cylinders", palette = "Set2") +
  scale_shape_manual(
    name = "Cylinders",
    values = c(1, 2, 3), # Customize shape values as needed
    labels = c("4", "6", "8")
  )
```

Asking chatGPT to use `scale_shape` and `solid=TRUE` gives:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl))) +
  geom_point(size = 4) +
  scale_color_brewer(name = "Cylinders", palette = "Set2") +
  scale_shape(solid = TRUE,
             name = "Cylinders",
             breaks = c("4", "6", "8"),
             labels = c("4", "6", "8"),
             values = c(15, 16, 17),
             guide = guide_legend(
               override.aes =
                 list(fill = c("red", "blue", "green")))
  )
```

.... but apparently chatGPT doesn't know how to use `scale_shape` (in the first instance), because it `scale_shape` cannot take `values` as an argument. So when I reply with the following *"I get the following error: Error: unused argument (values = c(15, 16, 17))"*, chatGPT goes back to use the `'scale_fill_manual'`.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             color = as.factor(cyl),
             shape = as.factor(cyl))) +
  geom_point(size = 4) +
  scale_color_brewer(name = "Cylinders", palette = "Set2") +
  scale_shape_manual(
    name = "Cylinders",
    breaks = c("4", "6", "8"),
    labels = c("4", "6", "8"),
    values = c(15, 16, 17)
  ) +
  scale_fill_manual(
    name = "Cylinders",
    breaks = c("4", "6", "8"),
    labels = c("4", "6", "8"),
    values = c("red", "blue", "green")
  )
)
```

AND FINALLY we have a nice and simple fill with one legend..... but also lots of unnecessary code. So after a few exchanges to make the code more concise this is what came out of it:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             shape = as.factor(cyl),
             color = as.factor(cyl))) +
  geom_point(size = 4) +
  scale_color_brewer(palette = "Set2",
                    name = "Cylinders") +
  scale_shape_manual(name = "Cylinders",
                    values = c("4" = 15,
                              "6" = 16,
                              "8" = 17)) +
  scale_fill_brewer(palette = "Set2",
                   name = "Cylinders")
```

After asking to use `scale_shape` instead of `scale_shape_manual`, chatGPT generates:

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             shape = as.factor(cyl),
             color = as.factor(cyl))) +
  geom_point(size = 4, stroke = 1) +
  scale_color_brewer(palette = "Set2", name = "Cylinders") +
  scale_shape(solid = TRUE, name = "Cylinders")
```

This works nicely, but chatGPT introduces `stroke = 1`, which is not needed here, so again we have unnecessary code. So after I asked “*can I leave out the stroke argument?*” we get the easiest solution and exactly the same solution as I came up with myself **without chatGPT**.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp,
             shape = as.factor(cyl),
             color = as.factor(cyl))) +
  geom_point(size = 4) +
  scale_color_brewer(palette = "Set2", name = "Cylinders") +
  scale_shape(solid = TRUE, name = "Cylinders")
```

Please note, that when building the ggplot example, I did use google (...off course) to get some solutions, I liked the `scale_shape` and `solid=TRUE` solution that I found, because it made the code so concise and I don't like to type in `values` and `breaks` manually. ChatGPT use in science and coing is just dipping the toe in the water. ChatGPT is likely to better not be used as knowledge database but instead as “**reasoning or inferring agents**” <https://www.nature.com/articles/s41591-023-02594-z>. ChatGPT can produce false information, also described as “**hallucinations**” <https://www.nature.com/articles/d41586-023-00816-5>, which makes it difficult to use it for getting knowledge and facts. That said, it can be used to gain knowledge and learn better coding skills. Here is a nice *quick tips* paper from PLOS computational biology on how to “*harness the power of chatGPT*” <https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1011319>.



## 3.2 Basic R semantics

When starting using R and tidyverse the new language can be daunting. So here is a short primer of common semantics that are often not directly understood from code.

I took some of these example directly or indirectly from:

<https://uc-r.github.io/basics>

### 3.2.1 Assignment

The most common way of assigning in R is the `<-` symbol. Although the `=` works in the same way, it is reserved by R users for other things. I tend to use it for assigning numbers to constants, and it is used in function arguments

```
#assignment
x <- 1

#is the same as:
x = 1

#but the <- is preferred
```

### 3.2.2 Vectors and lists

A **vector** in R is a collection of items (elements) of the same kind (types). A **list** is a collection of items that can also have different types. We make a vector with `c()` and a list with `list()`. The `c` in `c()` apparently stands for **combine** [link](#)

```
#vectors
x <- c(1, 2, 3)
y <- c("aap", "noot", "mies")
x
y
```

```
#lists
x <- list(1, 2, 3)
y <- list("aap", "noot", "mies", 1, c(22, 23, 25))
x
y
```

Also, R forces that a vector is of one type. You can see that when you specify a vector with numbers and characters eg. `c(1, 2, "1", "2")`. It forces the vector to be of **character** type.

```
#other vector semantics
x <- 1:10

#is the same as
x <- c(1:10)

#is the same as
x <- c(1,2,3,4,5,6,7,8,9,10)

#you can multiply all elements of a vector at the same time
x * 3

# or:
y <- 3
x * y

# or:
x / y

# also adding y to x will add 3 to each element
x + y

# you can also extend or combine two vectors
z <- 20:25

c(x, z)
```

Lists form the basis of all other data than vectors. Dataframes are collections of related

data with rows and columns and unique columns names and row names (or row numbers). `data.frame` is actually a wrapper around the `list` method. Tibbles are the tidyverse equivalent of `dataframes` with some more handy properties over `dataframes`. A 'list' can have names items or not.

```
#a list without named items
my_list <- list(1:10, letters[1:10], LETTERS[1:10])

#a list with named items
my_list <- list(my_numbers = 1:10,
               my_lowercase = letters[1:10],
               my_uppercase = LETTERS[1:10])

#this almost looks like a table, it only is not in a matrix format

#turning the list into a dataframe generates a table
as.data.frame(my_list)

#which is similar to making it a tibble
as_tibble(my_list)

#when the columns are not of the same length the df or tibble
#cannot be generated
my_list_2 <- list(my_numbers = 1:10,
                 my_lowercase = letters[1:10],
                 my_uppercase = LETTERS[1:9])

as.data.frame(my_list_2)
```

### 3.2.3 Common semantics

R language is different from other programming languages, and when starting out learning R there are some rules and common practices.

### 3.2.4 ~ (the “tilde”)

```
#the primary use case is to separate the left hand side
#with the right hand side in a formula

y ~ a*x+ b

#the ~ is also used in the ggplot facet_wrap or facet_grid
#it can be read as "by"
# separate the ggplot "by" cyl
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point()+
    facet_wrap(~cyl)
```

### 3.2.5 + (the plus)

Apart from the simple arithmetic addition + is also used in the ggplot functions. It adds the multiple layers to each ggplot

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  ggplot(aes(x = mpg, y = disp))+
    geom_point()+
    geom_line()+
    geom_boxplot()+
    labs(title = "Crazy plot")
```

### 3.2.6 %>% (the pipe)

The %>% is used to forward an object to another function or expression. It was first introduced in the `magrittr` package and is now also introduced in base R as the `|>` pipe, which are now identical. See [blogpost](#) for more info.

```
mtcars %>%
  select(mpg, cyl, disp) %>%
  mutate(new_column = mpg*cyl) %>%
  filter(new_column > 130)
```

### 3.2.7 == (equal to)

The == is the equal to operator. It is different than = which is used only for assignment.

```
#the equal to is validating whether the left hand side
#is the same as the right hand side and its output is TRUE or FALSE
7 == 7
#generates TRUE wheres
6 == 7
#generates FALSE
```

### 3.2.8 aes (aesthetics in ggplot)

The **aes** is important for telling the ggplot what to plot. **aes** are the aesthetics of the plot that need to be mapped to data. So the ggplot needs **data** and **mappings**.

The **ggplot** acronym is actually coming from the **grammar of graphics**, which is a book "The grammar of graphics" by Leland Wilkinson, and was used by Hadley Wickham to make the **ggplot** package in 2005.

A **ggplot** consists of: - data - aesthetic mappings (like x, y, shape, color etc) - geometric objects (like points, lines etc) - statistical transformations (stat\_smooth) - scales - coordinate systems - themes and layouts - faceting

```
#ggplot basics with one geometric object "geom_point"
#and several aesthetics
ggplot(data = mtcars,
  mapping = aes(x = mpg,
                y = disp,
                color = hp,
                shape = as.factor(cyl))) +
  geom_point()
```

### 3.2.9 %in% (match operator)

This is handy to check and filter specific elements from a vector

```
my_groups <- c("50.000", "100.000", "150.000")

"50.000" %in% my_groups #generates TRUE

#and the other way around
my_groups %in% c("50.000", "100.000")

#this is usefull when filtering specific elements in a tibble
iris %>%
  filter(species %in% c("setosa", "virginica"))
```

## 4 Practical tips

### 4.1 Running your code

Webr code in the browser can be run as a complete code block by clicking on the `Run code` button when the webr status is `Ready!`, right above the block.

WEBR STATUS

● Ready!

Run code

```
1 library(tidyverse)
2
```

Figure 4.1: Screenshot of a code block that is ready to run

Another option is to select a line of code (or more lines) and press `command` or `ctrl enter`. This will execute only the line or lines that you have selected.

### 4.2 Simple troubleshooting your pipelines and ggplots

It happens that your code is not right away typed in perfectly, so you will get errors and warnings. It is good practice to break down your full code block or pipe into parts and observe after which line of code the code is not working properly.

### 4.3 Building your data visualisation step by step

Let's take a built-in R dataset `USArrests`. We want to visualize how the relative number of murders in the state Massachusetts relates to the other states with the highest urban population in those state. In the dataset, the `murder` column represents the `number of murders per 100.000 residents`

```
USArrests
```

```
head(USArrests)
```

```
glimpse(USArrests)
```

```
#please note that the states are listed as rownames. The glimpse does not show the rownames
```

### 💡 Exercise x

Make a plot that addresses the above dataviz problem.

```
USArrests %>%  
  #.....  
  #.....  
  #.....  
  #.....  
  #ggplot.....  
  #.....  
  #etc
```



## **i** HINTS

Hints:

Do the following in your coding:

- `glimpse` at the data and look at the top5 rows using `head()`
- use `tibble::rownames_to_column()` to make a separate column called `states`
- clean the column names using `janitor::clean_names()`
- turn the datatable into a `tibble` using `'as_tibble'`
- take only the the top states by using a filter on the urban population (take it higher than 74)
- plot the data using a `geom_col`
- label the x axis and not the y-axis
- highlight the massachusetts column using a separate `geom_col` layer, were you put a filter on the original data by using in the `geom_col` a call to `'data = . %>% filter(str_detect(states, "Mass"))`. Also give this bar a red color.
- apply a nice theme so that there are only x axis grid lines and no lines for y and x axis.
- Also make sure that x-axis starts at zero

Include all these aspects step by step.

## Solution to Exercise x

```
mUSArrests %>%
  tibble::rownames_to_column(var = "states") %>%
  janitor::clean_names() %>%
  as_tibble() %>%
  filter(urban_pop > 74) %>%
  ggplot(aes( x = murder,
             y = forcats::fct_reorder(states, murder)))+
  geom_col(fill = "grey70")+
  geom_col(data = . %>%
           filter(stringr::str_detect(states, "Mass")),
           fill = "red")+
  labs(y = "",
       x = "number of murders per 100.000 residents")+
  scale_x_continuous(expand = c(0,0))+
  theme_minimal(base_size = 18)+
  theme(panel.grid.major.y = element_blank())
```

### 4.3.1 Find your info online and in documentation

R has so many functions that it is impossible to know everything by heart. So documentation of functions and the internet are always your best friend.

**Stackexchange** is an excellent resource. Almost 90 to 99% of your questions related to how you should use your R and tidy functions has been asked before by others. The nice thing is that the active coding community put those questions with reproducible code in Stackexchange. More importantly, almost all questions has been accurately answered in multiple ways.

Other resources that come up more often in my search results are either forums on **POSIT community**, **Reddit**, or **Github discussions or issues** can also be usefull, but these are more forum-like comments, with not such a good solvability structure as stackexchange.

Then there are many more resources that somehow scrape the internet and collect basic info. Most of the time the info is correct but too simplistic. Not real issues are tackled. These are sites like **geeksforgeeks**, **datanovia**, **towardsdatascience**, some have better info then others, but most of the time these have commercial activities and in the end want to sell you courses or get your clicks.

### 4.3.2 R and tidyverse documentation

All functions in R and tidyverse are accurately documented. All its arguments are described and especially the **examples** that are given are really helpful. Packages have often even more documentation called **vignettes** that explain certain topics and contexts on how and when to use the functions.

### 4.3.3 Style and layout

Writing your code benefits from proper readability. Just like we layout our texts, manuscripts and excel data files, we also need a good layout for our code.

```
# NOT VERY READABLE (but runnable )
  ggplot(data=mtcars,   mapping=aes.   (x = mpg,y   = disp,
color = hp,shape =           as.factor(cyl))           ) +geom_point()
```

There are multiple ways to organize your code, I try to adhere to: - short lines (max 60 characters per line) - indent after first line - indent after ggplot - each next function call aligns with the above function - each argument aligns with the previous argument - each ggplot layer gets its own line - I put the x and y aesthetics for ggplot mapping on one line

Other good practices are: - use the package name before a function, like `dplyr::mutate` - use comments to annotate the code, when you put a `#` before it, it is not executed

So here is an example on what not to do and its corrections

```
#NOT GOOD
iris %>%
as_tibble() %>% janitor::clean_names() %>%
filter(species
      %in% c("setosa", "virginica")) %>%
      ggplot(aes(x = sepal_length, y = sepal_width,group = petal_length, color = petal_w
geom_point()+ geom_line() +
      theme_bw(base_size = 16)

#GOOD
iris %>%
  as_tibble() %>%
  janitor::clean_names() %>%
  filter(species %in% c("setosa", "virginica")) %>%
  ggplot(aes(x = sepal_length, y = sepal_width,
```

```
        group = petal_length,  
        color = petal_width)) +  
geom_point() +  
geom_line() +  
theme_bw(base_size = 16)
```

## 5 Plotting seahorse

Now, lets plot some Seahorse data. For this we need to import some here into this session. We use a dataset that was we published in Scientific Reports [Janssen et al.](#). It is data from PBMCs where we followed OCR and ECAR using Extracellular Flux analysis with the XFe96 over time and during that time we injected after three measurement phases FCCP, and after six measurement phases we injected Antimycin/Rotenone (AM/Rot). The data is available from [github](#).

```
library(tidyverse)

root_srcfile <-
  "https://raw.githubusercontent.com/vcjdeboer/"
repository_srcfile <-
  "seahtrue/main/inst/extdata/"

#download file and rename to "VB.xlsx"
download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    "20191219 SciRep PBMCs donor A.xlsx"),
  "VB.xlsx")

#read xlsx file
xf<-readxl::read_xlsx("VB.xlsx", sheet = "Rate")

xf %>% glimpse()
```

As you can see from the `glimpse`, the data table that we have now (we call it a `tibble` in tidy language), contains 7 columns; `Measurement`, `Well`, `Group`, `Time`, `OCR`, `ECAR`, `PER`. The data is allready nice and tidly organized in the `Rate` sheet of the excel file that we have loaded. The file was generated in the Wave Agilent software and directly comes from exporting the Seahorse data to `xlsx`.

I prefer to use lower case column names without any spaces, so for these column names we have to turn them into lower case first. We use some easy functions from the `janitor` package

for this.

```
webr::install("janitor")
xf %>% janitor::clean_names()
```

Next, we can start plotting data using ggplot. Let's introduce the **filter** command from **dplyr**. Whereas **select** is there to select columns, **filter** is there to select rows. So let's filter the rows for the group with is labeled "200.000" (200.000 cells/per well) and the "Background" group.

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  glimpse()
```

### The filter command

Filtering data is selecting the rows based on some arguments. You need some to understand some semantics here. For filtering based on multiple conditions we use **group %in% c("200.000", "Background")**, for filtering based on a single condition we can use **group == "200.000"**. The **%in%** operator is used to match two items.

```
1 %in% c(1,2,3,4,5) #is TRUE

[1] TRUE

# just like
1 == 1 #is TRUE

[1] TRUE

#the reverse is also possible
c(1,2,3,4,5) %in% 1

[1] TRUE FALSE FALSE FALSE FALSE

#is TRUE FALSE FALSE FALSE FALSE
```

```

#Try the 1 = 1 here as well

1 = 1

#the = operator is reserved for assignment
#just like the <- operator
# == is used for comparison


#say that we have the following vector
#( = groups in experiment)
group <- c("Background", "50.0000",
           "100.000", "150.000",
           "200.000", "250.000",
           "300.000")

#we can do the same without typing the
#names by hand like this:
group <- xf %>%
  pull(Group) %>%
  unique()

#then

c("200.000", "Background") %in% group

#generates TRUE TRUE

group %in% c("200.000", "Background")

#generates:
#TRUE FALSE FALSE FALSE TRUE FALSE FALSE

```

Thus the `group %in% c("200.000", "Background")` statement in the filter function above tells which group items to use. For 200.000 there is match (TRUE), but for 100.000 there is not a match (it is FALSE).

Now that we know how to filter we can use the filtered data to make the ggplot.

```

xf %>%

```

```

janitor::clean_names() %>%
filter(group %in% c("200.000", "Background")) %>%
ggplot(aes(x = time, y = ocr))+
geom_point()

```

That plot is not so informative. Let's make it prettier. First, add a line plot:

```

xf %>%
janitor::clean_names() %>%
filter(group %in% c("200.000", "Background")) %>%
ggplot(aes(x = time, y = ocr,
#the group command in aes is important for lines
#ggplot wants to know how to connect dots
          group = well,
          color = group))+
geom_point()+
geom_line()

```

Next, change colors:

```

xf %>%
janitor::clean_names() %>%
filter(group %in% c("200.000", "Background")) %>%
ggplot(aes(x = time, y = ocr,
          group = well,
          color = group))+
geom_point()+
geom_line() +
scale_color_brewer(palette = "Set1")

```

Change theme and text size:

```

xf %>%
janitor::clean_names() %>%
filter(group %in% c("200.000", "Background")) %>%
ggplot(aes(x = time, y = ocr,
          group = well,
          color = group))+

```



```
geom_point()+
geom_line() +
scale_color_brewer(palette = "Set1")+
theme_bw(base_size = 16)
```

Add titles:

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  scale_color_brewer(palette = "Set1")+
  labs(subtitle = "200.000 cells per well vs Background",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
```

This is a very nice plot. It shows all OCR curve for each well for the 200.000 and the background groups. The information that is now not in the plot is which line matches to which well.

Exercise We can color each line (but there are too many wells so it will not be nice!). \* Change this in the above code `color = well` instead of `color = group`. \* You will notice that there are not enough colors in the `brewer` palette `Set1`, so you go back to the default coloring by deleting the `scale_color_brewer` line as well. Use the `#` to comment out the line. \* Now notice that the legend is huge and not completely visible, again indicating that this is not the way to go

Instead, we can try to label the lines. The `ggrepel` package is not yet available in `webr`, so for now we have to use the `geom_text` or `annotate` commands from `ggplot`. The benefit of `ggrepel` is that it automatically prevents text overlap.

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr,
```

```

      group = well,
      color = group)) +
  # here are the labels using geom_text
  geom_text(data = . %>%
    filter(time ==
      max(time)) %>%
    filter(ocr ==
      min(ocr)),
    aes(label = well),
    vjust = 2,
    hjust = 1) +
  geom_text(data = . %>%
    filter(between(time, 15, 20)) %>%
    filter(ocr == max(ocr)),
    aes(label = well),
    vjust = -0.3,
    hjust = 1) +
  geom_point() +
  geom_line() +
  scale_color_brewer(palette = "Set1") +
  labs(subtitle = "200.000 cells per well vs Background",
    x = "time (minutes)",
    y = "OCR (pmol/min)") +
  theme_bw(base_size = 16)

```

Although we now labeled lines that are at the minimum and maximum OCR, this is only useful for this one plot in these conditions. The position of the label is tweaked based on this specific plot, making this not such a quick solution to our problem.

### **i** Subsetting of data within the ggplot commands

In the above ggplot commands, we included the `geom_text`, but we only used a subset of the full data for this geom. We use the `.` (dot) operator to get the original data (so in our case the filtered data that went into the ggplot), and piped that into another two filters. Basically we do the following, but then within one layer of the ggplot:

```

xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  filter(time == max(time)) %>%
  filter(ocr == min(ocr))

#and we also use the between function from dplyr
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  filter(between(time, 15, 20)) %>%
  filter(ocr == max(ocr))

```

Thus here we are filtering all the way to getting only one row of the full dataset. The well name “C08” or “B08” is then given to the `label` argument of `geom_text`.

Let’s do some more layout adjustments. Although the `theme_bw` gives a basic plotting layout, we often want to change the formatting. There are again great resources for this, for example this one: <https://ggplot2.tidyverse.org/articles/faq-customising.html>, but we explain the basics here. By giving options to the `theme` function we can change specific elements of a ggplot.

For example, if we want to change the text size of the axis title (or leave it blank), we give arguments to the `axis.title` options. Also please note the `rel(1.2)` argument which means relative 1.2 times higher than `base_size`. I think it is good practice to use the `rel` here instead of absolute numbers.

```

xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr))+
  geom_point()+
  labs(x = "time (min)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size= 16)+
  theme(
    axis.title.x = element_blank(),
    axis.title.y = element_text(size = rel(1.2))
  )

```

Change the `rel 1.2` to `0.5` in the above code and see what happens.

Next, we change the grid lines

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr))+
  geom_point()+
  labs(x = "time (min)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size= 16)+
  theme(

    panel.grid.minor = element_line(color = "red"),
    panel.grid.major.x = element_line(color = "blue"),

  )
```

Next, we change the orientation of the x axis labels.

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr))+
  geom_point()+
  labs(x = "time (min)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size= 16)+
  theme(
    axis.text.x = element_text(
      angle = 45,
      vjust = 1, # vertical alignment
      hjust = 1, # horizontal alignment
      size = rel(2))
  )
```

We can also try to use the **ggiraph** package. This brings in some nice interactivity into the plot. Since we are now working with the plot in a browser, this can be very handy. Also if we want to publish the plot as html and not a plain PDF this can be usefull. **ggiraph** is unfortunately also not available for wasm/webr since one dependent package is not available **uuid**, and I also can't get it to run via quarto.....

So let's try something else in a couple of exercises.

### 💡 Exercise 1

Add three vertical lines to the plot. You can use the `geom_vline` command with `xintercepts` set at 15, 33 and 48; so that the the line is approximately at the injection time point. Also give it a shade of grey, eg. `grey40`.

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  # add code here

  scale_color_brewer(palette = "Set1")+
  labs(subtitle = "200.000 cells per well vs Background",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
```

## Solution to Exercise 1

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
            color = "grey40")+
  scale_color_brewer(palette = "Set1")+
  labs(subtitle = "200.000 cells per well vs Background",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
```

## Exercise 2

Now add the injection labels. Use the **annotate** command and

```

xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
             color = "grey40")+
  # example annotate
  annotate("text", x = 0, y = 155,
          label = "init", color = "grey40",
          hjust = 1, vjust = -0.1,size = 4, angle = 90)+
  # add other annotates here

  scale_color_brewer(palette = "Set1")+
  labs(subtitle = "200.000 cells per well vs Background",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)

```

## 🔥 Solution to Exercise 2

```
xf %>%
  janitor::clean_names() %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(0,15,33,48),
             color = "grey40")+
  annotate("text", x = 0, y = 155,
           label = "init", color = "grey40",
           hjust = 1, vjust = -0.1, size = 4, angle = 90)+
  annotate("text", x = 15, y = 155,
           label = "fccp", color = "grey40",
           hjust = 1, vjust = -0.1, size = 4, angle = 90)+
  annotate("text", x = 33, y = 155,
           label = "am/rot", color = "grey40",
           hjust = 1, vjust = -0.1, size = 4, angle = 90)+
  annotate("text", x = 48, y = 155,
           label = "monensin", color = "grey40",
           hjust = 1, vjust = -0.1, size = 4, angle = 90)+
  scale_color_brewer(palette = "Set1")+
  labs(subtitle = "200.000 cells per well vs Background",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
```

## 💡 Exercise 3

Use the `facet_wrap` command to plot all groups (except background) in separate plots and in each plot show the wells. First, we will need to filter away the background data. Instead of selecting all groups we need it is better and easier to filter out the background data using `filter(group != "Background")`. The `!=` means “is not” this is the reverse of the `==` operator.

Next, add the `facet_wrap` command to the ggplot. I prefer to do that always at the bottom, so that I can easily see if a plot is wrapped.



```

xf %>%
  janitor::clean_names() %>%
  #change this line:
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
             color = "grey40")+
  scale_color_brewer(palette = "Set2")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
#add facet wrap here
#and do not forgot to add a "+" in previous line

```

### Solution to Exercise 3

```

xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(0,15,33,48),
             color = "grey40")+
  scale_color_brewer(palette = "Set2")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)+
  facet_wrap(~group)

```

#### 💡 Exercise 4

The plot in exercise 3 looks great already, but the order of the plots is important! We would like to see it go from low to high OCR. We can fix that using the **forcats** package commands. A nice and quick way to sort is based on the name of the group. Important to realize is that the **Group** column in the **XF** data are characters and not numbers. That is also the reason why that does not sort in the most natural way. It is sorted based on the first character, thus the “50.000” group comes last. If we would change the “group” column to **double** (that is a number format), it would sort better, but also your group name will change because it will recognize the . as a decimal operator. So it is better to leave the group names as they are and do it differently.

In comes **forcats**, you can relevel and reorder the crap out of your data in the ggplot! We often do the releveling at the point where you use your parameter, without making any changes the type of the columns. So that means you can use `~fct_reorder(group, group)` in the `facet_wrap` instead of only `~group`.

Please note that **fct\_reorder** first argument is the parameter that you plot or need, and the second argument is the parameter that is used for sorting the data. In our case now that is the same, both are “group”, but we also need to add something else. If we would do it like this there will be no difference from when ggplot takes `facet_wrap` only takes `~group`. Thus we can make the second argument into a number by using `as.double`.

```
xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
             color = "grey40")+
  scale_color_brewer(palette = "Set2")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)+
  facet_wrap(~group)
#change the facet_wrap to
#sort by number using fct_reorder and as.double
```

Also, try in the above code what happens if you: - only use `as.double` in the `facet_wrap`

- change the type of data to **double** for the group column

#### 🔥 Solution to Exercise 4

```
xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(0,15,33,48),
            color = "grey40")+
  scale_color_brewer(palette = "Set2")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)+
  facet_wrap(~fct_reorder(group, as.double(group)))
```

#### 💡 Exercise 5

Now that the **facet\_wrap** is sorted nicely, we would also like to have the legend sorted nicely. Use the same **fct\_reorder** trick to reorder the color legend.

```

xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+ #change here
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
             color = "grey40")+
  scale_color_brewer(palette = "Set2")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
  #take solution from previous exercise
  #to have facet_wrap sorted here

```

If you didn't already change the title of the legend, do that as well. You can specify the name of the legend manually using the `name` argument in the `scale_color_brewer` command.

### Solution to Exercise 5

```
xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = fct_reorder(group, as.double(group))))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
             color = "grey40")+
  scale_color_brewer(palette = "Set2",
                    name = "group")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)+
  facet_wrap(~fct_reorder(group, as.double(group)))
```

### Exercise 6

Please change the `facet_wrap` command so that the y-axis is not fixed for all groups. Make the output so that each individual plot has its own y-axis scale.

```

xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
             color = "grey40")+
  scale_color_brewer(palette = "Set2")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
#take solution from previous
#and adjust that facet_wrap

```

## Solution to Exercise 6

```

xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  ggplot(aes(x = time, y = ocr,
             group = well,
             color = fct_reorder(group, as.double(group))))+
  geom_point()+
  geom_line() +
  geom_vline(xintercept = c(15,33,48) ,
             color = "grey40")+
  scale_color_brewer(palette = "Set2",
                    name = "group")+
  labs(subtitle = "OCR of increasing cell densities",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)+
  facet_wrap(~fct_reorder(group, as.double(group)),
            scales = "free_y")

```

### 💡 Exercise 7

Now, it is up to you to build a whole ggplot using the XF data. Instead of plotting time vs OCR, now plot cell density vs maximal capacity. For this you need to know some stuff.

1. we define maximal capacity as the OCR at measurement 4
2. we should filter out the “Background” group
3. we should convert the group names to numbers
4. we can also add the mean of all wells for each group by using: `stat_summary()`

```
xf %>%  
  janitor::clean_names()  
  #filters and ggplot here
```

### 🔥 Solution to Exercise 7

```
xf %>%  
  janitor::clean_names() %>%  
  filter(group != "Background") %>%  
  filter(measurement == 4) %>%  
  ggplot(aes(x = as.double(group)*1000, y = ocr))+  
    geom_point()+  
    stat_summary(fun = "median",  
      colour = "red",  
      size = 16,  
      shape = "-",  
      geom = "point")+  
    scale_x_continuous(  
      limits = c(0, NA),  
      expand = expansion(mult = c(0, 0.1)))+  
    scale_y_continuous(  
      limits = c(0, NA),  
      expand = expansion(mult = c(0, 0.1)))+  
    labs(subtitle =  
      "Maximal capacity at different cell densities",  
      x = "cell density (#cells)",  
      y = "OCR (pmol/min)")+  
    theme_bw(base_size = 16)
```

### 💡 Exercise 8

The previous plot showed the data from individual wells as well as the median for that group. You can also calculate the `median` before plotting using the `dplyr summarize` command. You can find `summarize` info here: <https://dplyr.tidyverse.org/reference/summarise.html>

```
xf %>%
  janitor::clean_names()
```

### 🔥 Solution to Exercise 8

```
xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  filter(measurement == 4) %>%
  summarize(median = median(ocr), .by = group)%>%
  ggplot(aes(x = as.double(group)*1000, y = median))+
  geom_point()+
  geom_line()+
  labs(x = "cell density",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
```

### 💡 Exercise 9

We can also perform a linear regression on the maximal capacity at different densities. For this we can use the `geom_smooth` command. The arguments should be `method = "lm"` and `formula = y~x`.

```
xf %>%
  janitor::clean_names()
#use your code from exercise 7
```



## 🔥 Solution to Exercise 9

```
xf %>%
  janitor::clean_names() %>%
  filter(group != "Background") %>%
  filter(measurement == 4) %>%
  #filter(str_detect(well, "A|H")) %>%
  ggplot(aes(x = as.double(group)*1000, y = ocr))+
    geom_point()+
    stat_summary(fun = "median",
      colour = "red",
      size = 16,
      shape = "-",
      geom = "point")+
    geom_smooth(method = "lm",
      formula = y~x)+
    scale_x_continuous(
      limits = c(0, NA),
      expand = expansion(mult = c(0, 0.1)))+
    scale_y_continuous(
      limits = c(0, NA),
      expand = expansion(mult = c(0, 0.1)))+
    labs(subtitle =
      "Maximal capacity at different cell densities",
      x = "cell density (#cells per well)",
      y = "OCR (pmol/min)")+
    theme_bw(base_size = 16)
```

Observe also what the difference is when only using the data from row A and H. You can uncomment the line in the above code. Please note I use the very usefull `str_detect` function for this from the `stringr` package that is also in the `tidyverse`.

## 💡 Exercise 10

Next, you can decide yourself what you want to plot. Have a **glimpse** at the data and think of another important visualisation that you want to make using all the tools that you have learned so far, or the tools that you found on the internet.

```
xf %>%  
  janitor::clean_names() %>%  
  glimpse()  
  
xf %>%  
  janitor::clean_names()
```

## 6 Summary

### 6.0.1 Ditching

- The benefits of using R over point-and-click software for data analysis in biological and biomedical sciences are that it is open-source, it has a wide and diverse community with a huge number of resources, it is relatively easy to learn, and it offers workflows that is very well suited for doing reproducible and responsible data analysis.
- The tidyverse offers advantages over base R. It offers an intuitive way of coding with functional names and tidy data handling and coding in mind
- R in the browser offers easy access to R without installing software

### 6.0.2 Cars

- General R coding and execution of code
- How to look at data tables: `head`, `tail`, `glimpse`
- The pipe operator `%>%` or `|>`
- Making factorial data using `as.factor`
- the `dplyr` function `select`
- basic `ggplot` functions using `aes` aesthetics and geoms such as `geom_point`
- adding `color` and `shape` and using `scale_brewer_manual` and `scale_shape`
- improving layout; `theme_bw`, `base_size` and `labs`
- using chatGPT for coding improvements

### 6.0.3 Seahorse

- Loading data and working with typical Seahorse data
- Using `janitor` `clean_names`
- Using the `dplyr` function `filter`
- Using the `%in%` operator
- Changing the layout of ggplots using `theme` elements and arguments.
- adding text to ggplot using `geom_text` and `annotate`
- Adding lines to ggplot using `geom_vline`
- nesting pipes in ggplot function for subsetting data
- Using `facet_wrap` to make multiple similar plots from one datatable

- Using the `forcats fct_reorder` function
- Changing data formats to numbers using `as.double`
- Using the `dplyr summarize` function
- Using `stat_summary` to compute means or medians in ggplots
- Using `geom_smooth` to make regression lines

## 6.1 What we did not learn?

- base R functions and how to address data in base R, eg `xf$OCR[xf$Group == "Background"]` and `xf$Well[10]`
- other important tidyverse functions, like `pivot_wider`, `pivot_longer`,
- more complicated functions like the `map` function from the `purrr` package
- other simple ggplot geoms, like `geom_bar`, `geom_boxplot`, `geom_density`
- how to save images and plots for using them in other software

## **Part II**

# **Swim underwater**

Now we will go for a bit more than just swimming in R code. We will go underwater and swim in shallow water to look at those seahorses. In this section we will introduce functions from the **seahtrue** package and explore the output of **seahtrue** functions.

## 7 Seahtrue functions

First, let's see what a **function** is in R. In the previous section, we used functions that changed our data, eg. `filter`, `select` and `clean_names`. Functions are just a bunch of code lines using any code and other functions you want to accomplish a task. Here are some formal definitions:

Functions are “self contained” modules of code that accomplish a specific task. Functions usually take in some sort of data structure (value, vector, dataframe etc.), process it, and return a result. [link](#)

A function in R is an object containing multiple interrelated statements that are run together in a predefined order every time the function is called. [link](#)

Functions take **arguments**, these are used as input for your function.

```
library(tidyverse)

#make the function
change_mtcars_cyl_to_x <- function(x){
  mtcars %>%
    mutate(cyl = x)
}

#call the function
change_mtcars_cyl_to_x(8)
```

Please note that it is good practice to use verbs in function names and address in the name what a function is doing. In our case we define a function `change_mtcars_cyl_to_x` because this is exactly what this function is doing.

### Exercise 1

The `change_mtcars_cyl_to_x` function in the above code is a nonsensical function, because you never want to change a column to one specific value. Let alone a specific column named `cyl`. Also, you don't have to write a whole separate function for this, you

can also directly use the `mutate` function from `dplyr`. Write the code without using the `change_mtcars_cyl_to_x` function, but achieve the same result.

```
# use mtcars and mutate from dplyr
```

### Solution to Exercise 1

```
mtcars %>%  
  mutate(cyl = 8)
```

Please note that you can change the data in a column to anything you want. R is very very flexible in datatypes (compared to other languages). So if you would do this:

```
mtcars %>%  
  mutate(cyl = "eight")
```

that is also fine.

The data types are given when you `glimpse` the data.

```
mtcars %>% glimpse()
```

You see that all columns are `<dbl>` which stands for `double`, which is a numeric data type. `integer` is another common numerical datatype.

When you replace the `cyl` column data with `"eight"`, which is of the `character` type, the data type will change.

```
mtcars %>%  
  mutate(cyl = "eight") %>%  
  glimpse()
```

This is all fine in R. **Important to note** though is that a column can have only one data type. In Excel you can define each cell a different data type, but in R that is not possible. So it is either a column of type `character` or `double` in our case.

Now let's extend the function a bit to have two arguments:



```

#make the function
change_df_cyl_to_x <- function(df, x){

  df %>%
    mutate(cyl = x)

}

#call the function
change_df_cyl_to_x(mtcars, 7)

#or call with pipe
mtcars %>% change_df_cyl_to_x(7)

#you can also see this as:
mtcars %>% change_df_cyl_to_x(., 7)

```

Although the function is a bit more general, because we can now also input the tibble that we want to change, it is still not very useful in practice. A single `mutate` function is preferred to be used here. On the other hand it is an easy example to demonstrate what a function is and how it works.

## 7.1 Seahtrue read data function

Now let's start with the functions from the **seahtrue** package. Since **seahtrue** is not available for **webr**, we need to load in the functions manually. The first thing we do is to read data from the excel file that is generated using the Wave software. In the previous section we only loaded in one sheet of that datafile **Rate**, but the **seahtrue** package takes all data and organizes it nicely (and tidily) into a **nested tibble**.

One of the functions is the `get_xf_raw`. It reads the Raw sheet from the excel file.

```

get_xf_raw <- function(fileName){

  xf_raw <- readxl::read_excel(fileName, sheet = "Raw")

}

```

The argument `fileName` and its location is important. If we work with data input for your scripts, you need to be precise where your files are located. On windows and mac computers and with cloud services and web apps, it can get confusion what this exact location is of

your files, either locally or on network or cloud. Sometimes they are on the desktop or in a documents folder, or they can live on a network drive. Properly addressing these files can be difficult because the **full path** is not always known. It is often recommended to put data files in the Rstudio project folder that you work with, so that you can work with relative paths from your project root directory. This is another example of good practice.

Using `webr/wasm` we do a similar thing, we download the file to our local drives. On my computer when I download a file it goes into the `/home/web_user/` directory. Apparently this is my working directory in my `webr/wasm` sessions. Since it is the working directory, everything that is in there is directly accessible with only the filename. You don't need a full path name like `C:\Users\MyName\Desktop\R\projects\blabla\datafolder\data\`. So for the `get_xf_raw` function to work we first download the file into our session working directory and then we call the `get_xf_raw` function.

```
library(tidyverse)

#set file source
root_srcfile <-
  "https://raw.githubusercontent.com/vcjdeboer/"
repository_srcfile <-
  "seahtrue/main/inst/extdata/"

#download file and rename to "VB.xlsx"
download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    "20191219 SciRep PBMCs donor A.xlsx"),
  "VB.xlsx")

#define the function
get_xf_raw <- function(fileName){

  xf_raw <- readxl::read_xlsx(fileName, sheet = "Raw")

}

#set the file name variable
fileName <- "VB.xlsx"

#read xlsx file
xf<-get_xf_raw(fileName)
```

```
#glimpse at the xf tibble
xf %>% glimpse()
```

Please note that we also tell our session what the `get_xf_raw` function is here. Basically, we assign the code lines to `get_xf_raw`, so when we call `get_xf_raw` with its argument, these lines of code are run.

#### **i** using functions from packages

In the `get_xf_raw` function we call the `read_xlsx` function. However, we also include the package from which the function is from, like this `readxl::read_xlsx`. This has two advantages. First, it is good practice to show where your function comes from, because sometimes a function name is used in multiple different packages. For example, the `filter` function we often use is from `dplyr`, but the `stats` package also uses the `filter` function but then in a slightly different way. Second, when using the `package::function` annotation you don't have to load the package using the `library` command.

In other languages, such as python, you are required to also include the library, when calling a function from that library [https://www.rebeccabarter.com/blog/2023-09-11-from\\_r\\_to\\_python](https://www.rebeccabarter.com/blog/2023-09-11-from_r_to_python).

Apart from reading the `Raw` data sheet there are a couple more functions to read the other data and meta info.

```
#raw data
get_xf_raw()

#rate data
get_xf_rate()

#normalization data
get_xf_norm()

#buffer factors
get_xf_buffer()

#injection info
get_xf_inj()

#pH calibration data
get_xf_pHcal()
```

```

#O2 calibration data
get_xfO2cal()

#flagged wells
get_xf_flagged()

#assay info
get_xf_assayinfo()

```

Furthermore, there is a function that combines all functions as above and outputs them in a list:

```

#raw data
read_xf_plate()

```

The input argument for all is the filename or path of the input xlsx data file.

## 7.2 Seahtrue preprocess data function

Following reading the data, the data needs to be processed to a tidy format so that it can be easily used for downstream processing.

For example, there is a function which changes the columns from the input file data into names without capitals and spaces. The `clean_names` from the `janitor` package can also be used, but in this case we wanted to be a bit more precise on what the names should be.

```

rename_columns <- function(xf_raw_pr) {

  # change column names into terms without spaces
  colnames(xf_raw_pr) <- c(
    "measurement", "tick", "well", "group",
    "time", "temp_well", "temp_env", "O2_isvalid", "O2_mmHg",
    "O2_light", "O2_dark", "O2ref_light", "O2ref_dark",
    "O2_em_corr", "pH_isvalid", "pH", "pH_light", "pH_dark",
    "pHref_light",
    "pHref_dark", "pH_em_corr", "interval"
  )

  return(xf_raw_pr)
}

```

The next preprocessing function takes the `timestamp` (column name is now `time`) from the Raw data sheet and converts the `timestamp` into minutes and seconds. This function has some more plines of code, but all it does is to add three columns to the tibble: `totalMinutes`, `minutes` and `timescale`. I used `timescale` here to make sure that I can recognize it as different from the `time` column.

```
convert_timestamp <- function(xf_raw_pr) {

  # first make sure that the data is sorted correctly
  xf_raw_pr <- dplyr::arrange(xf_raw_pr, tick, well)

  # add three columns to df (totalMinutes, minutes and time) by converting the timestamp i
  xf_raw_pr$time <- as.character(xf_raw_pr$time)
  times <- strsplit(xf_raw_pr$time, ":")
  xf_raw_pr$totalMinutes <- sapply(times, function(x) {
    x <- as.numeric(x)
    x[1] * 60 + x[2] + x[3] / 60
  })
  xf_raw_pr$minutes <- xf_raw_pr$totalMinutes - xf_raw_pr$totalMinutes[1] # first row need
  xf_raw_pr$timescale <- round(xf_raw_pr$minutes * 60)

  return(xf_raw_pr)
}
```

All other preprocessing steps and functions can be looked up in the `preprocess_xfplate.R` file on github [https://github.com/vcjdeboer/seahtrue/blob/develop-gerwin/R/preprocess\\_xfplate.R](https://github.com/vcjdeboer/seahtrue/blob/develop-gerwin/R/preprocess_xfplate.R). Combined the `preprocess_xfplate` function takes the output of the `read_xfplate` function and outputs all data in a nice data table consisting of a bunch of nested tibbles.

The `preprocess_xfplate` and `read_xfplate` functions are combined in the `run_seahtrue` function. The `seahtrue` has some extensive unit testing, user interaction, and input testing build-in using the `testthat`, `cli`, `logger` and `validate`.

The basic read and preprocess function looks like this.

```
run_seahtrue() <- function(filepath_seahorse){

  filepath %>%
    read_xfplate() %>%
    preprocess_xfplate()
}
```

In the next section we will explore the output of the `run_seahtrue` function.

### Exercise 2

The `rename_columns` function could have also been written using `clean_names` from the `janitor` package. This would have been likely faster to implement. Replace a `clean_names` code that does the same as the `rename_columns` function

```
#
```

### Solution to Exercise 2

### Exercise 3

Do the same for the `convert_timestamp` function. Use the `lubridate` package to write a simpler code

```
#
```

### Solution to Exercise 3

## 8 Seahtrue outputs

The data output format of the `run_seahtrue` function is a list of lists. List of lists is also called nesting of data. The advantage of this is that the data is properly organized, but also easily accessible. Here is an example that I took from a `tidyr` vignette <https://tidyr.tidyverse.org/articles/nest.html>.

```
library(tidyverse)

mtcars %>%
  nest(.by = cyl)
```

You can see that the the data is now nicely organized by the `cylinder` parameter. Since there are only 3 different values for the `cyl` in the `mtcars` dataset, there are now three rows and two columns, one column has the `cyl` parameter all other data is nested into a `data` column.

### **i** `.by` vs `group_by`

In one of the latest releases of the `tidyverse` the use of `.by` was introduced. Previously we used the `group_by` to tell R how to organize the data. The grouping of data remains attached to the data tibble, which sometimes could result in unintentional things to happen, when you forgot that the tibble was grouped. The `group_by` can be undone with the `ungroup` command.

With the `.by` the grouping is only apparent while using the function in which you use it as argument. `group_by` and `.by` are doing similar things so they can be used both. Let's have a look at how they work:

```

#first do a complete summarize
mtcars %>%
  summarize(meamn = mean(dis))

#second only summarize the disp for each cyl
mtcars %>%
  summarize(mean = mean(dis), .by = cyl)

#alternatively you can use
mtcars %>%
  group_by(cyl)%>%
  summarize(mean = mean(dis))

```

If you glimpse the results of the two ways of using grouping above you will see that `group_by` is doing stuff to your data, that you might not want. In this case it turns the `mtcars` dataframe into a tibble, whereas the result of the `.by` in the `summarize` function is still a dataframe. Although it might not really matter whether your data is a tibble or dataframe, it shows that `group_by` is a bit more invasive on your data.

You can use `pluck` to get to the nested `data`. Basically you just pluck a part of the data out of the full dataset.

```

mtcars %>%
  nest(.by = cyl) %>%
  pluck("data", 1)

```

Please note that we use here `"data"` instead of `data`. It can be confusing when to use the `"` or not. For example, with the `pull` function which takes one full column out of a tibble, you are not using `"`.

Also, `pluck` uses indexing for retrieving its components, it is not possible to directly get the element that belongs to `cyl == 3` for example. You would need to `filter` first on that parameter and then `pluck` the first row of data.

## 8.1 The purrr map function

The cool thing about a nested tibble is that you can quickly perform stuff on each nested tibble. A really good introduction to this is described in this blog post by Rebecca Barter



[https://www.rebeccabarter.com/blog/2019-08-19\\_purrr](https://www.rebeccabarter.com/blog/2019-08-19_purrr). You can map a function on each item from that row.

```
mtcars %>%
  nest(.by = cyl) %>%
  mutate(model =
    map(data, function(df) lm(mpg ~ wt, data = df)))
```

You see that a new column is generated named `model`, if you pluck the one of the models, you can see the typical output of the linear model (`lm`) function. For each cylinder now you creates a linear model!

```
mtcars %>%
  nest(.by = cyl) %>%
  mutate(model =
    map(data,
      function(df) lm(mpg ~ wt, data = df))) %>%
  pluck("model", 1)
```

The semantics and how to use the `map` function is nicely explained in the blog post that was referenced here above. But some more considerations here:

```
#this is the original form
mtcars %>%
  nest(.by = cyl) %>%
  mutate(model =
    map(data, function(df) lm(mpg ~ wt, data = df)))

#some of the parts are left out
mtcars %>%
  nest(.by = cyl) %>%
  mutate(model =
    map(.x = data,
      .f = function(df) lm(mpg ~ wt, data = df)))

#this makes it a bit more clear
#now .x is the data and .f is the function
```

```

#also you can change the function syntax
mtcars %>%
  nest(.by = cyl) %>%
  mutate(model = map(.x = data,
                     .f = ~ lm(mpg ~ wt, data = .x)))

#now function was replaced with ~
#and the df was replaced with .x

#this also works . instead of .x
mtcars %>%
  nest(.by = cyl) %>%
  mutate(model = map(.x = data,
                     .f = ~ lm(mpg ~ wt, data = .)))

```

Another good resource for the purrr map function is <https://dcl-prog.stanford.edu/purrr-basics.html>. map has many more forms and ways to use, which are summarized in its cheat sheet <https://github.com/rstudio/cheatsheets/blob/main/purrr.pdf>.

## 8.2 The seahtrue ouput

Now go and have a look at the `run_seahtrue` output.

```

library(tidyverse)

root_srcfile <-
  "https://raw.githubusercontent.com/vcjdeboer/"
repository_srcfile <-
  "seahtrue/develop-gerwin/data/"

download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    "seahtrue_output_donor_A.rda"),
  "seahtrue_output_donor_A.rda")

load("seahtrue_output_donor_A.rda")

```

```
seahtrue_output_donor_A %>% glimpse()
```

Also pluck some of the data

```
# get the original input filename and location
seahtrue_output_donor_A %>%
  pluck("filepath_seahorse",1)

# get the injection info
seahtrue_output_donor_A %>%
  pluck("injection_info",1)

# get the date when exp was run
seahtrue_output_donor_A %>%
  pluck("date",1)
```

Some data are simple character strings, like the `date` column, whereas others are large tables like the `raw_data` column

With this loaded data (`seahtrue_output_donor_A`) you can now do similar plotting as in the `plotting seahorse` chapter. For this we only have to `pluck` the `rate_data` out of the data set. Be carefull that we preprocessed the data and we have other column names now so first `glimpse` the data.

```
seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  glimpse()

#or get only the column names
seahtrue_output_donor_A %>%
  colnames()
```

You will see that the column names are labeled with `wave`, in this way we can distinguish for example the time column in the `raw_data` tibble from the `time_wave` column in the `rate_data` tibble. Also, please notice that we have `OCR_wave_bc` and `OCR_wave`. This distinctino is made because we can have OCR data that is background corrected or not. When clicking on the background slider in the Wave software from Agilent, the OCR data will be changed to non background corrected. If at this point the data is exported the `xlsx` input file is not background corrected. In the `seahtrue` this will show up as `OCR_wave`. Typically however the data is

background corrected, so we most of the time have `OCR_wave_bc`.

### **i** time and time again

Since rate is an aggregate of multiple O2 or pH readings, also the definition of the timing of each measurement is different between the `rate_data` and the `raw_data`. Therefore in the `seahtrue` package both times are labeled differently. For the `rate_table` we labeled it with `time_wave` and for the `raw_data` we labeled it with `timescale`. And again, we used `timescale` to distinguish it from the `time` in the original input file.

Please note if we want to plot the OCR vs time, we have to use the `OCR_wave_bc` vs `time_wave` in our ggplot aesthetics.

It is good practice to have a quick look at how the groups were named in the experiment. We can use the `pull(group)` and `unique()` commands for this:

```
#first have a look at what groups we have
seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  pull(group) %>% unique()
```

Next, take some of the groups and plot them in a ggplot:

```
seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  filter(group %in% c("200.000", "Background")) %>%
  ggplot(aes(x = time_wave, y = OCR_wave_bc,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  scale_color_brewer(palette = "Set1")+
  labs(subtitle = "200.000 cells per well vs Background",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)
```

Great, this looks exactly the same as the plot we generated using the data from the downloaded excel file in the “plotting seahorse” chapter.

## 9 Summary

### 9.0.1 Functions

- Definitions of functions
- Function argument
- How to build and execute functions
- How to use the `dplyr` `mutate` function
- Some basic understanding of the `seahtrue` `read_xfplate` functions
- Some of the `seahtrue` `preprocess_xfplate` functions
- The `run_seahtrue` function

### 9.0.2 Outputs

- What nested tibbles are, and why they are useful
- How to generate nested tibbles
- How to use the `purrr` `map` function on nested tibbles
- Get familiar with the semantics of the `map` function
- The difference between `.by` and `group_by`
- How to isolate or `pluck` data from a larger dataset
- Get familiar with the `seahtrue` output nested tibble
- The different parameters for `time` in a seahorse dataset
- How to access the data in `seahtrue` output and use it in `ggplot`

## 9.1 What we did not learn?

- How to run `seahtrue` on our own data
- How all the assertions and input checking in the `seahtrue` package are implemented
- More elaborate use cases of the `map` function from the `purrr` package, like `map2` or `map_dbl`

# **Part III**

## **Diving deeper**

In the previous chapters we learned R and got familiar with the `seahtrue` package and its output. Here we will explore what is possible with the seahtrue data in R. We will go through making some nice visualisations of the raw data and rate data. Also we will show how to combine multiple experiments into one tibble and visualize the data.

## 10 Single experiment

To dive a bit deeper into a single **seahtrue** experiment, we will first generate an overview of what the experimental set-up was.

Let's load the data first

```
library(tidyverse)

root_srcfile <-
  "https://raw.githubusercontent.com/vcjdeboer/"
repository_srcfile <-
  "seahtrue/develop-gerwin/data/"

download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    "seahtrue_output_donor_A.rda"),
  "seahtrue_output_donor_A.rda")

load("seahtrue_output_donor_A.rda")
```

Next, we make a theme that we can use for the heatmap

```
theme_hmp <- function(){
  theme_bw(base_size = 15) %+replace%
  theme(panel.grid.minor.x = element_blank(),
        panel.grid.major.x = element_blank(),
        panel.grid.minor.y = element_blank(),
        panel.grid.major.y = element_blank(),
        axis.ticks.x = element_blank(),
        axis.ticks.y = element_blank(),
        axis.title.x = element_blank(),
        axis.title.y = element_blank(),
```



```

axis.text.x = element_text(size = rel(1.3),
                           hjust = 0.5,
                           vjust = 0),
axis.text.y = element_text(size = rel(1.3),
                           hjust = 0.5,
                           vjust = 0.5),

legend.position="right",
legend.margin=margin(0,0,0,0),
legend.box.margin=margin(-10,0,-10,-10)
})

```

Then we make a nice default heatmap with the `geom_tile` function.

```

seahtrue_output_donor_A %>%
  pluck("rate_data", 1)%>%
  separate(well,
           into = c("row", "column"),
           sep = 1,
           convert = TRUE) %>%
  ggplot(aes(x = column, y = forcats::fct_rev(row))) +
  geom_tile(aes(
    fill = group),
    color = "grey50",
    show.legend = TRUE)+
  scale_x_continuous(limits= c(0.5, 12.5),
                    breaks = c(1:12),
                    position = "top",
                    expand = c(0,0))+
  scale_y_discrete(expand = c(0,0))+
  labs(fill = "group")+
  theme_hmp()

```

The default ggplot colors are quite colorfull, but might hurt your eyes... If we want colors that are different than the default ggplot colors, and we want the legend to be nicely in order we need to add some additional code.

```

#first get number of groups
number_of_groups <-
  seahtrue_output_donor_A %>%
    pluck("rate_data",1) %>%
    pull(group) %>% unique() %>%

```

```

length()

#next make a color palette that matches
#the number of groups
group_colors <-
  colorRampPalette(
    RColorBrewer::brewer.pal(8, "BrBG"))(number_of_groups)

#plot a platemap
seahtrue_output_donor_A %>%
  pluck("rate_data", 1)%>%
  filter(group != "Background") %>%
  separate(well,
            into = c("row", "column"),
            sep = 1,
            convert = TRUE) %>%
  ggplot(aes(x = column, y = forcats::fct_rev(row))) +
  geom_tile(aes(
    fill = forcats::fct_reorder(group,
                                parse_number(group))),
    color = "grey50",
    show.legend = TRUE)+
  scale_fill_manual(values= group_colors)+
  scale_x_continuous(limits= c(0.5, 12.5),
                     breaks = c(1:12),
                     position = "top",
                     expand = c(0,0))+
  scale_y_discrete(expand = c(0,0))+
  labs(fill = "group")+
  theme_htmp()

```

Another option would be to manually arrange the factors in a way that suits you best.

```

group_order <- seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  pull(group) %>% unique()

seahtrue_output_donor_A %>%
  pluck("rate_data", 1)%>%
  separate(well,
            into = c("row", "column"),

```

```

      sep = 1,
      convert = TRUE) %>%
ggplot(aes(x = column, y = forcats::fct_rev(row))) +
  geom_tile(aes(
    fill = group),
    color = "grey50",
    show.legend = TRUE)+
  scale_fill_manual(values= group_colors,
                    breaks = group_order)+ #added here
  scale_x_continuous(limits= c(0.5, 12.5),
                    breaks = c(1:12),
                    position = "top",
                    expand = c(0,0))+
  scale_y_discrete(expand = c(0,0))+
  labs(fill = "group")+
  theme_hmp()

```

## 10.1 Background

In Seahorse experiments the corners of the plate are by default assigned as **Background** wells, meaning that in these wells there is no sample but does have the same conditions and culture medium as your sample wells. Background wells need to be checked for outliers. This is not obvious from the Wave software interface, because the background is by default subtracted and users will never see the actual background data, unless they really select for it in the point-and-click software Wave. So let's make some plots of the raw background O2 data.

We will now use the `raw_data` table for plotting, and we assume you already loaded the data file above in this session.

```

seahtrue_output_donor_A %>%
  pluck("raw_data", 1) %>%
  filter(group == "Background") %>%
  ggplot(aes(x = minutes, y = O2_mmHg, color = well))+
  geom_point()+
  theme_bw(base_size = 16)+
  labs(x = "time (min)",
       y = "O2 (mmHg)")

```

This is a nice plot of the background O2 readings. It does look weird, especially because there is one well H01 which has a completely different trend than the other wells. This might be

suspected as a technical outlier. Possibly in this well there was not enough culture medium or the sensor was damaged. The lab details and observations should be aligned with the outlier calling to make sure to not erroneously flag a well as an outlier.

To make an even better visual representation of the background and to account for the different aspects how the background well data behaves we can plot only the first ticks of each measurement. We will also shift here now to the fluorescence readings of the Seahorse. Since the O2 is derived from fluorescence values in our experiments it would be good to really look at the most raw data that we get out of our experiment. The fluorescence is given as the parameter `O2_em_corr`

We have a plotting function that automates this.

```
plot_raw_BKGD <- function(total_df, var, flnme){

  theme_maxTick <- function(){
    theme_classic(base_size = 18) %+replace%
    theme(panel.grid.minor.x = element_blank(),
           panel.grid.major.x = element_blank(),
           panel.grid.minor.y = element_blank(),
           panel.grid.major.y = element_blank(),
           panel.border = element_blank(),
           axis.ticks.x = element_line(),
           axis.ticks.y = element_line(),
           axis.line.y = element_line(),
           legend.text = element_text(size = rel(0.7)),
           legend.title = element_text(size = rel(0.7)),
           axis.title.x = element_text(size = rel(0.8)),
           axis.title.y = element_text(size = rel(0.8)),
                                           angle = 90)
  )
}

custom.col <- c("#D16103", "#4E84C4", "#52854C", "#C4961A",
               "#FFDB6D", "#C4961A", "#F4EDCA",
               "#D16103", "#4E84C4", "#52854C", "#C4961A",
               "#FFDB6D", "#C4961A", "#F4EDCA" )

df<- total_df %>% filter(group == "Background") %>%
  select(measurement, well, group,
         timescale, minutes, tick, emission = all_of(var))
df <- df[!is.na(df$emission), ]
```

```

O2_targetEmission <- 12500
pH_targetEmission <- 30000
O2_target_mmHg <- 151

targetEMS <- 0

if (var == "O2_em_corr"){
  targetEMS<- O2_targetEmission
  ylabel = "emission (AU)" }
if (var == "pH_em_corr") {
  targetEMS <- pH_targetEmission
  ylabel = "emission (AU)" }
if (var == "O2_mmHg") {
  targetEMS <- O2_target_mmHg
  ylabel = "O2 (mmHg)" }

if ((targetEMS == 0)){
  targetEMS <- O2_target_mmHg
  ylabel = "O2 (mmHg)" }

ggplot(data = df)+
  geom_point(mapping = aes(x = minutes, y = emission,
                           color = well),
             alpha = 0.5, size = 3)+
  geom_hline(yintercept = targetEMS,
             linetype = "dashed", color = "#D16103")+
  theme_maxTick()+
  scale_color_manual(values = custom.col)+
  labs(title = var,
       subtitle = flnme,
       x = "time (min)",
       y = ylabel)+
  theme(plot.title = element_text(hjust = 0.5,
                                   size = 18),
        plot.subtitle = element_text(hjust = 0.5,
                                       size = 10))
}

```

We can use this function when we provide the right arguments. The argument option for the var are: O2\_em\_corr, pH\_em\_corr and O2\_mmHg.

```
seahtrue_output_donor_A %>%
  pluck("raw_data", 1) %>%
  plot_raw_BKGD(total_df = .,
                var = "O2_em_corr",
                flnme = seahtrue_output_donor_A %>%
                  pluck("plate_id", 1))
```

### Exercise 1

To calculate O2 from emission, Seahorse uses the Stern-Volmer equation. Find out (using google or chatGPT) what the stern-volmer equation is and write it in the form of a function. Use the arguments x, KSV, and F0.

You can also use the Gerencser et al. paper that describes the calculations. The method and algorithms described in this Analytical Chemistry paper from 2009 are still used today. [Gerencser et al. Anal Chem 2009](#)

```
# stern_volmer <- function(x, KSV, F0){

#}
```

### Solution to Exercise 1

```
stern_volmer <- function(x, KSV, F0){

  O2 = (1/KSV)*((F0/x)-1)
}
```

Where x is the emission (O2\_em\_corr), KSV is a constant, the stern-volmer konstant, and F0 is the emission at zero oxygen. The values of these two constants is are unique to the cartridge that you used during your experiment. Seahorse provides these numbers when updating your Wave software and matches them via a barcode read on the cartridge each run.

### 💡 Exercise 2

The KSV and F0 are provided in the assay configuration sheet of the excel output. Seahtrue puts that information in the `assay_info` table. You can access it using the `pluck` function. In this case you have to use `pluck` two times, first to get to the `assay_info` and next to the KSV or F0

```
#seahtrue_output_donor_A %>%
```

### 🔥 Solution to Exercise 2

```
KSV <- seahtrue_output_donor_A %>%  
  pluck("assay_info", 1) %>%  
  pluck("KSV", 1)  
  
F0 <- seahtrue_output_donor_A %>%  
  pluck("assay_info", 1) %>%  
  pluck("F0", 1)
```

### 💡 Exercise 3

Now use the two constants KSV and F0, and the function `stern_volmer` to calculate the O2 from `O2_em_corr`. Also use `select(well, measurement, tick, O2_mmHg, O2)` to compare the O2 with the `O2_mmHg` in the output.

```
#seahtrue_output_donor_A %>%  
#   pluck("raw_data", 1) %>%
```

### 🔥 Solution to Exercise 3

```
seahtrue_output_donor_A %>%  
  pluck("raw_data", 1) %>%  
  mutate(O2 = stern_volmer(O2_em_corr, KSV, F0)) %>%  
  #use this select to compare the output  
  select(well, measurement, tick, O2_mmHg, O2)
```

### 💡 Exercise 4

Plot the O2 background values that you just calculated using the `plot_raw_BKGD` function. Compare the plot to when plotting the `O2_mmHg` that was derived from the Seahorse output `xlsx`.

```
#seahtrue_output_donor_A %>%  
#   pluck("raw_data", 1) %>%
```

### 🔥 Solution to Exercise 4

```
seahtrue_output_donor_A %>%  
  pluck("raw_data", 1) %>%  
  mutate(O2 = stern_volmer(O2_em_corr, KSV, F0)) %>%  
  plot_raw_BKGD(total_df = .,  
                var = "O2",  
                flnme = seahtrue_output_donor_A %>%  
                  pluck("plate_id", 1))
```

### 💡 Exercise 5

Apparently the `O2_mmHg` is different from our own calculated O2 concentrations. When looking at the `O2_mmHg` background plot it looks like that these O2 values are also corrected for a background. Let's see if that is indeed the case.

Seahorse Wave subtracts the mean background from all samples. So the mean `O2_mmHg`



of the “Background group is subtracted from all samples wells (and background wells apparently). We can also do that with our `seahtrue` data. We should take care of what we need to `summarize` here, each `tick` is a unique measurement in the `raw_data`, do let’s take `tick` as the `.by` parameter

```
O2_bkgd <-
  seahtrue_output_donor_A %>%
    pluck("raw_data", 1) %>%
    filter(group == "Background") %>%
    summarize(O2_bkgd =
      mean(O2_mmHg),
      .by = tick)

#add the O2_bkgd to our seahtrue_output_donor_A
seahtrue_output_donor_A %>%
  pluck("raw_data", 1) %>%
  left_join(O2_bkgd, by = c("tick"))
```

Now we need to subtract the background O2 from all other wells and the backgrounds wells themselves.

The way this is done in the Seahorse algorithm is to take into account the ambient O2 levels. Basically what Seahorse calculates is the following:

```
correct_O2_for_background <-function(O2, O2_bkgd){
  O2_0_mmHg = 151.6900241
  O2_corrected <- O2 - O2_bkgd + O2_0_mmHg
}

seahtrue_output_donor_A %>%
  pluck("raw_data", 1) %>%
  left_join(O2_bkgd, by = c("tick")) %>%
  mutate(O2_corrected =
    correct_O2_for_background(O2_mmHg, O2_bkgd)) %>%
  select(well, tick, O2_mmHg, O2_corrected)
```

Now compare the `O2_corrected` with the original `O2_mmHg`. Do this in two ways. 1) make a `ggplot` with the `O2_corrected` on x-axis and `O2_mmHg` on the y-axis. and 2) use the `plot_raw_BKGD` function with `O2_corrected` and compare with the output from the `O2_mmHg` `plot_raw_BKGD` plot

## Solution to Exercise 5

```
#solution 1
seahtrue_output_donor_A %>%
  pluck("raw_data", 1) %>%
  left_join(O2_bkgd, by = c("tick")) %>%
  mutate(O2_corrected =
    correct_O2_for_background(O2_mmHg, O2_bkgd)) %>%
  filter(group == "Background") %>%
  ggplot(aes(x = O2_corrected, y = O2_mmHg,
    color = well))+
  geom_point()

#solution 2
seahtrue_output_donor_A %>%
  pluck("raw_data", 1) %>%
  left_join(O2_bkgd, by = c("tick")) %>%
  mutate(O2_corrected =
    correct_O2_for_background(O2_mmHg, O2_bkgd)) %>%
  plot_raw_BKGD(total_df = .,
    var = "O2_corrected",
    flnme = seahtrue_output_donor_A %>%
      pluck("plate_id", 1))
```

Although the values are not identical, they are pretty close. Indicating that the O2\_mmHg background data is likely also corrected for background in this dataset.

## 10.2 Low signals

Sometimes we don't have much sample. In most cases the sample is cells, and with low cell number the O2 consumption and extracellular acidification can be low. Seahorse defines an pretty arbitrary cut-off for basal respiration at 20 pmol/min. Below this value OCR becomes less reliable.

In the loaded experiment `seahtrue_output_donor_A`, we have a group labeled with `50.000`. In these wells we only have 50.000 cells in each well, which makes its signal difficult to detect.

```

seahtrue_output_donor_A %>%
  pluck("rate_data", 1) %>%
  filter(group %in% c("Background", "50.000", "200.000")) %>%
  ggplot(aes(x = time_wave, y = OCR_wave_bc,
             group = well,
             color = group))+
  geom_point()+
  geom_line() +
  scale_color_brewer(palette = "Set1")+
  labs(subtitle = "200.000 and 50.000 cells per well",
       x = "time (minutes)",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 16)

```

Please notice that the OCR signal for the 50.000 group is definitely below 20 pmol/min.

What if we want to investigate in more detail how our signals are for our samples with this low respiration?

We can make use of the `raw_data` again and plot the **background** O2 signal with the **sample** O2 signal in one plot. Since in the previous section we saw that O2 signals for the background wells were also corrected for background (?!), we will work with our own calculated O2 levels using the `stern_volmer` function we wrote in the previous section.

Also, we will use quite a big plotting function for this. It offers some flexibility on whether we want to plot means and/or scale the data. Also we can select specific wells and which measurements.

```

plot_raw_whichGroup_dots <-
  function(var, total_df, flnme, groupString,
           plot_the_mean, y_label, measurementString,
           wellString, targetEMS, scalingON,
           lgdWellName, ylim_lo, ylim_hi){

    # var = "O2_em_corr"
    # flnme = fileName
    # groupString = grp
    # plot_the_mean = TRUE
    # total_df = XFe96data
    # measurementString = msrs
    # wellString = wlls

```

```

# y_label = "emission"
#targetEMS = 12500
# scalingON = TRUE
# lgdWellName = "F01"

theme_maxTick <- function(){
  theme_classic(base_size = 18) %+replace%
    theme(panel.grid.minor.x = element_blank(),
          panel.grid.major.x = element_blank(),
          panel.grid.minor.y = element_blank(),
          panel.grid.major.y = element_blank(),
          panel.border = element_blank(),
          axis.ticks.x = element_line(),
          axis.ticks.y = element_line(),
          axis.line.y = element_line(),
          legend.text = element_text(size = rel(0.7)),
          legend.title = element_text(size = rel(0.7)),
          axis.title.x = element_text(size = rel(0.8)),
          axis.title.y = element_text(size = rel(0.8), angle = 90)
    )
}

custom.col <- c("#D16103", "#4E84C4", "#52854C", "#C4961A",
               "#FFDB6D", "#C4961A", "#F4EDCA",
               "#D16103", "#4E84C4", "#52854C", "#C4961A",
               "#FFDB6D", "#C4961A", "#F4EDCA" )

# first item in groupString is supposed to be Background!
# (else legend is not correct)

df<- total_df %>%
  filter(group %in% groupString) %>%
  filter(measurement %in% measurementString) %>%
  filter(well %in% wellString) %>%
  select(measurement, well, group, timescale,
         minutes, tick, param_toPlot = all_of(var))

if (scalingON == TRUE){
  firstTicks <-
    df %>%

```

```

    group_by(well, measurement) %>%
    slice(1) %>%
    mutate(scaleFactor = targetEMS/param_toPlot) %>%
    select(well, measurement, scaleFactor)

df <- left_join(df, firstTicks,
                by = c("measurement", "well"))
df <- df %>%
  mutate(newParam = param_toPlot*scaleFactor) %>%
  select(!param_toPlot)
names(df)[names(df) == 'newParam'] <- 'param_toPlot'
}

#levels(as.factor(df$group))
#df$group <- factor(df$group, levels = grp)

df_mean <- df %>%
  group_by(group, minutes) %>%
  summarize(mn= mean(param_toPlot),
            sd = sd(param_toPlot))

df_mean <- df_mean %>%
  arrange(desc(group))
df <- df %>%
  arrange(desc(group))

if (plot_the_mean == TRUE){
  ggplot(data = df_mean)+
    geom_errorbar(mapping = aes(x = minutes,
                               y = mn,
                               ymin = mn - sd,
                               ymax = mn + sd),
                 width = 0,
                 color = "#293352",
                 alpha = 0.6,
                 size = 0.3)+
    geom_point(mapping = aes(x = minutes, y = mn,
                             color = group),
              alpha = 0.8, size = 3)+

```

```

geom_hline(yintercept = targetEMS,
           linetype = "dashed",
           color = "#D16103")+
scale_color_manual(name = "well",
                   breaks = groupString,
                   values = custom.col,
                   labels = c("Background",
                              lgdWellName))+

theme_maxTick()+
labs(title = var,
      subtitle = flnme,
      x = "time (min)",
      y = y_label)+
theme(plot.title = element_text(hjust = 0.5,
                                size = 18),
      plot.subtitle = element_text(hjust = 0.5,
                                   size = 10))+
ylim(ylim_lo, ylim_hi)

} else{
ggplot(data = df)+
  geom_point(mapping = aes(x = minutes,
                          y = param_toPlot,
                          color = group),
            alpha = 0.8, size = 3)+
  #geom_line(aes(x = minutes, y = param_toPlot,
  # color = group, group = well),
  # alpha = 0.6, size = 1)+
  geom_hline(yintercept = targetEMS,
            linetype = "dashed",
            color = "#D16103")+
  scale_color_manual(name = "well",
                    breaks = groupString,
                    values = custom.col,
                    labels = c("Background",
                               lgdWellName))+

  theme_maxTick()+
  labs(title = var,
        subtitle = flnme,
        x = "time (min)",
        y = y_label)+

```

```

    theme(plot.title = element_text(hjust = 0.5,
                                     size = 18),
           plot.subtitle = element_text(hjust = 0.5,
                                         size = 10))+
    ylim(ylim_lo, ylim_hi)

  }

}

```

Let's explore this huge function (with not so tidy coding in it...), by using it:

```

#define the df to plot
#don't forget to have the stern_volmer function
#and the KSV and F0 loaded for this
XFe96data <- seahtrue_output_donor_A %>%
  pluck("raw_data", 1) %>%
  mutate(O2 = stern_volmer(O2_em_corr, KSV, F0))

#set input parameters for function
raw_em_corr <- c("O2")
emission_target = 151.67
label <- c("O2 (mmHg)")

ylim_lo <- 150
ylim_hi <- 156

fileName <- "well D02"

grp <- c("Background", "50.000")

background_wells <- c("A01", "A12", "H01", "H12")
sample_wells <- c("D02")
wlls <- c(sample_wells, background_wells)
msrs <- c("1", "2", "3")

legendWellName <- "D02"

#call function

```

```

plot_raw_whichGroup_dots(raw_em_corr,
                        XFe96data,
                        fileName,
                        grp,
                        plot_the_mean = TRUE,
                        label,
                        msrs,
                        wlls,
                        emission_target,
                        scalingON = TRUE,
                        legendWellName,
                        ylim_lo,
                        ylim_hi)

```

You can see in this plot that background O2 levels rise in each measurement. This drift is consistently seen in all instruments and experimental conditions. The upward drift in O2, is also why OCRs for background wells are often negative in your Seahorse software Wave graphs (when you point-and-click to have the background not subtracted). The explanation that Gerenscer et al. gave for the drift was that either 1) temperature is not stable during a measurement and the fluorescent sensors are temperature sensitive or 2) that O2 levels in the microchamber that is formed when probe is at its measuring position is entering from the plastic or culture medium above. Both reasons are debatable I think.

Although the O2 levels of backgrounds increase, it can be seen that the O2 levels of well D02 increase less. Meaning that there oxygen consumption is higher than the background.

#### Exercise 6

Change the inputs for the `plot_raw_whichGroup_dots` (in a meaningful way), to plot the 1) `O2_em_corr`, 2) plot another well D04 (please note that you also have change the group name because it is from the 100.000 group)

```
#set the inputs for the plot_raw_whichGroup_dots function
```



## Solution to Exercise 6

```
#changing to O2 emission
raw_em_corr <- c("O2_em_corr")
emission_target = 12500
label <- c("emission (AU)")

ylim_lo <- 12000
ylim_hi <- 12800

# changing well
sample_wells <- c("D04")
wlls <- c(sample_wells, background_wells)
grp <- c("Background", "100.000")
legendWellNName <- "D04"
fileName <- "well D04"
```

## 10.3 Plotting basal and maximal respiration

Pluck the `injection_info` table from the `seahtrue_output_donor_A` dataset to see what how the injections were defined in the experimental set-up before running the seahorse.

```
seahtrue_output_donor_A %>%
  pluck("injection_info",1)
```

This is not a typical mito-stress test experiment, where we inject oligomycin, FCCP and antimycinA/rotenone sequentially. Instead we inject only FCCP and antimycinA/rotenone.

To get the maximal and basal respiration out of the ocr rate table, we need to do some calculations. We first make some assumptions and definitions:

We call each interval between two injections or between start and an injection or between an injection and the end a **phase**

Each phase has a unique name that is named after the injection that was last. The first phase after the start is called `init_ocr` and we also typically have the phases

om\_ocr, fccp\_ocr and amrot\_ocr. Phases are marked with either \_ocr or \_ecar, because these are distinct parameters.

To calculate respiration parameters, like basal respiration (= basal\_ocr), we define the following:

- $\text{basal\_ocr} = \text{init\_ocr} - \text{amrot\_ocr}$ .
- $\text{max\_ocr} = \text{fccp\_ocr} - \text{amrot\_ocr}$
- $\text{spare\_ocr} = \text{fccp\_ocr} - \text{init\_ocr}$
- $\text{proton\_leak} = \text{om\_ocr} - \text{amrot\_ocr}$
- $\text{atp\_linked} = \text{init\_ocr} - \text{om\_ocr}$

We also use indices to have relative parameters:

- $\text{spare\_ocr\_index} = (\text{spare\_ocr} / \text{basal\_ocr}) * 100$
- $\text{basal\_ocr\_index} = (\text{basal\_ocr} / \text{max\_ocr}) * 100$
- $\text{leak\_index} = (\text{proton\_leak} / \text{basal\_ocr}) * 100$
- $\text{coupling\_index} = (\text{atp\_linked} / \text{basal\_ocr}) * 100$  %>%

Another important assumption is that we are not using average values to represent each phase, but instead we use a specific measurement. The reason for this is that we assume that for all phases, except FCCP, three measurements are needed in time to get to steady-state. For FCCP injection, we assume that it reaches steady-state fast, or at least its maximal ocr, so we take the first measurement after injection as the measurement representing the FCCP phase.

Let's now put that into R code. We call the type of experiment we did in this dataset a `maximal_capacity` (`maxcap`) test.

We also injected monensin, which can maximize ECAR, but we don't need it for OCR calculations.

```
# first define which timepoints are what
param_set_maxcap_ocr <- c(init_ocr = "m3",
                          fccp_ocr = "m4",
                          amrot_ocr = "m9",
                          mon_ocr = "m12"
                          )

#next do some pivoting, renaming and selecting
seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  select(well, measurement, group, ocr = OCR_wave_bc) %>%
  pivot_wider(names_from = measurement, names_prefix = "m", values_from = ocr) %>%
```

```

rename(all_of(param_set_maxcap_ocr)) %>%
select(contains(c("wel"," group", "ocr")))

```

Now for each well we have the parameters related to the phases that we defined in the parameter set.

Next we want to calculate the respiration parameters and indices.

```

seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  select(well, measurement, group, ocr = OCR_wave_bc) %>%
  pivot_wider(names_from = measurement,
              names_prefix = "m",
              values_from = ocr) %>%
  rename(all_of(param_set_maxcap_ocr)) %>%
  select(contains(c("well","group", "ocr"))) %>%
  #piped here to the mutate
  mutate(non_mito_ocr = amrot_ocr,
         basal_ocr = init_ocr - non_mito_ocr,
         max_ocr = fccp_ocr - amrot_ocr,
         spare_ocr = max_ocr - basal_ocr,
         spare_ocr_index = (spare_ocr / max_ocr)*100,
         basal_ocr_index = (basal_ocr/max_ocr)*100)

```

With this data we can plot our typical basal and maximal bar/scatter plots that we see in our lovely papers, presentations and theses.

```

webr::install("ggdist")

param_set_maxcap_ocr <- c(init_ocr = "m3",
                          fccp_ocr = "m4",
                          amrot_ocr = "m9",
                          mon_ocr = "m12"
                          )

seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  select(well, measurement, group, ocr = OCR_wave_bc) %>%
  pivot_wider(names_from = measurement,
              names_prefix = "m",
              values_from = ocr) %>%

```

```

rename(all_of(param_set_maxcap_ocr)) %>%
select(contains(c("well", "group", "ocr")) %>%
#piped here to the mutate
mutate(non_mito_ocr = amrot_ocr,
       basal_ocr = init_ocr - non_mito_ocr,
       max_ocr = fccp_ocr - amrot_ocr,
       spare_ocr = max_ocr - basal_ocr,
       spare_ocr_index = (spare_ocr / max_ocr)*100,
       basal_ocr_index = (basal_ocr/max_ocr)*100) %>%
filter(group %in% c("150.000", "250.000")) %>%
ggplot(aes(x = group, y = max_ocr, color = group))+
  geom_bar(data = . %>%
           summarize(median_max_ocr = median(max_ocr),
                     .by = group),
           mapping = aes(
             x = forcats::fct_reorder(
               group,
               parse_number(group)),
             y = median_max_ocr,
             fill = group),
           stat="identity",
           alpha=0.4,
           width=0.2))+
  ggdist::geom_weave() +
  ggdist::stat_pointinterval()+
  colorspace::scale_colour_discrete_divergingx(
    palette = "Geyser",
    rev = FALSE)+
  colorspace::scale_fill_discrete_divergingx(
    palette = "Geyser",
    rev = FALSE)+
  labs(subtitle = "Maximal OCR",
       x = "",
       y = "OCR (pmol/min)")+
  theme_bw(base_size = 18)

```

### 💡 Exercise 7

The plot above shows `maximal ocr`. Now make your own plot with 1) `basal_ocr` and 2) all groups except background. Make sure to order the group legend tidily and have readable x-axis labels.

```
#use the code block above and change that code
```

## Solution to Exercise 7

```

#solution 1 and 2 together
#lines with changes marked with #VB

webr::install("ggdist")

param_set_maxcap_ocr <- c(init_ocr = "m3",
                          fccp_ocr = "m4",
                          amrot_ocr = "m9",
                          mon_ocr = "m12"
                          )

group_order <- seahtrue_output_donor_A %>% #VB
  pluck("rate_data",1) %>%
  pull(group) %>% unique()

seahtrue_output_donor_A %>%
  pluck("rate_data",1) %>%
  select(well, measurement, group, ocr = OCR_wave_bc) %>%
  pivot_wider(names_from = measurement,
              names_prefix = "m",
              values_from = ocr) %>%
  rename(all_of(param_set_maxcap_ocr)) %>%
  select(contains(c("well", "group", "ocr"))) %>%
  #piped here to the mutate
  mutate(non_mito_ocr = amrot_ocr,
         basal_ocr = init_ocr - non_mito_ocr,
         max_ocr = fccp_ocr - amrot_ocr,
         spare_ocr = max_ocr - basal_ocr,
         spare_ocr_index = (spare_ocr / max_ocr)*100,
         basal_ocr_index = (basal_ocr/max_ocr)*100) %>%
  filter(group!= c("Background")) %>% #VB
  ggplot(aes(x = group, y = basal_ocr, color = group))+ #VB
  geom_bar(data = . %>%
           summarize(
             median_basal_ocr = median(basal_ocr), #VB
             .by = group),
           mapping = aes(
             x = forcats::fct_reorder(
               group,
               parse_number(group)),
             y = median_basal_ocr, #VB
             fill = group),
           stat="identity",
           alpha=0.4,
           width=0.4)+ #VB 119

ggdist::geom_weave() +
ggdist::stat_pointinterval()+
colorspace::scale_colour_discrete_divergingx(
  palette = "Geyser",
  rev = FALSE,
  breaks =
    group_order

```

## 11 Multiple experiments

The reading and processing that **seahtrue** uses, together with the nested tibble output, has another benefit. It allows for collecting and organizing multiple experiments into one nested tibble. We can do that by loading multiple **xlsx** files into our working directory, making a file list of those excel file names and mapping the **run\_seahtrue** function against the file list.

For now we will first have a look at how the output looks like. We load the three experiment output from github.

```
library(tidyverse)

root_srcfile <-
  "https://raw.githubusercontent.com/vcjdeboer/"
repository_srcfile <-
  "seahtrue/renv-sync_issue_2/data/"

download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    "xf_3.rda"),
  "xf_3.rda")

load("xf_3.rda")

xf_3
```

You can see we have one experiment per row of the tibble. They are labeled with the **plate\_id**, **file\_path**, and **date** in the first three columns. Since it is in one tibble, we can now pluck for example all **raw\_data** from the complete tibble:

```
xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data))
```



You see we now have a tibble of almost 2 million rows and 22 columns, nicely and tidily loaded in our webR/wasm R environment!! How cool is that! Our favorite point-and-click software Excel wouldn't even be able to handle this amount of rows.

Now we can plot for example the distribution of emission values for each experiment. We use the `ggridges` package for this.

```
webr::install("ggridges")

xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data)) %>%
  filter(group != "Background") %>%
  ggplot(aes(x = O2_mmHg, y = plate_id))+
    ggridges::geom_density_ridges()+
    facet_wrap(~forcats::fct_reorder(group,
                                     parse_number(group)))
```

With only a couple of lines of code we go from `raw_data` to real insights in your data. For example,

- The three experiments seem to have similar distributions between groups, but the third experiment V01744... looks a bit different than the first two
- From the plot, it can be seen that in the group with highest cell density 300.000 the O2 levels are still in a range that the wells do not become hypoxic. The lowest O2 levels are at around 100 mmHg.
- For the 50.000 group in the first two experiments there seems to be some signal in the right tail of the distribution which is not obviously in the third experiment 50.000 group or in the other groups.

Let's explore this right tail of the 50.000 group in more detail.

```
library(ggridges)

xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data)) %>%
  filter(group == "50.000") %>%
  ggplot(aes(x = O2_mmHg, y = plate_id,
            fill = ifelse(after_stat( x > 154),
                          "above 154", "below 154")))+
  stat_density_ridges(geom = "density_ridges_gradient",
```

```

    quantile_lines = TRUE,
    quantiles = 2) +
  theme_ridges() +
  scale_fill_manual(values = c("red", "gray70"),
                    name = NULL)

```

So what are these high O2\_mmHg values. We can find out by looking at the data for the individuals wells. By filtering for the first plate we can see the O2\_mmHg in more detail. By using a short unique string as input for the `str_detect` we only have to type a couple of characters, which saves us some time. Now we make a simple scatter plot.

```

library(RColorBrewer)
xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data)) %>%
  filter(group == "50.000") %>%
  filter(str_detect(plate_id, "V01947")) %>%
  ggplot(aes(x = minutes, y = O2_mmHg,
             color = well)) +
  geom_point() +
  scale_color_manual(values =
    colorRampPalette(
      brewer.pal(4, "PuOr"))(14)) +
  theme_bw(base_size = 18) +
  labs(y = "O2 (mmHg)",
       x = "time (minutes)")

```

Indeed, there are two wells that have high O2\_mmHg. Because of the many colors it is difficult to see which wells they are. Now do some ggplot trickery to label the two wells with higher O2.

```

xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data)) %>%
  filter(group == "50.000") %>%
  filter(str_detect(plate_id, "V01947")) %>%
  ggplot(aes(x = minutes, y = O2_mmHg,
             color = well)) +
  geom_point() +
  geom_text(data = . %>%

```

```

        filter(minutes == max(minutes)) %>%
        filter(O2_mmHg >153),
        aes(label = well),
        vjust = 2.4,
        show.legend = FALSE)+
scale_color_manual(values =
  colorRampPalette(
    brewer.pal(4, "PuOr"))(14))+
theme_bw(base_size= 18)+
labs(y = "O2 (mmHg)",
      x = "time (minutes)")

```

### 💡 Exercise 1

Make the same scatter plot for the other two plates for the 50.000 group, and observe if there are wells with abnormally high O2.

```

xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data)) %>%
  filter(group == "50.000") #>%

```

## Solution to Exercise 1

```
#solution for experiment 2
xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data)) %>%
  filter(group == "50.000") %>%
  filter(str_detect(plate_id,"V01941")) %>% #change 7 to 1
  ggplot(aes(x = minutes, y = O2_mmHg,
    color = well))+
    geom_point()+
    geom_text(data = . %>%
      filter(minutes == max(minutes)) %>%
      filter(O2_mmHg >153),
      aes(label = well),
      vjust = 2.4)+
  scale_color_manual(values =
    colorRampPalette(
      brewer.pal(4, "PuOr"))(14))+
  theme_bw(base_size= 18)+
  labs(y = "O2 (mmHg)",
    x = "time (minutes)")

# wells C02 and F01 are high
```

```

#solution for experiment 3
xf_3 %>%
  select(plate_id, raw_data) %>%
  unnest(c(raw_data)) %>%
  filter(group == "50.000") %>%
  filter(str_detect(plate_id,"V01744")) %>% #change to 1744
  ggplot(aes(x = minutes, y = O2_mmHg,
    color = well))+
    geom_point()+
    scale_color_manual(values =
      colorRampPalette(
        brewer.pal(4, "PuOr"))(14))+
    theme_bw(base_size= 18)+
    labs(y = "O2 (mmHg)",
      x = "time (minutes)")

#O2 mmHg is similar in all wells (no high O2 outlier)

```

## **Part IV**

# **Surfing the waves**

## 12 Running seahtrue functions

In this section we will work directly with the functions from the **seahtrue** package, since we didn't make the package available via webr/wasm, we cannot do a library call to the **seahtrue** package. Instead we just load all three functions manually here in our session. The loading and running should go pretty fast!!

You can click on the **Run code** to get all functions into the memory of your session. Since R uses **lazy evaluation** what happens is that just a bunch a text is assigned to the function name and there is no evaluation of the code. That also means that it is quickly run. I have a small text output to see that indeed the code has been run.

### 12.0.1 read\_xf\_plate()

```
read_xfplate <- function(filepath_seahorse) {  
  
  # read data  
  xf_raw <- get_xf_raw(filepath_seahorse)  
  xf_rate <- get_xf_rate(filepath_seahorse) #outputs list of 2  
  xf_norm <- get_xf_norm(filepath_seahorse) #outputs list of 2  
  xf_buffer <- get_xf_buffer(filepath_seahorse)  
  xf_inj <- get_xf_inj(filepath_seahorse)  
  xf_pHcal <- get_xf_pHcal(filepath_seahorse)  
  xf_O2cal <- get_xf_O2cal(filepath_seahorse)  
  #xf_flagged <- get_xf_flagged(filepath_seahorse)  
  xf_assayinfo <- get_xf_assayinfo(filepath_seahorse,  
                                   norm_available = xf_norm[[2]],  
                                   xls_ocr_backgroundcorrected = xf_rate[[2]])  
  
  xf_norm <- xf_norm[[1]]  
  xf_rate <- xf_rate[[1]]  
  
  # make the output list  
  xf <- list(  
    raw = xf_raw,
```

```

        rate = xf_rate,
        assayinfo = xf_assayinfo,
        inj = xf_inj,
        pHcal = xf_pHcal,
        O2cal = xf_O2cal,
        norm = xf_norm,
        flagged = "empty",
        buffer = xf_buffer,
        filepath_seahorse = filepath_seahorse
    )

    return(xf)
}

get_xf_raw <- function(filepath_seahorse){

    xf_raw <- readxl::read_excel(filepath_seahorse,
                                sheet = "Raw")
}

get_xf_norm <- function(filepath_seahorse){

    norm_info <- get_platelayou_data(filepath_seahorse,
                                    my_sheet = "Assay Configuration",
                                    my_range = "B84:N92",
                                    my_param = "cell_n")

    if (sum(is.na(norm_info$cell_n)) > 90){
        norm_available <- FALSE
    } else {
        norm_available <- TRUE}

    xf_norm <- list(norm_info, norm_available)

    return(xf_norm)
}

get_originalRateTable <- function(filepath_seahorse){

```



```

original_rate_df <- readxl::read_excel(filepath_seahorse, sheet = "Rate")

# because rate data can be either background corrected or not this should be checked first
# first verify whether a "Background" group exists in the original_rate_df

if ("Background" %in% {original_rate_df$Group %>% unique()}) {

  check_background <- original_rate_df %>%
    dplyr::filter(Group == "Background") %>%
    dplyr::select(OCR) %>%
    dplyr::reframe(mean = mean(OCR)) %>%
    dplyr::pull(mean)

  if (check_background == 0) {
    corrected_allready <- TRUE
  } else {
    corrected_allready <- FALSE
  }

} else {

  #in case when there is no Background group we work with the original data
  # that is in the input file "Rate" sheet
  # please note that there will be warning logged, but the columns will be
  # labeled incorrectly as if the data is background corrected

  corrected_allready <- TRUE

}

if (corrected_allready == TRUE){
  colnames(original_rate_df) <-
    c("measurement", "well", "group",
      "time_wave", "OCR_wave_bc",
      "ECAR_wave_bc", "PER_wave_bc")
  original_rate_df <- original_rate_df %>%
    dplyr::mutate(OCR_wave = 0, ECAR_wave = 0)

  original_rate_df <- original_rate_df %>%
    dplyr::select(measurement, well, group,
                  time_wave, OCR_wave, OCR_wave_bc,

```

```

        ECAR_wave, ECAR_wave_bc)

} else{
  colnames(original_rate_df) <-
    c("measurement", "well", "group",
      "time_wave", "OCR_wave",
      "ECAR_wave", "PER_wave")

  #do background subtraction forr wave table
  background <- original_rate_df %>%
    dplyr::filter(group=="Background") %>%
    dplyr::group_by(measurement) %>%
    dplyr::reframe(bkg_OCR_wave = mean(OCR_wave),
                  bkg_ECAR_wave = mean(ECAR_wave)
    )
  original_rate_df <- dplyr::left_join(original_rate_df,
                                     background,
                                     by = c("measurement"), copy = TRUE)

  original_rate_df$OCR_wave_bc <- original_rate_df$OCR_wave - original_rate_df$bkg_OCR_w
  original_rate_df$ECAR_wave_bc <- original_rate_df$ECAR_wave - original_rate_df$bkg_ECA

  original_rate_df <- original_rate_df %>%
    dplyr::select(measurement, well, group,
                  time_wave, OCR_wave, OCR_wave_bc,
                  ECAR_wave, ECAR_wave_bc)
}

original_rate_df_list <- list(original_rate_df, corrected_allready)

return(original_rate_df_list)

}

get_xf_rate <- function(filepath_seahorse){

  #first item is table, second item is background_corrected logical
  xf_rate_list <- get_originalRateTable(filepath_seahorse)

  return(xf_rate_list)
}

```

```

get_xf_buffer <- function(filepath_seahorse){

  bufferfactor_info <- get_platelayout_data(filepath_seahorse,
                                             my_sheet = "Assay Configuration",
                                             my_range = "B96:N104",
                                             my_param = "bufferfactor")

  return(bufferfactor_info)
}

get_xf_pHcal <- function(filepath_seahorse){

  pH_calibration <- get_platelayout_data(filepath_seahorse,
                                           my_sheet = "Calibration",
                                           my_range = "P16:AB24",
                                           my_param = "pH_cal_em")

  return(pH_calibration)
}

get_xf_O2cal <- function(filepath_seahorse){

  O2_calibration <- get_platelayout_data(filepath_seahorse,
                                           my_sheet = "Calibration",
                                           my_range = "B7:N15",
                                           my_param = "O2_cal_em")

  return(O2_calibration)
}

get_xf_inj <- function(filepath_seahorse, injscheme = "HAP"){

  #command_index in "Operation Log" sheet give numbers to the phases in a seahorse exp
  # each command (eg. "mix", "measure") gets the command_index for that phase
  # 0 = moving operation
  # 1 = calibration
  # 2 = baseline

```

```

# 3 = injection 1
# 4 = injection 2
# 5 = injection 3
# 6 = injection 4

#read injection strategy and measurements from "Operation Log" sheet
info_sh<-readxl::read_excel(filepath_seahorse, sheet = "Operation Log")
colnames(info_sh) <- c("instruction_name","command_name",
                      "command_index","start_time","end_time",
                      "completion_status")

if (injscheme == "HAP"){
  #assumes injection names are available in operation log file (this is the case for most)
  measurement_info <- dplyr::filter(info_sh, command_name == "Measure")
  measurement_info$interval <- measurement_info$command_index -1
  measurement_info$measurement <- 1:nrow(measurement_info)
  measurement_info <- measurement_info %>% dplyr::select(measurement, interval, injection_name)
}

if (injscheme == "manual"){

  #in case there is no command index in "operation log"
  command_names <- c("XF - PC_Measure", "XF - PC_Inject")
  measurement_info <- dplyr::filter(info_sh, command_name %in% command_names)

  # "PC - inject" has a number as command_index
  # "PC - measure" command_index == 0
  # I use that to set the command_index
  interval = 1
  for (i in 1:nrow(measurement_info)){
    if(measurement_info$command_index[i] == 0){
      measurement_info$command_index[i] <- interval } else {
        interval <- interval +1
        measurement_info$command_index[i] <- interval}
  }
  colnames(measurement_info)[3] <- "interval"
  measurement_info <- dplyr::filter(measurement_info, command_name == "XF - PC_Measure")
  measurement_info$measurement <- 1:nrow(measurement_info)
  measurement_info <- measurement_info %>% dplyr::select(measurement, interval)

  #gives name of the injection manually
  # case mitostress

```

```

injections <- c("basal", "OM", "FCCP", "AM/rot")
injections_mitostress <- tibble::tibble(interval = 1:4, injection=c("basal", "OM", "FCCP", "AM/rot"))
measurement_info <- dplyr::left_join(measurement_info, injections_mitostress, by = c("interval" = "interval"))

## case glycostress
#injections <- c("basal", "glucose", "OM", "2DG")
#injections_glycostress <- tibble(interval = 1:4, injection=injections)
#measurement_info <- left_join(measurement_info, injections_glycostress, by = c("interval" = "interval"))
}

return(measurement_info)
}

get_xf_assayinfo <- function(filepath_seahorse,
                             date_style = "empty",
                             instrument = "XFe96",
                             norm_available,
                             xls_ocr_backgroundcorrected) {

  if (instrument == "XFHSmini"){
    gain1_cell <- "D68"
    gain2_cell <- "E68"
  }

  if (instrument == "XFe96"){
    gain1_cell <- "D70"
    gain2_cell <- "E70"
  }

  # read Assay Configuration sheet
  meta_df <- readxl::read_excel(filepath_seahorse,
                                sheet = "Assay Configuration",
                                col_names = c("parameter", "value"),
                                range = "A1:B83"
  )

  meta_df <- meta_df[!is.na(meta_df$parameter), ]

```

```

# read Assay Configuration sheet gain1

gain1 <- readxl::read_excel(filepath_seahorse,
                             sheet = "Assay Configuration",
                             col_names = c("value"),
                             range = gain1_cell
)

# read Assay Configuration sheet gain2

gain2 <- readxl::read_excel(filepath_seahorse,
                             sheet = "Assay Configuration",
                             col_names = c("value"),
                             range = gain2_cell
)

# read target emission cells
O2_target_emission <- readxl::read_excel(filepath_seahorse,
                                           sheet = "Calibration",
                                           col_names = FALSE,
                                           range = "B4"
)

# read pH target emission cells
pH_target_emission <- readxl::read_excel(filepath_seahorse,
                                           sheet = "Calibration",
                                           col_names = FALSE,
                                           range = "P4"
)

FO <- as.double(meta_df$value[meta_df$parameter == "Calculated FO"])
V_C <- as.double(meta_df$value[meta_df$parameter == "Pseudo Volume"])
Tau_AC <- as.double(meta_df$value[meta_df$parameter == "TAC"])
Tau_W <- as.double(meta_df$value[meta_df$parameter == "TW"])
Tau_C <- as.double(meta_df$value[meta_df$parameter == "TC"])

```

```

Tau_P <- as.double(meta_df$value[meta_df$parameter == "TP"])
KSV_original <- as.double(meta_df$value[meta_df$parameter == "ksv"])
KSV_corrected <- as.double(meta_df$value[meta_df$parameter == "Corrected Ksv"])
KSV_tempCorrection <- as.logical(meta_df$value[meta_df$parameter == "Ksv Temp Correction"])
KSV <- KSV_corrected

pH_0 <- as.double(meta_df$value[meta_df$parameter == "Calibration pH"])
pH_plateVolume <- as.double(meta_df$value[meta_df$parameter == "Plate Volume"])
pH_kVol <- as.double(meta_df$value[meta_df$parameter == "kVol"])

plate_id <- meta_df$value[meta_df$parameter == "Plate Barcode"]
cartridge_barcode <- meta_df$value[meta_df$parameter == "Cartridge Barcode"]
assay_name <- meta_df$value[meta_df$parameter == "Assay Name"]
instrument_serial <- meta_df$value[meta_df$parameter == "Instrument Serial"]

pH_targetEmission <- as.double(pH_target_emission[[1]])
O2_targetEmission <- as.double(O2_target_emission[[1]])
gain1 <- as.double(gain1[[1]])
gain2 <- as.double(gain2[[1]])

# other constants
O2_0_mmHg <- 151.6900241
O2_0_mM <- 0.214

if (date_style == "US"){
  date_run <- lubridate::mdy_hm(meta_df$value[meta_df$parameter == "Last Run"])
  logger::log_info("Converted date to US format (US = mdy_hm, NL = dmy_hm).") # (Date-time)
  #be carefull with the data format in excel! either mdy or dmy
}

if (date_style == "NL"){
  date_run <- lubridate::dmy_hm(meta_df$value[meta_df$parameter == "Last Run"])
  logger::log_info("Converted date to NL format (US = mdy_hm, NL = dmy_hm).") # (Date-time)
  #be carefull with the data format in excel! either mdy or dmy
}

if (date_style == "empty"){
  date_run <- meta_df$value[meta_df$parameter == "Last Run"] # (Character instead of date)
  logger::log_info("Date-style is empty, no date conversion was performed. Format is 'ch'")
  #be carefull with the data format in excel! either mdy or dmy
}

```

```

}

if(instrument == "XFHSmini"){
  tibbler <- tibble::tibble(
    F0 = 4.63e04,
    V_C = 9.15,
    Tau_AC = 746,
    Tau_W = 296,
    Tau_C = 246,
    Tau_P = 60.9,
    KSV = 2.06e-02,
    KSV_corrected = 2.06e-02,
    KSV_original = 2.06e-02,
    KSV_tempCorrection = FALSE,
    gain1,
    gain2,
    pH_0,
    pH_plateVolume,
    pH_kVol,
    pH_targetEmission,
    O2_targetEmission,
    plate_id,
    cartridge_barcode,
    date_run,
    assay_name,
    instrument_serial,
    O2_0_mmHg,
    O2_0_mM
  )
}

if(instrument == "XFe96"){
  tibbler <- tibble::tibble(
    F0,
    V_C,
    Tau_AC, Tau_W,
    Tau_C, Tau_P,
    KSV,
    KSV_tempCorrection,
    KSV_original,
    gain1,

```



```

    gain2,
    pH_0,
    pH_plateVolume,
    pH_kVol,
    pH_targetEmission,
    O2_targetEmission,
    plate_id,
    cartridge_barcode,
    date_run,
    assay_name,
    instrument_serial,
    O2_0_mmHg,
    O2_0_mM
  )
}

tibbler$norm_available <- norm_available
tibbler$xls_ocr_backgroundcorrected <- xls_ocr_backgroundcorrected

return(tibbler)
}

get_platelayou_data <- function(filepath_seahorse, my_sheet, my_range, my_param ){

  df <- readxl::read_excel(filepath_seahorse, sheet = my_sheet, range = my_range)

  colnames(df)[1] <- "firstCol"

  df <- tidyr::gather(df, key = "key", value = "my_value", -firstCol) %>%
    dplyr::mutate(firstCol = paste0(firstCol, key) ) %>%
    dplyr::select(well = firstCol, my_value) %>%
    dplyr::arrange(gsub("\\d", "", well, as.numeric(gsub("\\D", "", well))))

  colnames(df)[2] <- my_param

  # add a zero between letter and number if wellname has 2 characters for normalization
  for (i in 1:nrow(df)){
    if (nchar(df$well[i]) == 2) {
      wellName <- sub("(.{1})(.*)", "\\10\\2", df$well[i])
    }
  }
}

```

```

    } else {
      wellName <- df$well[i]
    }
    df$well[i] <- wellName
  }

  return(df)
}

print("read_xfplate is ready")

```

## 12.0.2 preprocess\_xfplate()

```

preprocess_xfplate <- function(xf){

  # Use our xf list from read_xfplate() with all the necessary Seahorse data to fill this

  xf_raw_pr <- preprocess_xf_raw(xf$raw,
                                xf$pHcal,
                                xf$inj,
                                xf$assayinfo,
                                xf$buffer,
                                xf$norm,
                                xf$flagged)

  xf_rate_pr <- preprocess_xf_rate(xf$rate,
                                   xf$norm,
                                   xf$flagged)

  xf_plate_pr <- xf_raw_pr %>%
    dplyr::group_by(plate_id) %>%
    tidyr::nest() %>%
    dplyr::mutate(filepath_seahorse = list(tibble::tibble(
      directory_path = dirname(as.character(xf$filepath_seahorse)),
      base_name = basename(as.character(xf$filepath_seahorse)),
      full_path = xf$filepath_seahorse
    )),

```

```

        date = xf$assayinfo$date_run,
        assay_info = list(tibble::tibble(xf$assayinfo)),
        rate_data = list(tibble::tibble(xf_rate_pr)),
        injection_info = list(tibble::tibble(xf$inj))) %>%
dplyr::select(plate_id, filepath_seahorse, date, assay_info, injection_info,
              raw_data = data, rate_data)

return(xf_plate_pr)
}

preprocess_xf_raw <- function(xf_raw,
                              xf_pHcal,
                              xf_inj,
                              xf_assayinfo,
                              xf_buffer,
                              xf_norm,
                              xf_flagged) {

# convert the original integer column to integers again, instead of double
xf_raw_pr <- xf_raw #VB did original did not work in webr ?!

# rename columns
xf_raw_pr <- rename_columns(xf_raw_pr)

# convert time column
xf_raw_pr <- convert_timestamp(xf_raw_pr)

# correct pH_em_corr
xf_raw_pr$pH_em_corr_corr <- correct_pH_em_corr(xf_raw_pr$pH_em_corr,
                                                xf_pHcal$pH_cal_em,
                                                xf_assayinfo$pH_targetEmission[1])

# calculate backgrounds and join
background <- calc_background(xf_raw_pr)

xf_raw_pr <- xf_raw_pr %>%
  dplyr::left_join(background, by = c("measurement"), relationship = "many-to-many")

```

```

# add injection info
xf_raw_pr <- dplyr::left_join(xf_raw_pr, xf_inj, by = "measurement")

#add plate_id to df
xf_raw_pr$plate_id <- xf_assayinfo$plate_id

#add norm_info
xf_raw_pr <- xf_raw_pr %>% dplyr::left_join(xf_norm, by = c("well"))

#add bufferfactor
xf_raw_pr <- xf_raw_pr %>% dplyr::left_join(xf_buffer, by = c("well"))

#add flag well columnn
xf_raw_pr$flagged_well <- FALSE
xf_raw_pr$flagged_well[xf_raw_pr$well %in% xf_flagged] <- TRUE

# select columns that are needed
xf_raw_pr <- xf_raw_pr %>% dplyr::select(
  plate_id, well, measurement, tick, timescale, minutes, group, interval, injection,
  O2_em_corr, pH_em_corr, O2_mmHg, pH, pH_em_corr_corr, O2_em_corr_bkg,
  pH_em_corr_bkg, O2_mmHg_bkg, pH_bkgd, pH_em_corr_corr_bkg, bufferfactor, cell_n, flagged
)

return(xf_raw_pr)
}

preprocess_xf_rate <- function(xf_rate,
                               xf_norm,
                               xf_flagged){
  #add norm_info to rate data
  OCR_from_excel <- xf_rate %>% dplyr::left_join(xf_norm, by = c("well"))

  OCR_from_excel$flagged_well <- FALSE
  OCR_from_excel$flagged_well[OCR_from_excel$well %in% xf_flagged] <- TRUE

  return(OCR_from_excel)
}

rename_columns <- function(xf_raw_pr) {

```

```

# change column names into terms without spaces
colnames(xf_raw_pr) <- c(
  "measurement", "tick", "well", "group",
  "time", "temp_well", "temp_env", "O2_isvalid", "O2_mmHg",
  "O2_light", "O2_dark", "O2ref_light", "O2ref_dark",
  "O2_em_corr", "pH_isvalid", "pH", "pH_light", "pH_dark",
  "pHref_light",
  "pHref_dark", "pH_em_corr", "interval"
)

return(xf_raw_pr)
}

convert_timestamp <- function(xf_raw_pr) {

  # first make sure that the data is sorted correctly
  xf_raw_pr <- dplyr::arrange(xf_raw_pr, tick, well)

  # add three columns to df (totalMinutes, minutes and time) by converting the timestamp i
  xf_raw_pr$time <- as.character(xf_raw_pr$time)
  times <- strsplit(xf_raw_pr$time, ":")
  xf_raw_pr$totalMinutes <- sapply(times, function(x) {
    x <- as.numeric(x)
    x[1] * 60 + x[2] + x[3] / 60
  })
  xf_raw_pr$minutes <- xf_raw_pr$totalMinutes - xf_raw_pr$totalMinutes[1] # first row need
  xf_raw_pr$timescale <- round(xf_raw_pr$minutes * 60)

  return(xf_raw_pr)
}

correct_pH_em_corr <- function(pH_em_corr, pH_cal_em, pH_targetEmission){

  correct_pH_em_corr <- (pH_targetEmission / pH_cal_em) * pH_em_corr
}

calc_background <- function(xf_raw_pr){

  background <- xf_raw_pr %>%
    dplyr::select(group, well, measurement, timescale, O2_em_corr,

```

```

        pH_em_corr, O2_mmHg, pH, pH_em_corr_corr) %>%
dplyr::filter(group == "Background") %>%
dplyr::reframe(
  measurement,
  O2_em_corr_bkg = mean(O2_em_corr),
  pH_em_corr_bkg = mean(pH_em_corr),
  O2_mmHg_bkg = mean(O2_mmHg),
  pH_bkgd = mean(pH),
  pH_em_corr_corr_bkg = mean(pH_em_corr_corr)
)

return(background)
}

print("preprocess_xfplate is ready")

```

### 12.0.3 run\_seahtrue()

```

run_seahtrue <- function(filepath_seahorse){

  filepath_seahorse %>%
  read_xfplate() %>%
  preprocess_xfplate()

}

print("run_seahtrue is ready")

```

## 12.1 run it

For one file

```

library(tidyverse)

```

```

webr::install("logger")
library(logger)

#set file source
root_srcfile <-
  "https://raw.githubusercontent.com/vcjdeboer/"
repository_srcfile <-
  "seahtrue/main/inst/extdata/"
name_srcfile <- "20191219 SciRep PBMCs donor A.xlsx"
output_srcfile <- "VB.xlsx"

#download file and rename to "VB.xlsx"
download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    name_srcfile),
  output_srcfile)

XF <-run_seahtrue("VB.xlsx")

XF

```

For three files

```

library(tidyverse)

webr::install("logger")
library(logger)

#set file source
root_srcfile <-
  "https://raw.githubusercontent.com/vcjdeboer/"
repository_srcfile <-
  "seahtrue/renv-sync_issue_2/inst/extdata/"

#file 1
name_srcfile <- "20191219 SciRep PBMCs donor A.xlsx"
output_srcfile <- "20191219 SciRep PBMCs donor A.xlsx"

```

```

#download file 1
download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    name_srcfile),
  output_srcfile)

# file 2
name_srcfile <- "20200110 SciRep PBMCs donor B.xlsx"
output_srcfile <- "20200110 SciRep PBMCs donor B.xlsx"

#download file 2
download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    name_srcfile),
  output_srcfile)

# file 3
name_srcfile <- "20200110 SciRep PBMCs donor C.xlsx"
output_srcfile <- "20200110 SciRep PBMCs donor C.xlsx"

#download file 3
download.file(
  paste0(
    root_srcfile,
    repository_srcfile,
    name_srcfile),
  output_srcfile)

file_list <- c("20191219 SciRep PBMCs donor A.xlsx",
               "20200110 SciRep PBMCs donor B.xlsx",
               "20200110 SciRep PBMCs donor C.xlsx"
               )

```

Here we use the nice `map` function from `purrr` that we introduced previously:



```
xf_3_read <- file_list %>%
  map_dfr(~run_seahtrue(.x))

xf_3_read
```

... and we plot the time vs O2 for the raw data for the second experiment

```
xf_3_read %>%
  filter(str_detect(plate_id, "V01941")) %>%
  select(plate_id, raw_data)%>%
  unnest(c(raw_data)) %>%
  ungroup()%>%
  select(well, minutes, group, O2_mmHg) %>%
  filter(group == "100.000") %>%
  ggplot(aes(x = minutes, y = O2_mmHg,
             group = well)) +
    geom_point() +
    theme_bw(base_size = 15)+
    facet_wrap(~well)
```

## 12.2 Bioenergetic space plot

For making the bioenergetic space plots that were proposed in the Mookerjee et al. [JBC Quantifying intracellular rates of glycolytic and oxidative ATP production and consumption using extracellular flux measurements](#), we need to do four things:

- Establish buffering power of the culture medium
- Correct ECAR for OCR-derived acidification
- Get the ATP assumptions and formulas into R
- Plot the ATP fluxes in a space plot and derive the indices

The first needs to be done in the lab using established protocols for calculating buffering power, preferably with H2SO4 as acid. The second to fourth are merely some data analysis and plotting in R.

### 12.2.1 Buffering power

We assume you already know what the buffering power is of your particular culture medium and assigned that to the meta info of your experiment before running your seahorse plate. In

that case it is listed in the `bufferfactor` column in our `raw_data` output data. Since each well can theoretically have a different type of culturing medium and thus also a different buffering factor, each well gets its own buffer factor

Let's have a look at the buffering power numbers in the `xf_3_read` data.

```
xf_3_read %>%
  filter(str_detect(plate_id, "V01941")) %>%
  select(plate_id, raw_data )%>%
  unnest(c(raw_data)) %>%
  pull(bufferfactor) %>%
  unique()
```

You will see that if there is 0 and 2.4. Likely, the background wells were assigned zero and the sample wells 2.4.

```
xf_3_read %>%
  filter(str_detect(plate_id, "V01941")) %>%
  select(plate_id, raw_data )%>%
  unnest(c(raw_data)) %>%
  select(well, bufferfactor) %>%
  unique() %>%
  head(12) #or tail(12)
```

That is indeed the case, as you can see if you run the above code.

## 12.2.2 Calculations

### 12.2.2.1 OCR-derived ECAR

We can calculate the contribution of OCR-derived  $H^+$  in multiple ways. Agilent takes an empirical approach and in the Mookerjee paper it is based on enzyme kinetics and steady-state assumptions. For the agilent approach the contribution was empirically determined as 0.61 which was named the CCF (CO<sub>2</sub> contribution factor). For the mookerjee method the factor was 0.953.

### 12.2.2.2 ATP per O2

For the agilent approach they take the P/O ratio as 2.75, which was assumed and empirically tested. For the mookerjee approach, the P/O ratio was assumed to be 2.611, also with a hyperpolarization factor included.

### 12.2.2.3 ATP per lactate

Agilent assumed the ATP/lactate ratio is 1. Although Mookerjee is doing the same, they also take into account that for each each lactate derived from glucose there can be O2 consumption that produced ATP, this is a factor of 0.242.

### 12.2.2.4 Combined

Since these are just transformations of our OCR and ECAR data, we can combine all these factors into three separate factors:

- Jglyco\_ecar\_factor
- Jglyco\_ocr\_factor
- Joxphos\_ocr\_factor

We will also include a scaling factor for data that is not normalized to protein level or cell number. This scaling factor brings the data into the same range as the Mookerjee paper. The estimation of protein concentration in a seahorse well is 10-30 ug per 100000 cells. We thus take 20 ug as an estimate of protein amount in a well.

### 12.2.2.5 Code

Here is a function for calculating and a pipe for plotting the space. The first is `calculate_space`, it takes in the following arguments:

- `rate` => `rate_data` format
- `param_set_ocr` => timepoints for phases ocr
- `param_set_ecar` => timepoints for phases ecar
- `atp_factors` => atp calculation factors
- `ug_protein_scaling_factor` => scaling factor
- `OCR_var` => either `OCR_wave_bc` or `J_oxpos`
- `ECAR_var` => either `ECAR_wave_bc` or `J_glyco`

```
calculate_space <- function(rate,
                             param_set_ocr, param_set_ecar,
```

```

        atp_factors,
        ug_protein_scaling_factor,
        OCR_var, ECAR_var){

SFI <- function(x, y, max_ecar, max_ocr){

  x_0 <- x + y - max_ocr
  y_0 <- x + y - max_ecar
  if(x_0 < 0){x_0 <- 0}
  if(y_0 < 0){y_0 <- 0}
  angle_ocr <- atan(max_ocr/x_0)*(180/pi)
  angle_ecar <- atan(y_0/max_ecar)*(180/pi)
  SFI <- ((angle_ocr - angle_ecar)/90) *100 #angle relative to 90 degrees

  return(SFI)

}

rate <- rate %>%
  mutate(J_glyco = (ECAR_wave_bc * atp_factors$Jglyco_ecar_factor -
                    OCR_wave_bc * atp_factors$Jglyco_ocr_factor)/ug_protein_scaling_fa
        J_oxphos = (OCR_wave_bc * atp_factors$Joxphos_ocr_factor)/ug_protein_scaling_fa

df_ocr <- rate %>%
  select(my_OCR = all_of(OCR_var), everything()) %>%
  summarize(OCR = mean(my_OCR), # this was OCR_wave_bc
            .by = c(group, measurement)) %>%
  pivot_wider(names_from = measurement, names_prefix = "m", values_from = OCR) %>%
  rename(all_of(param_set_ocr)) %>%
  select(group, init_ocr, om_ocr, fccp_ocr, amrot_ocr) %>%
  mutate(non_mito_ocr = amrot_ocr,
         basal_ocr = init_ocr - non_mito_ocr,
         max_ocr = fccp_ocr - amrot_ocr,
         spare_ocr = max_ocr - basal_ocr,
         spare_ocr_index = spare_ocr / max_ocr,
         max_ocr_index = max_ocr/basal_ocr,
         atp_linked = init_ocr-om_ocr,
         proton_leak = om_ocr - amrot_ocr,
         leak_index = proton_leak / basal_ocr,
         coupling_index = atp_linked / basal_ocr)

```

```

df_ecar <- rate %>%
  select(my_ECAR = all_of(ECAR_var), everything() ) %>%
  summarize(ECAR = mean(my_ECAR),
    .by = c(group, measurement)) %>%
  pivot_wider(names_from = measurement, names_prefix = "m", values_from = ECAR) %>%
  rename(all_of(param_set_ecar)) %>%
  select(group, init_ecar, om_ecar, fccp_ecar, amrot_ecar) %>%
  mutate(basal_ecar = init_ecar,
    max_ecar = om_ecar,
    spare_ecar = max_ecar-basal_ecar,
    spare_ecar_index = spare_ecar/max_ecar,
    max_ecar_index = max_ecar/basal_ecar)

df_space <- df_ocr %>%
  left_join(df_ecar, by = c("group")) %>%
  ungroup() %>%
  mutate(bioenergetic_scope = max_ecar+max_ocr, #or max*max
    glyco_index = (basal_ecar/bioenergetic_scope)*100,
    bio_index = ((basal_ecar+basal_ocr)/(max_ecar+ max_ocr))*100, #or basal*basal/m
    glyco_index_max = (max_ecar/bioenergetic_scope)*100,
    supply_index = pmap_dbl(list(basal_ecar, basal_ocr, max_ecar, max_ocr), SFI))

  return(df_space)
}

print("calculate_space is ready")

```

### 12.2.3 Space plot

We set the arguments first for plotting the biospace

```

#mito stress test params
param_set_ocr <- c(init_ocr = "m2",
  om_ocr = "m3",
  fccp_ocr = "m4",
  amrot_ocr = "m9")

param_set_ecar <- c(init_ecar = "m2",
  om_ecar = "m3",
  fccp_ecar = "m4",

```

```

amrot_ecar = "m9")

#brand/mookerjee factors
mookerjee_atp_factors <- list(Jglyco_ecar_factor = 7.23,
                             Jglyco_ocr_factor = 0.469,
                             Joxphos_ocr_factor = 4.6988)

#agilent factors
agilent_atp_factors <- list(Jglyco_ecar_factor = 8.7552,
                             Jglyco_ocr_factor = 0.61,
                             Joxphos_ocr_factor = 5.5)

#for scale comparison to mookerjee JBC (0 - 70 pmol ATP/min/ug)
#estimation of protein concentration in well 10-30 ug per 100000 cells
#20 ug in well approximately
ug_protein_scaling_factor = 20

#either J or wave_bc
ECAR_var <- c("J_glyco")
OCR_var <- c("J_oxphos")

#axis titles for plot
ECAR_title <- c("J ATP glyco (pmol ATP/min/ug protein)")
OCR_title <- c("J ATP mito (pmol ATP/min/ug protein)")

#plot title
plot_title <- "Space plot (mookerjee params)"
plot_title <- "Space plot (agilent params)"

legend_title <- "Group"

##for plotting the raw ECAR/OCR without Jatp conversions:
# ECAR_var <- c("ECAR_wave_bc")
# OCR_var <- c("OCR_wave_bc")
#
# ECAR_title <- c("ECAR (mpH/min)")
# OCR_title <- c("OCR (pmol/min)")
#
# ug_protein_scaling_factor = 1

```

```
# a theme layout setting
webr::install("MetBrewer")
my_cols_two <-
  MetBrewer::met.brewer(name = "Hiroshige", n = 2)

print("arguments are set")
```

Next we set the input rate table argument

```
rate <- xf_3_read %>%
  filter(str_detect(plate_id, "V01941")) %>%
  select(plate_id, rate_data )%>%
  unnest(c(rate_data)) %>%
  ungroup()

print("rate is set")
```

Run the space function

```
df_space <- calculate_space(
  rate %>%
    filter(str_detect(group,
                      "100.000|200.000")),
  param_set_ocr, param_set_ecar,
  agilent_atp_factors,
  ug_protein_scaling_factor,
  OCR_var, ECAR_var
)

print("df_space is ready")
```

Plot the df\_space

```
#for two groups only (because of #colors)
df_space %>%
  mutate(max_axis =
    case_when(
```

```

    fccp_ocr>=amrot_ecar ~ fccp_ocr+fccp_ocr*0.1,
    fccp_ocr<amrot_ecar ~ amrot_ecar+amrot_ecar*0.1)) %T>%
{my_max <- max(.$max_axis)} %>%
arrange(desc(group)) %>%
select(group,basal_ocr,fccp_ocr,
        basal_ecar, amrot_ecar) %>%
ggplot(aes(xmin = 0, ymin = 0,
            ymax = fccp_ocr, xmax = amrot_ecar,
            fill = forcats::fct_rev(group)))+
geom_rect(alpha = 0.5, color ="black")+
geom_segment(aes(x=0, xend=amrot_ecar,
                 y=0, yend=fccp_ocr),
             color="grey10", linetype = "dashed")+
geom_point(aes(x = basal_ecar, y = basal_ocr,
               fill = group),
           color = "black", pch = 21, size = 4)+
scale_colour_manual(values = my_cols_two) +
scale_fill_manual(
  values = my_cols_two %>% colorspace::lighten(.2),
  name = legend_title)+
labs(title = plot_title,
      x = ECAR_title,
      y = OCR_title)+
#coord_fixed(ratio = 1,
#            xlim = c(0,my_max), ylim = c(0,my_max),
#            expand = FALSE, clip = "on")+
theme_classic()

```

## 12.2.4 Gauge plots

```

plot_multi_space_gauge <- function(df_space,
                                   my_param_to_plot,
                                   my_colors,
                                   my_gauge_title,
                                   my_gauge_subtitle,
                                   my_label_text_lo,
                                   my_label_text_hi){

  my_param_df <- df_space %>%

```



```

    select(param = all_of(my_param_to_plot),
           group) %>%
    mutate_if(is.numeric, round)

number_of_plots <- nrow(my_param_df)

plot_list <- list()
for (i in 1:number_of_plots){

  my_param_value <- my_param_df %>% pluck("param", i)
  my_axis_title <- my_param_df %>% pluck("group", i)

  plot_list[[i]] <- plot_one_space_gauge(my_param_value,
                                         my_colors,
                                         my_axis_title,
                                         my_label_text_lo,
                                         my_label_text_hi)

}

multi_plot <- patchwork::wrap_plots(plot_list) +
  plot_annotation(
    title = my_gauge_title,
    subtitle = my_gauge_subtitle
    #caption = ""
  ) & theme(text = element_text(size = 25),
            axis.title.x = element_text(size = 20))

return(multi_plot)
}

plot_one_space_gauge <- function(my_param,
                                my_colors,
                                my_axis_title,
                                my_label_text_lo,
                                my_label_text_hi){

```

```

breaks <- c(0, my_param, 100)

#gauge chart from
#https://stackoverflow.com/questions/24900903/how-to-draw-gauge-chart-in-r
get_poly <- function(a,b,r1=0.5,r2=1.0) {
  th.start <- pi*(1-a/100)
  th.end   <- pi*(1-b/100)
  th       <- seq(th.start,th.end,length=100)
  x        <- c(r1*cos(th),rev(r2*cos(th)))
  y        <- c(r1*sin(th),rev(r2*sin(th)))
  return(data.frame(x,y))
}

plot <- ggplot() +
  geomtextpath::geom_textcurve(
    data = data.frame(x = -1, xend = 0, y = 0, yend = 1),
    aes(x, y, xend = xend, yend = yend),
    hjust = 0.2,
    curvature = -0.4,
    label = my_label_text_lo,
    size = 6,
    vjust = -1,
    color = my_colors[1])+
  geomtextpath::geom_textcurve(
    data = data.frame(x = 0, xend = 1, y = 1, yend = 0),
    aes(x, y, xend = xend, yend = yend),
    hjust = 0.8,
    curvature = -0.4,
    angle = 90,
    label = my_label_text_hi,
    size = 6,
    vjust = -1,
    color = my_colors[2]
  )+
  geom_polygon(data=get_poly(breaks[1],
                             breaks[2]),
              aes(x,y),fill= my_colors[1])+
  geom_polygon(data=get_poly(breaks[2],
                             breaks[3]),
              aes(x,y),fill= my_colors[2])

```

```

        aes(x,y),fill= my_colors[2]))+
geom_polygon(data=get_poly(my_param-0.2,
                           my_param+0.2,
                           r1 = 0.5,
                           r2 = 1.1),
             aes(x,y))+
geom_text(data=as.data.frame(breaks),
          size= 6 , vjust=0,
          aes(x=1.1*cos(pi*(1-breaks/100)),
              y=1.125*sin(pi*(1-breaks/100)),
              label=paste0(breaks,"%")))+
labs(#title = my_gauge_title,
     x = my_axis_title )+
#coord_fixed()+
coord_fixed(ratio = 1,
            xlim = c(-1.2,1.2), ylim = c(0,1.2),
            expand = TRUE, clip = "on")+
theme_void()+
theme(plot.title = element_text(size = rel(0.2),
                                hjust = 0, vjust = -1),
      axis.title.x = element_text(size = rel(0.2),
                                vjust = 1)
)

return(plot)

}

print("multi_gauge functions are ready")

```

```

webr::install("geomtextpath")
webr::install("patchwork")
library(patchwork)

hiroshige_cols_four <- c("#ef8a47", "#ffd06f",
                        "#72bcd5", "#376795")
my_doughnut_cols_lightblue <- c(hiroshige_cols_four[3],
                                "#D1D3D4")
my_colors <- my_doughnut_cols_lightblue
my_param_to_plot <- "supply_index"

```

```
my_gauge_title <- "Supply index"
my_gauge_subtitle <- "Describes the flexibility in switching between glyco and mito."
my_label_text_lo <- "lo"
my_label_text_hi <- "hi"

plot_multi_space_gauge(df_space %>%
  slice(1:4),
  my_param_to_plot,
  my_colors,
  my_gauge_title,
  my_gauge_subtitle,
  my_label_text_lo,
  my_label_text_hi)
```