Lab 2: Combining and Visualizing Data Sets

Lab Outcomes

1. Combine the discrete and CTD data into one data set
2. Visualize the data in a few different ways.

How to approach a computational research question

1. Start with a question

When starting a research project, we often start with a great question, e.g.

How does picoplankton concentration change with temperature?

But, before we can even answer that question, we have to think about how we are going to *wrangle* our data into a form that makes it *easy* to answer that question.

In this case, we are going to have to combine data from two different datasets: the discrete data and the CTD data from the DaRTS data.

There are multiple ways to merge the datasets depending on the exact combination of variables you want e.g. do you want all the stations and/or all the depths and/or only one station and/or multiple years, etc.

To figure out the way to do the merge, we need to break down the process into the most basic/small steps. This is good practice for any kind of code development: breaking down your task into discrete, bitesize bundles. Think about building a robot to make a sandwich. You couldn't just code "make a sandwich", you'd need to break it down into steps, like a recipe e.g. go to the cupboard, take out the bread, put it on the table, etc. Some of those steps may seem obvious to you, but it wouldn't be to a robot.

2. *How* are you going to answer your question?

Here, we started with the question:

How does picoplankton concentration change with temperature?

Next, we need to think about ***how*** to answer this question i.e. ***what form does our data need to be in to answer this question?***

In the case of picoplankton concentration and temperature, we can envision making a graph with picoplankton concentration on the x-axis, and temperature on the y-axis. So we need to make a table of data where we have columns with identifying information (e.g. cruise, date, station number, depth), the picoplankton concentration and the temperature. Then, from that data table (or `data frame` in R talk) we can plot our data.

3. *What* general data formatting steps do you need to take?

We've decided we need a dataframe that includes some type of identifying information, plus picoplankton concentration and temperature. ***What*** do we need to do to our data to get it into that form?

We need to match the discrete and CTD data based on key identifying information. In this case, that information is:

1. The cruise date (for the DaRTS data, the different cruises are identified by the date),
2. The station number,
3. Depth (depending on your exact end goal / question, you'll need to think a bit carefully about how we want to match by depth).

4. Make a step-by-step plan

With the above information, we can make a step-by-step plan for the data *wrangling* (or formatting) that we need to do. Sometimes one of these steps ends up getting broken down into even more steps at a later point when you realise something may not be as straightforward as you first thought.

Step-by-step plan:

1. Import CTD data from csv
2. Make discrete data csv
3. Import discrete data from csv
4. Check and reformat dates in each dataset if necessary
5. Check and reformat depths in each dataset if necessary
6. Combine (merge) the discrete and CTD data
7. Make plot of picoplankton concentration vs temperature

Now we have our plan, we can start doing it!

Importing the data sets

Now we have our two data sets saved as separate CSV files. Let's load each of them into R. But first... we need to start our R session by

1. Selecting our R project in RStudio
2. Importing the packages we need for today

To select our project, you'll need to either click the icon in the right-hand corner of RStudio and select the R Project you started last session, or go to File > Recent Projects > {your RLab project name}

Now to load the libraries:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)
```

You may need to install this package before loading it:

```
##
## Attaching package: 'kableExtra'

## The following object is masked from 'package:dplyr':
##
##     group_rows
```

Ok, so now we are ready to import our data sets. We are going to load each one into a separate data frame:

```
ctdData <- read.csv('DaRTS_CTD_data.csv')
discreteData <- read.csv('DaRTS_discrete_data.csv')
```

Combining data sets

In our plan, we identified the important columns to merge on (cruise, date and depth), and we decided we needed to check the format of the date and depth data in each of our datasets.

Dates

We know one of our key identifying pieces of information is date. So let's take a closer look at the dates.

First, let's pull out just the dates column:

```
# one way
ctdDates <- ctdData$Date

# another way:
ctdDates <- ctdData[['Date']]

# and the discrete dates
discreteDates <- discreteData$Date
```

In the above two ways of pulling out the dates column we ended up with a vector object.

You can also pull out the dates column and have it as a 1 column data frame:

```
ctdDates_df <- ctdData['Date']
```

What format are these dates in? You can check by manually opening the two data frames (`ctdData` and `discreteData`) and looking at the Dates column (i.e. double click on the data frames in the Environment Panel). Or we can check using R:

```
ctdDates[1]
```

```
## [1] 20120911
```

What type of data is this? A number?

```
class(ctdDates[1])
```

```
## [1] "integer"
```

Numbers that do not contain decimal values have a data type as an integer.

What about the discrete data dates?

```
discreteDates[1]
```

```
## [1] "9/11/12"
```

```
class(discreteDates[1])
```

```
## [1] "character"
```

A character can be a letter/number or a combination of letters and/or numbers enclosed by quotes.

So in each case, the dates are different formats and different data types. Let's make them the same format and data type - this will make things much easier for matching the discrete and CTD data.

Should we have our dates represented by numeric values? Or characters?

If we use characters, we need to be careful about knowing which part of the date is the month, day and year: in the US it's normal for dates to be month/day/year, but in almost every other country it's day/month/year. This can lead to a lot of confusion when working collaboratively with international participants, or even within a research group if someone from Scotland joins a team of Americans! In addition to the confusion about the day-month-year order, when we use characters, it's really difficult to do any date math e.g. counting how many days between two dates.

This may make it seem like numbers are a better option - but this can still lead to difficulties when wanting to do date math, e.g. if you want to know how many days there are between dates that span multiple months, you need to know how many days are in each of those months, and if one of those months is February, are you in a leap year?

As you can see, we need to approach dates (& times) carefully. Most programming languages have their own class of data for dates & times (i.e. not numeric or characters). So a lot of those difficulties I mentioned are dealt with behind the scenes by the different date math functions that exist.

R deals with dates and times in a couple of different ways. We're going to use the `lubridate` package which is part of the `tidyverse`. There's this great cheat sheet which gives an overview of a lot of the functionality of the `lubridate` package.

Date reformatting

Let's first load the `lubridate` package

```
library("lubridate")
```

```
##
## Attaching package: 'lubridate'
```

```
## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
```

Now, let's deal with the CTD dates. We can use the `ymd` function to tell `lubridate` that our date is currently a number stored in the format yyyymmdd.

```
ctdDates <- lubridate::ymd(ctdData$Date)
```

What do these look like?

```
ctdDates[1]
```

```
## [1] "2012-09-11"
```

And what data type are they?

```
class(ctdDates)
```

```
## [1] "Date"
```

OK great - now what about the discrete dates? We can't use the same approach as above because our discrete dates are characters of the form mm/dd/yyyy. But we can use something really similar:

```
discreteDates <- lubridate::mdy(discreteData$Date)
```

And let's just check what these dates look like and their type:

```
discreteDates[1]
```

```
## [1] "2012-09-11"
```

```
class(discreteDates)
```

```
## [1] "Date"
```

Great - now we have nicely formatted dates that we can use later to match our CTD and discrete data together. To make this task easier, we want to include a new column in each of our data frames that contains these newly formatted dates. To do this, we are going to use `dplyr`'s `mutate` function. We used `filter` from `dplyr` last day. `dplyr` is another part of the `tidyverse` that is for data frame manipulation.

Adding new columns to a data frame with `mutate`

In the following, we are going to add a new column to our data frames with these newly formatted dates. In each case, the new column will have the name `Date`. Note if a column already exists with the name you are assigning in the mutate function, the original column will be overwritten. So here, we are overwriting the original dates, with the newly formatted dates.

```
ctdData <- mutate(ctdData, Date = lubridate::ymd(ctdData$Date))

discreteData <- mutate(discreteData, Date = lubridate::mdy(discreteData$Date))
```

To do any type of data frame manipulations, you can use *pipes*. Pipes are a recent addition to R. They let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same dataset. Pipe have two symbols to choose from, the symbol `%>%` or `|>`. We'll make full use of pipes later, but to add the dates column to the ctdData data frame using pipes you would do:

```
ctdData <- ctdData %>% mutate(Date = lubridate::ymd(ctdData$Date))
```

This is equivalent to `ctdData <- mutate(ctdData, Date = ymd(ctdData$Date))`.

Tip for working with dates I recommend including in your datasets separate columns for the year, month and day, as well as one column for the combined date. This is really useful when you start to do your analysis and you want to look at the data from one year, or one month or even one day! Let's do that now for our data

```
ctdData <- ctdData %>%
  mutate(year = year(ctdData$Date),
         month = month(ctdData$Date),
         day = day(ctdData$Date))

discreteData <- discreteData %>%
  mutate(year = year(discreteData$Date),
         month = month(discreteData$Date),
         day = day(discreteData$Date))
```

The other useful way to represent the date is day of year, it can be particularly useful for certain analyses. Let's also include a day of year column

```
ctdData <- ctdData %>%
  mutate(doy = yday(ctdData$Date))

discreteData <- discreteData %>%
  mutate(doy = yday(discreteData$Date))
```

Depths

The depth is another key piece of identifying information for merging our datasets.

What do the depths look like for each dataset?

Let's print out the first 10 depths of the CTD data:

```
ctdData$Depth_m[1:10]
```

```
##  [1] 1.363 1.346 1.345 1.354 1.360 1.362 1.355 1.342 1.342 1.356
```

And of the discrete data:

```
discreteData$Depth[1:10]
```

```
##  [1]  2 10 30  2 10 40  2 10 40 NA
```

The discrete measurements are at well defined depth intervals, but the CTD measurements were continuous - so how are we going to match these?

We need to make some assumption - here, we're going to *bin* the CTD data to the nearest meter. That is, we are going to take all the data within a depth interval (also known as a bin), and take the mean value of the data within that bin. Here, our depth interval will be 1 m, i.e. we'll bin our CTD data such that we end up with data at depths of `1, 2, 3, 4, 5, ...` meters.

Binning using `group_by` and `summarize`

This process of binning data is an example of the split-apply-combine paradigm: split the data into groups, apply some analysis to each group, and then combine the results. Many data analysis tasks can be approached using this technique.

In the case of binning:

- split the data into groups: we're splitting our data into groups based on the depth.
- apply some analysis to each group: we're taking the mean of the data in each of our groups
- combine the results: we're putting all the meaned data back into a new dataframe.

Here, we're going to make use of pipes too.

First, we need to round our depths to the nearest meter, then we can bin them.

```
ctdData <- ctdData %>% mutate(Depth = round(ctdData$Depth_m))
```

Now for the binning.

```
ctdDataBinned <- ctdData %>%
  group_by(Depth) %>%
  summarize_all(mean)
```

Let's take a look at our final dataframe, we'll open it from the Environment Pane. On first glance, this looks like it's done what we want - we've only got depths of rounded meters. But, if we look at the other columns, things don't look quite right. The stations are no longer sensible station numbers, and there's quite a lot of data missing.

What we've actually done is calculated the mean profile for all the CTD data together, *rather than the mean profile for the CTD data at every station on every cruise.* So we need to split our data using more criteria. Also, when we calculated the mean, our missing data got in the way. We need to tell R to ignore the missing data, and just calculate the mean on the data that are there.

```
ctdDataBinned <- ctdData %>%
  group_by(Date, Station, Depth) %>%
  summarize_all(mean, na.rm = TRUE)
```

Merging the discrete and CTD data

Now we've got dates & depths in the same format in our discrete and CTD datasets, we can merge them together.

Dataframe merging is a common task in many data science applications, and there is some terminology involved to describe the different ways you can merge the dataframes.

Dataframe merging approaches

Mutating joins allow you to combine variables from multiple tables. There are four types of mutating join, which differ in their behaviour when a match is not found. We'll illustrate each with an example. Let's start by making two dataframes:

```
##   x y
## 1 1 2
## 2 2 1
```

x	y
1	2
2	1

```
df2 <- data.frame(x = c(3, 1), a = 10, b = "a")
df2 %>% knitr::kable() %>% kable_styling(full_width = FALSE)
```

x	a	b
3	10	a
1	10	a

`inner_join(df1, df2)` Only includes observations that match in both `df1` and `df2`.

```
df_innerjoin <- df1 %>% inner_join(df2)
```

```
## Joining with `by = join_by(x)`
```

```
df_innerjoin  %>%  knitr::kable() %>% kable_styling(full_width = FALSE)
```

x	y	a	b
1	2	10	a

Here, R was clever, and merged on the common column between the two dataframes.

`left_join(df1, df2)` Includes all observations in `df1`, regardless of whether they match or not. This is the most commonly used join because it ensures that you don't lose observations from your primary table.

```
df_leftjoin <- df1 %>% left_join(df2)
```

```
## Joining with `by = join_by(x)`
```

```
df_leftjoin  %>%  knitr::kable() %>% kable_styling(full_width = FALSE)
```

x	y	a	b
1	2	10	a
2	1	NA	NA

`right_join(df1, df2)` Includes all observations in `df2`. It's equivalent to `left_join(df2, df1)`, but the columns and rows will be ordered differently.

```
df_rightjoin <- df1 %>% right_join(df2)
```

```
## Joining with `by = join_by(x)`
```

```
df_rightjoin  %>%  knitr::kable() %>% kable_styling(full_width = FALSE)
```

x	y	a	b
1	2	10	a
3	NA	10	a

In this case, we called `df1` first, so it's columns are listed first in the merged dataframe.

```
df2_leftjoin <- df2 %>% left_join(df1)
```

```
## Joining with `by = join_by(x)`
```

```
df2_leftjoin  %>%  knitr::kable() %>% kable_styling(full_width = FALSE)
```

x	a	b	y
3	10	a	NA
1	10	a	2

In this case, `df2` was listed first, so it's columns are first in the merged dataframe.

`full_join()` Includes all observations from `df1` and `df2`.

```
df_fulljoin <- df1 %>% full_join(df2)
```

```
## Joining with `by = join_by(x)`
```

```
df_fulljoin  %>%  knitr::kable() %>% kable_styling(full_width = FALSE)
```

x	y	a	b
1	2	10	a
2	1	NA	NA
3	NA	10	a

The left, right and full joins are collectively know as **outer joins**. When a row doesn't match in an outer join, the new variables are filled in with missing values.

Back to the discrete and CTD data

What type of join do we want to do here?

Ultimately, we want to look at picoplankton and temperature data, so we're only interested in places were there are data in both those columns: so that might imply an `inner_join`. **But** we are merging the *whole* of the discrete dataset (not just the picoplankton) and the *whole* of the binned CTD dataset (not just the temperature). **And** we are merging on date, depth and station. Thus, we want to include *all the observations* from each dataset, then filter out the data we're interested in (e.g. temperature and picoplankton concentration). To include all the observations, we need to do a full join:

```
combinedData <- ctdDataBinned %>%
  full_join(discreteData)
```

```
## Joining with `by = join_by(Date, Station, Depth, year, month, day, doy)`
```

Final step is to save our new dataframe as a CSV file. Then you can use this dataset to answer your own questions without having to redo the whole lab.

```
write.csv(combinedData,"DaRTS_combined_data.csv")
```

Visualizing Datasets

Now we've merged our data - let's take a look at it!

The question we posed at the start of the lab was

How does picoplankton concentration change with temperature?

One way to answer this question is to look at a plot of one against the other:

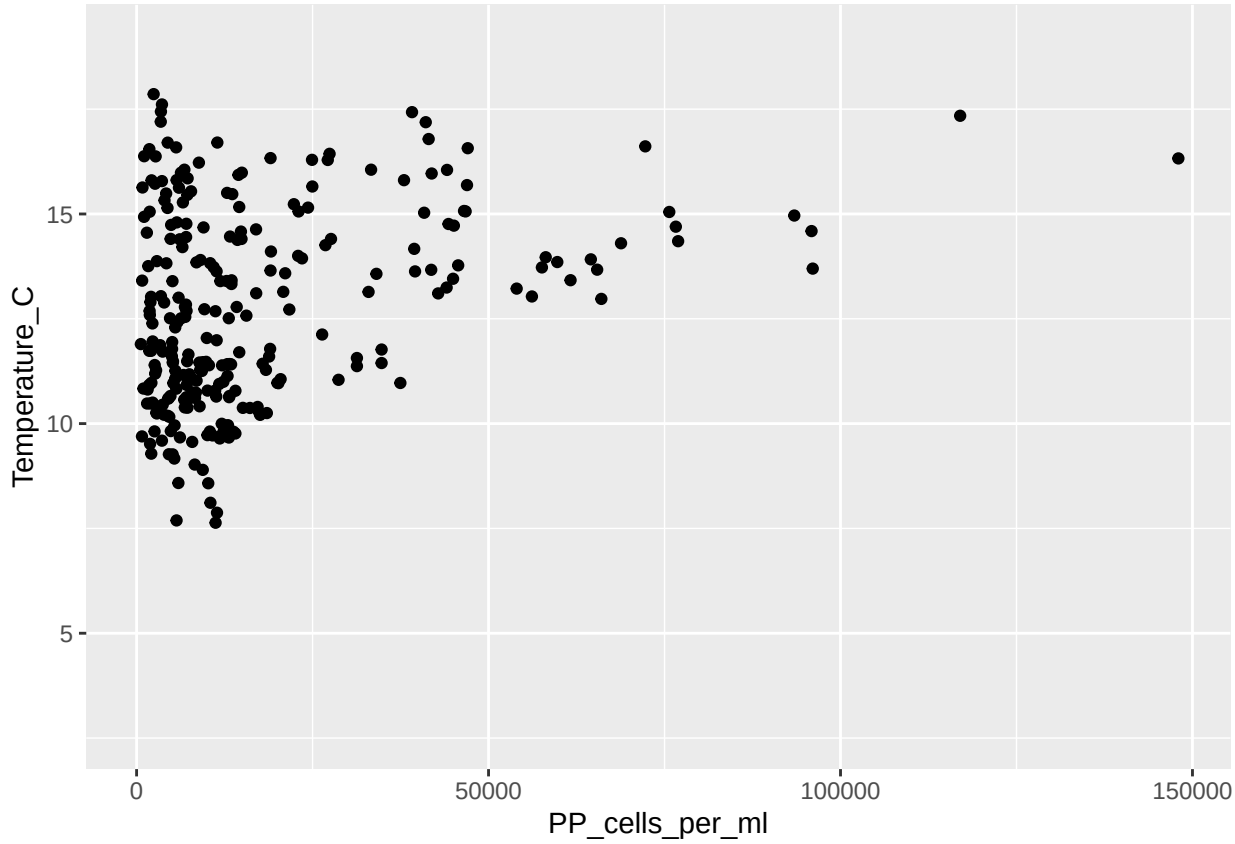
```
combinedData |> colnames()
```

```
##  [1] "Date"                "Station"             "Depth"
##  [4] "Depth_m"             "Conductivity"        "Temperature_C"
##  [7] "Salinity_PSU"        "Density"             "PAR"
## [10] "Fluorescence"        "TurbidityNTU"        "BeamC"
## [13] "O2Conc"              "O2Saturation"        "year"
## [16] "month"               "day"                 "doy"
```

```
## [19] "Cruise"               "Lat"                 "Long"
## [22] "pH"                   "Alk"                 "SiO4_um"
## [25] "NO3_um"               "PO4_um"              "NH4_um"
## [28] "PP_cells_per_ml"      "bac_per_ml"          "vir_per_ml"
## [31] "Chl_ug_per_l"         "CHL_20um_ug_per_l"   "CHL_3um_ug_per_l"
## [34] "Flowcam_Biomass"      "cyanobac_per_ml"     "picoeukaryote_per_ml"
## [37] "noeuk_per_ml"
```

```
ggplot(data = combinedData,
       aes(x=PP_cells_per_ml, y = Temperature_C)) +
  geom_point()
```

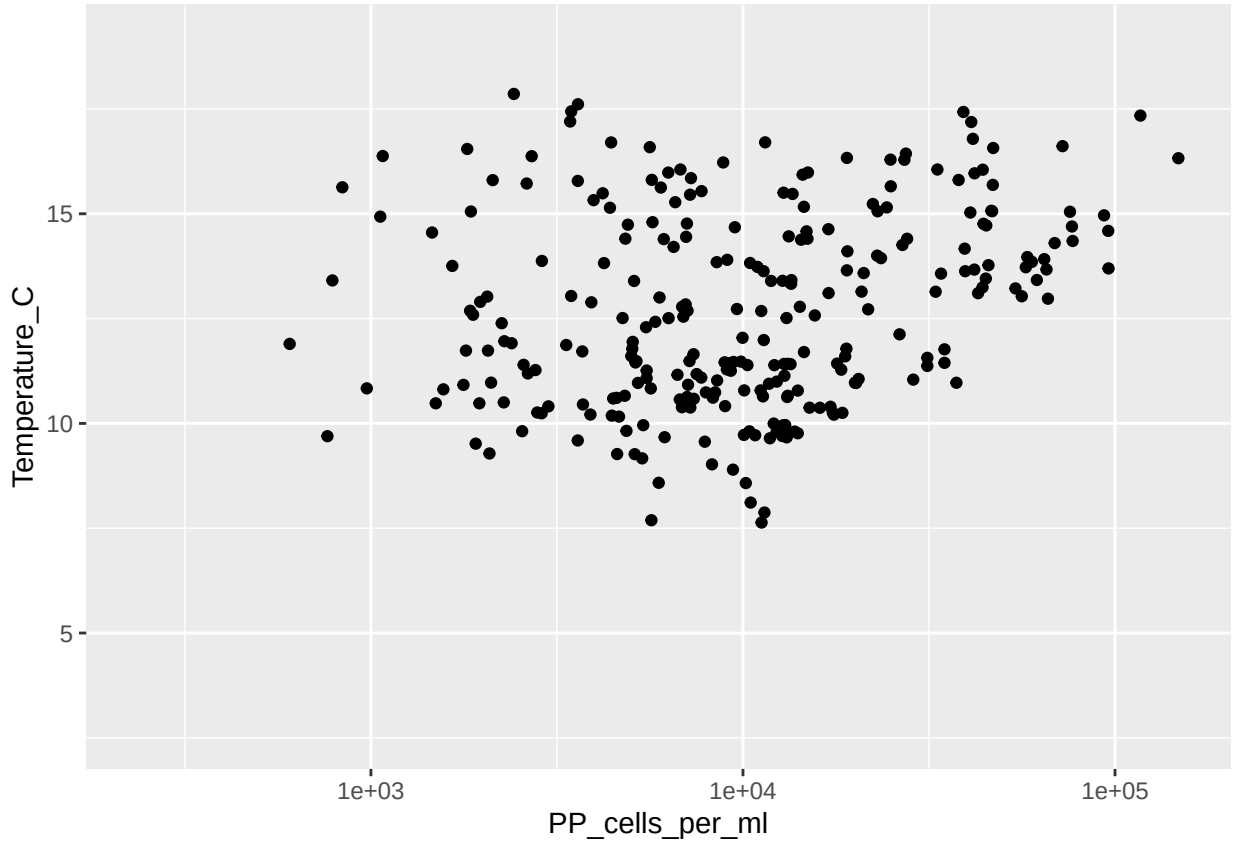
```
## Warning: Removed 8056 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```



There are a lot of data bunched up together at the low picoplankton cell counts, and not so much at the higher concentrations. This kind of distribution is very common in the natural world. When analysing data with this type of distribution, we look at it on a log scale:

```
ggplot(data = combinedData,
       aes(x=PP_cells_per_ml, y = Temperature_C)) +
  geom_point() +
  scale_x_log10()
```

```
## Warning: Removed 8056 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```



Here, we've got data more normally distributed across our x and y axes.

In the next lab we're going to go over how to quantify this kind of relationship using correlations and linear regressions - so we will revisit this. But for now, let's look at visualizing some of the data in a different way.

Visualizing change

Often we want to visualize how a variable has changed over time, be that during one year, at different stations, etc. Here, we're going to make a few plots showing change over different parameters.

How has surface picoplankton concentration changed over the time series?

We need to pull out (i.e. *filter*) the surface measurements for each picoplankton.

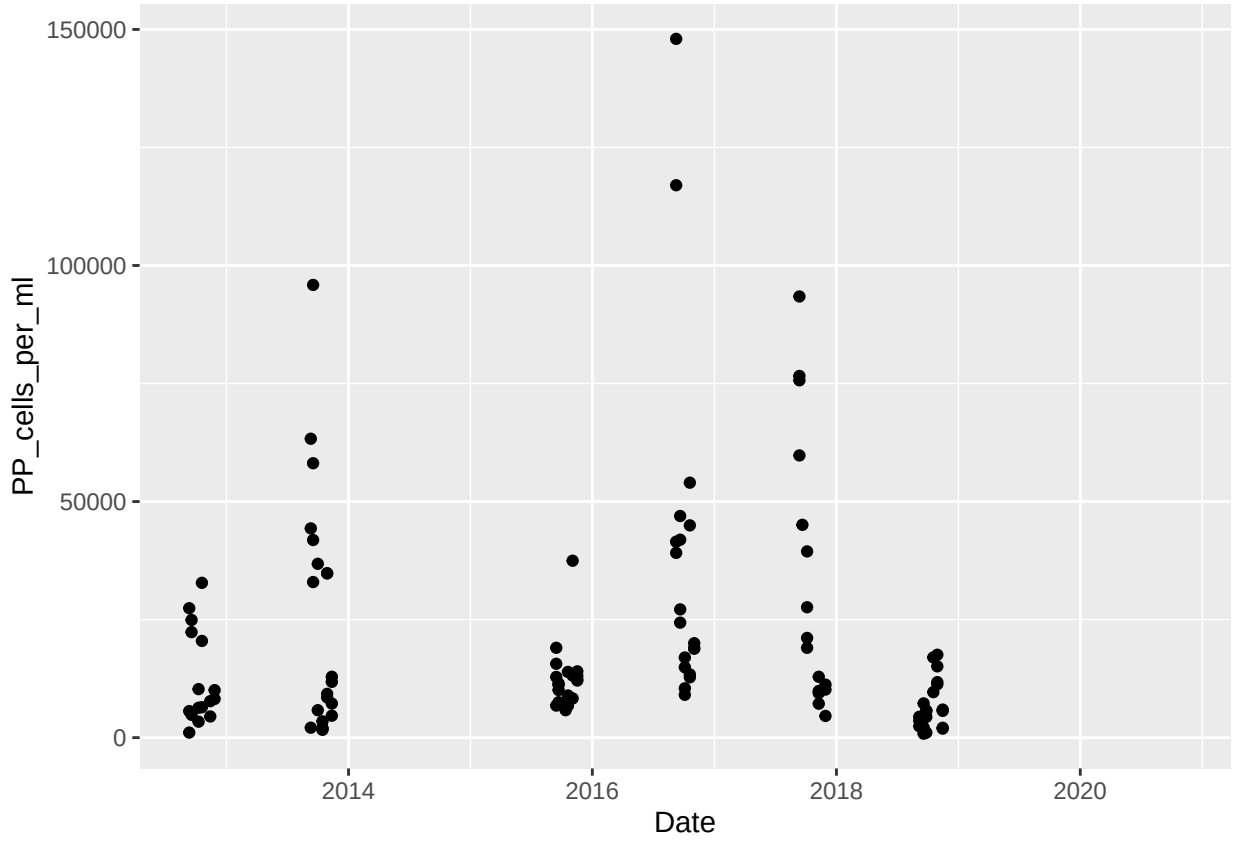
The surface discrete measurements were always taken at 2 m, so we need to filter our dataframe for depth values of 2.

```
surfaceData <- combinedData %>%
  filter(Depth == 2)
```

And now we can plot the data

```
ggplot(surfaceData, aes(x=Date, y= PP_cells_per_ml)) +
  geom_point()
```

```
## Warning: Removed 99 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



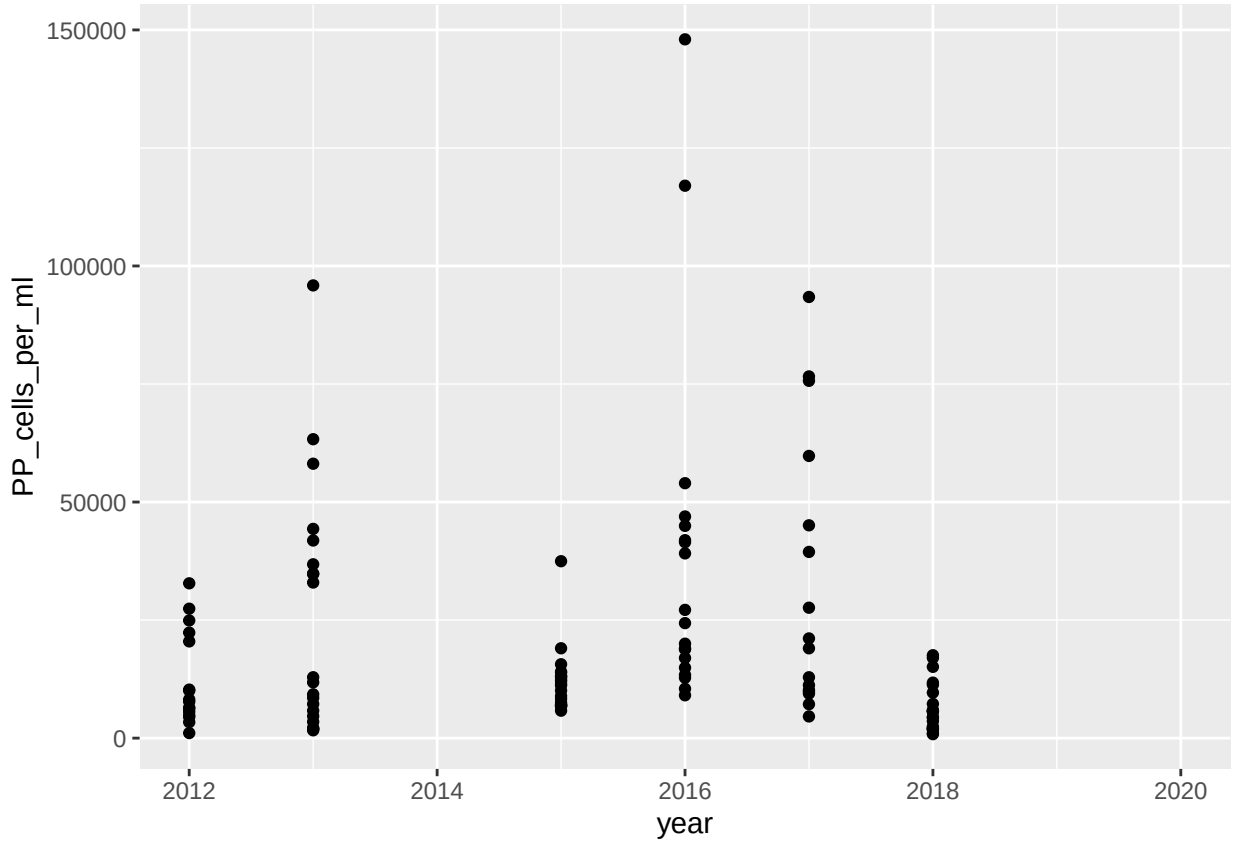
It's a bit hard to tell what is going on here.

How can we make this easier to see what's happening? Well, it depends on what our exact question is e.g.

1) What is the overall change year-on-year? Here, we want to group the data by year. Let's put year on the x-axis:

```
ggplot(surfaceData,
       aes(x=year, y= PP_cells_per_ml)) +
  geom_point()
```

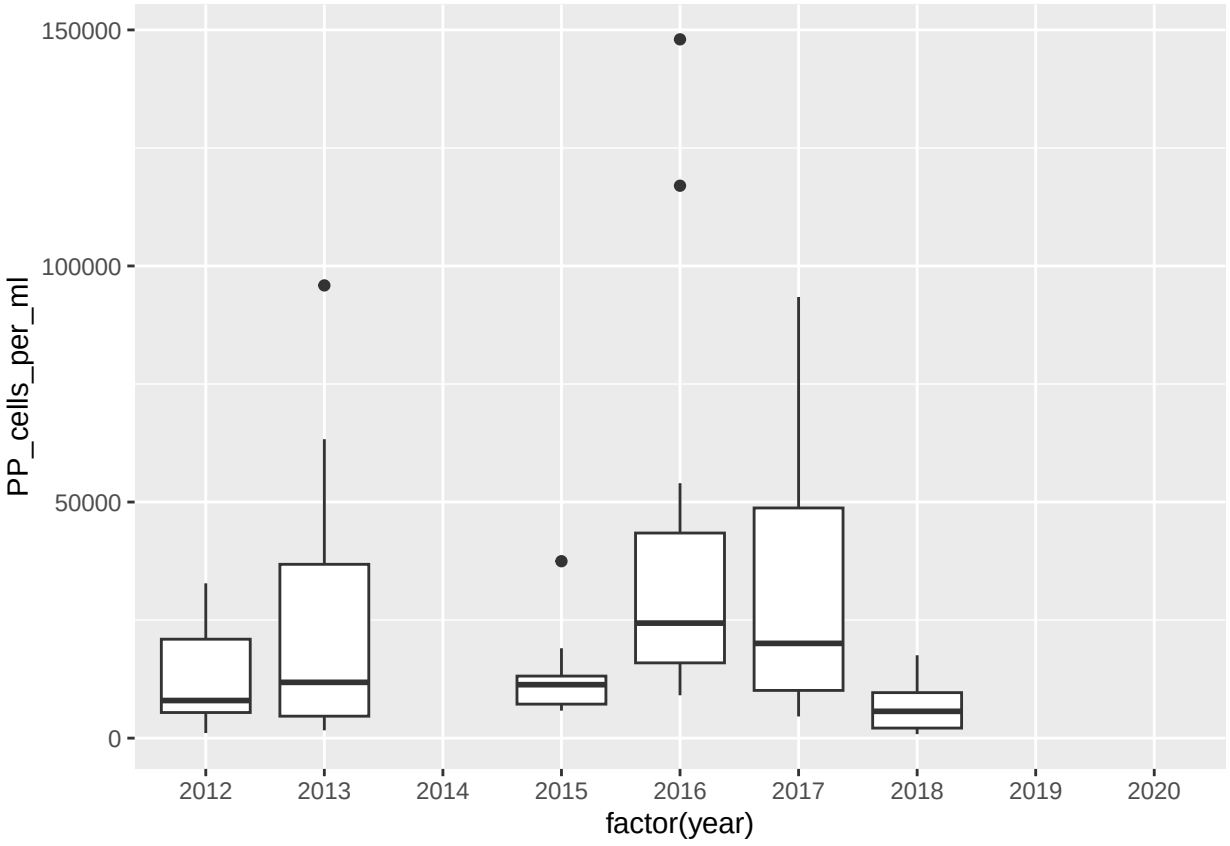
```
## Warning: Removed 99 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



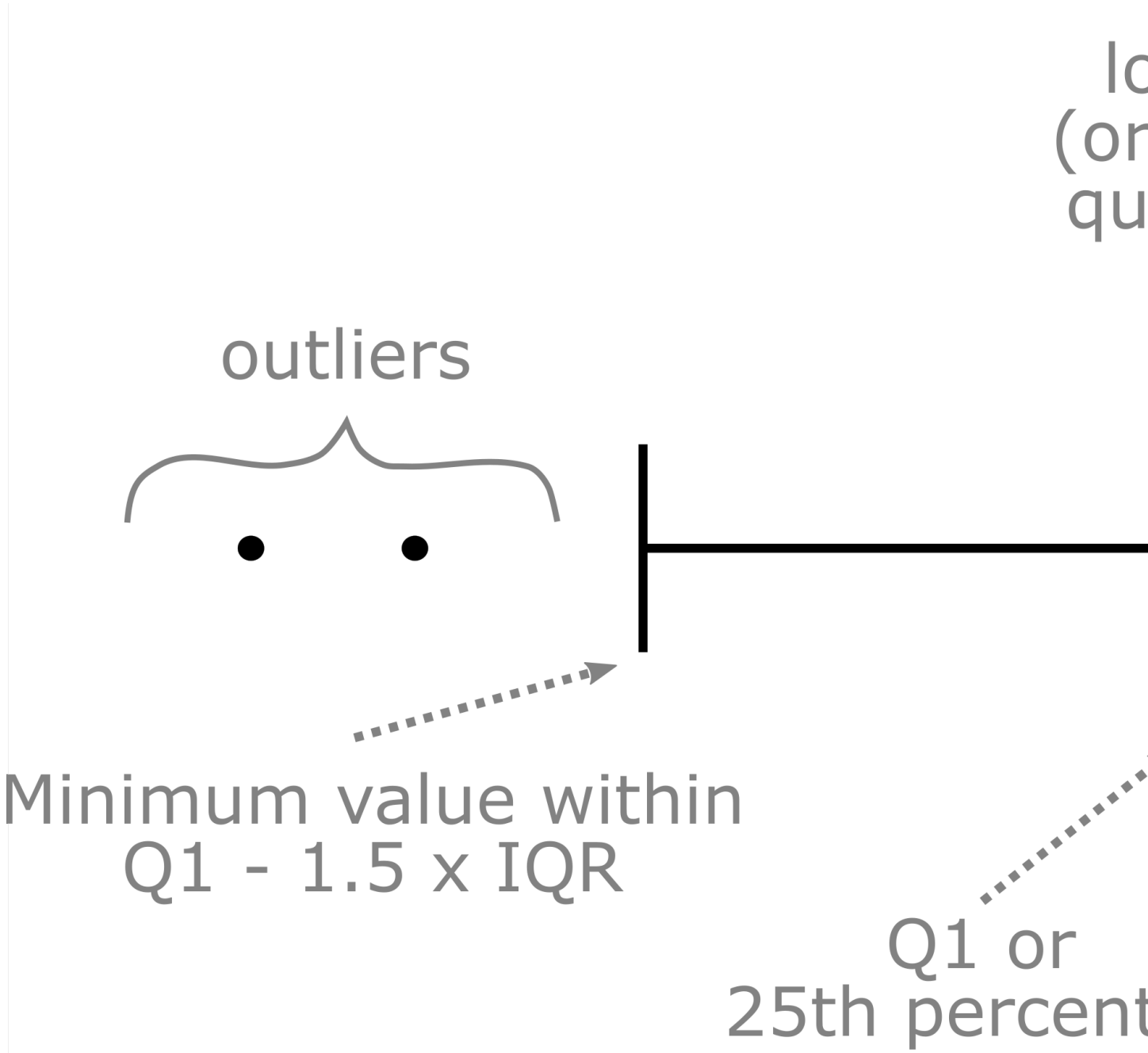
This still isn't great at looking at the year-on-year change. What we can do is create a boxplot for every year to show the distribution in each year. Here, we want to treat the year as a factor, rather than a regular number:

```
ggplot(surfaceData,
       aes(x=factor(year), y= PP_cells_per_ml)) +
  geom_boxplot()
```

```
## Warning: Removed 99 rows containing non-finite outside the scale range
## (‘stat_boxplot()‘).
```



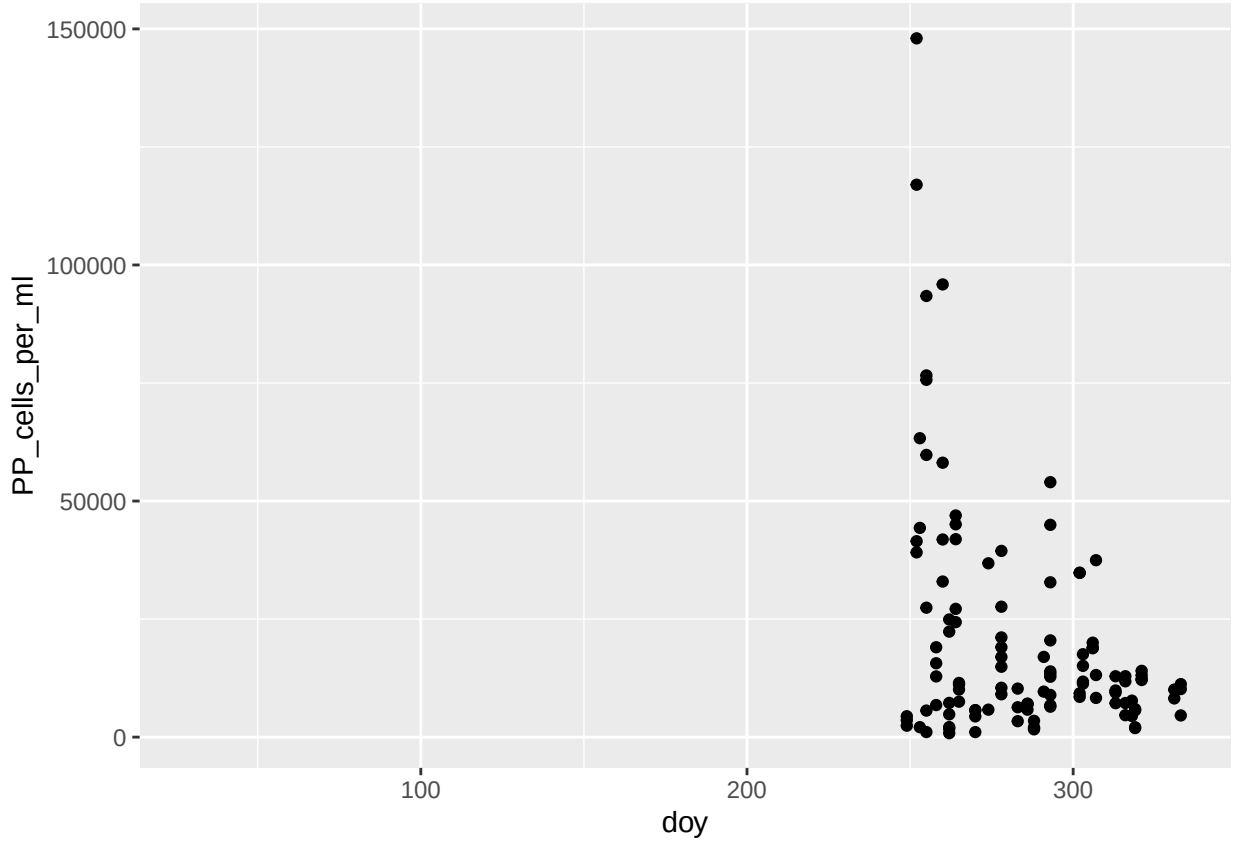
Boxplot explanation The following diagram shows what the different parts of a boxplot made with `geom_boxplot()` represent. Note, the convention for the box is consistent, but often different programming languages, software, or even research field, have different conventions for what the whiskers represent.


lo
(or
qu
outliers
Minimum value within
Q1 - 1.5 x IQR
Q1 or
25th percent

2) What is the mean change within a year? In this case, if we plot the data with day of year on the x-axis, then we can see the yearly variation

```
ggplot(surfaceData, aes(x=doy, y= PP_cells_per_ml)) +
  geom_point()
```

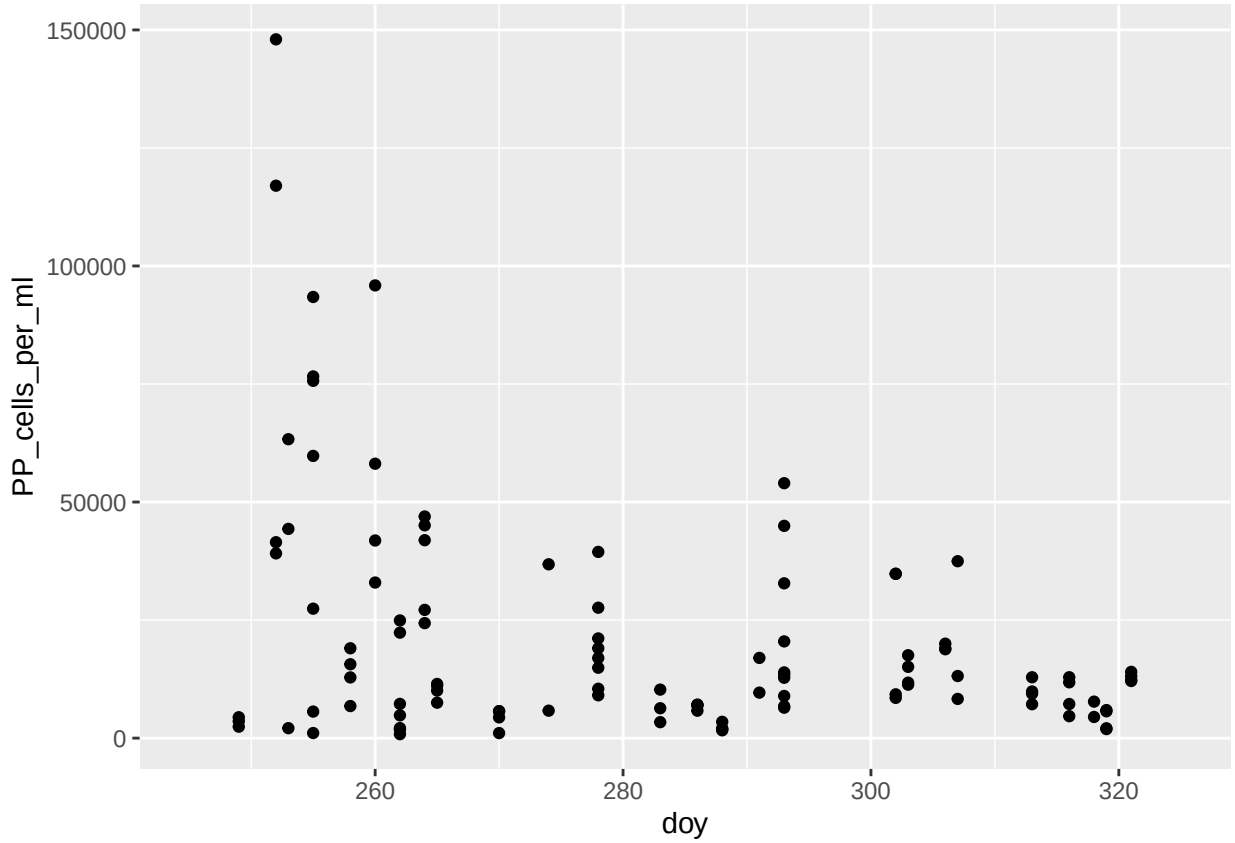
```
## Warning: Removed 99 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```



Let’s change the x-limits:

```
ggplot(surfaceData, aes(x=doy, y= PP_cells_per_ml)) +
  geom_point() +
  xlim(245,325)
```

```
## Warning: Removed 104 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```

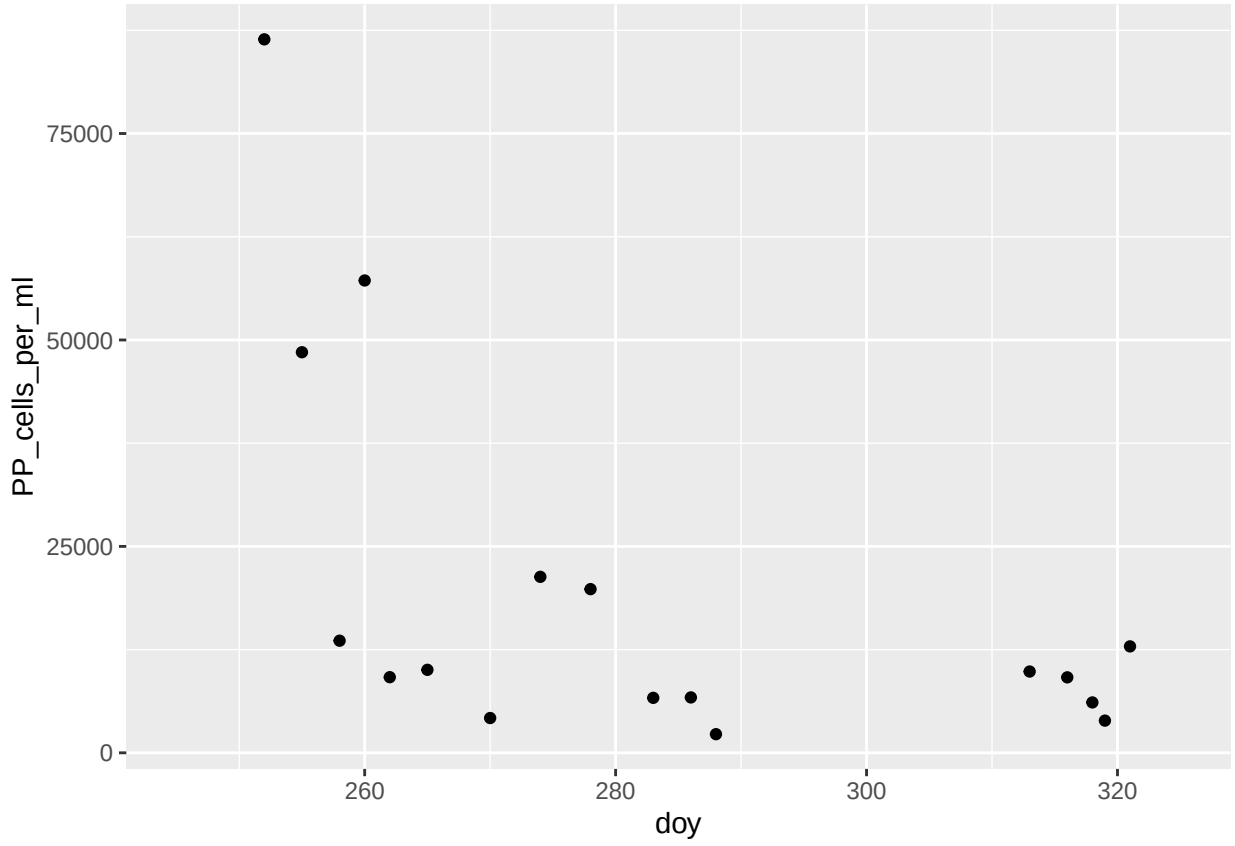


The final step is to calculate and plot the mean seasonal cycle. Note, the following step will give lots of warnings because the latitude and longitude data are stored as characters, not numbers, so when R tries to calculate the mean of those columns, it can't and so assigns the value `NA` to those columns. However, it still can for all the other columns.

```
surfaceData %>% group_by(doy) %>%
  summarize_all(mean) %>%
  ggplot(aes(x=doy, y= PP_cells_per_ml)) +
  geom_point() +
  xlim(245,325)
```

```
## Warning: There were 116 warnings in `summarise()`.
## The first warning was:
## i In argument: `Lat = (function (x, ...) ...`.
## i In group 1: `doy = 29`.
## Caused by warning in `mean.default()`:
## ! argument is not numeric or logical: returning NA
## i Run `dplyr::last_dplyr_warnings()` to see the 115 remaining warnings.
```

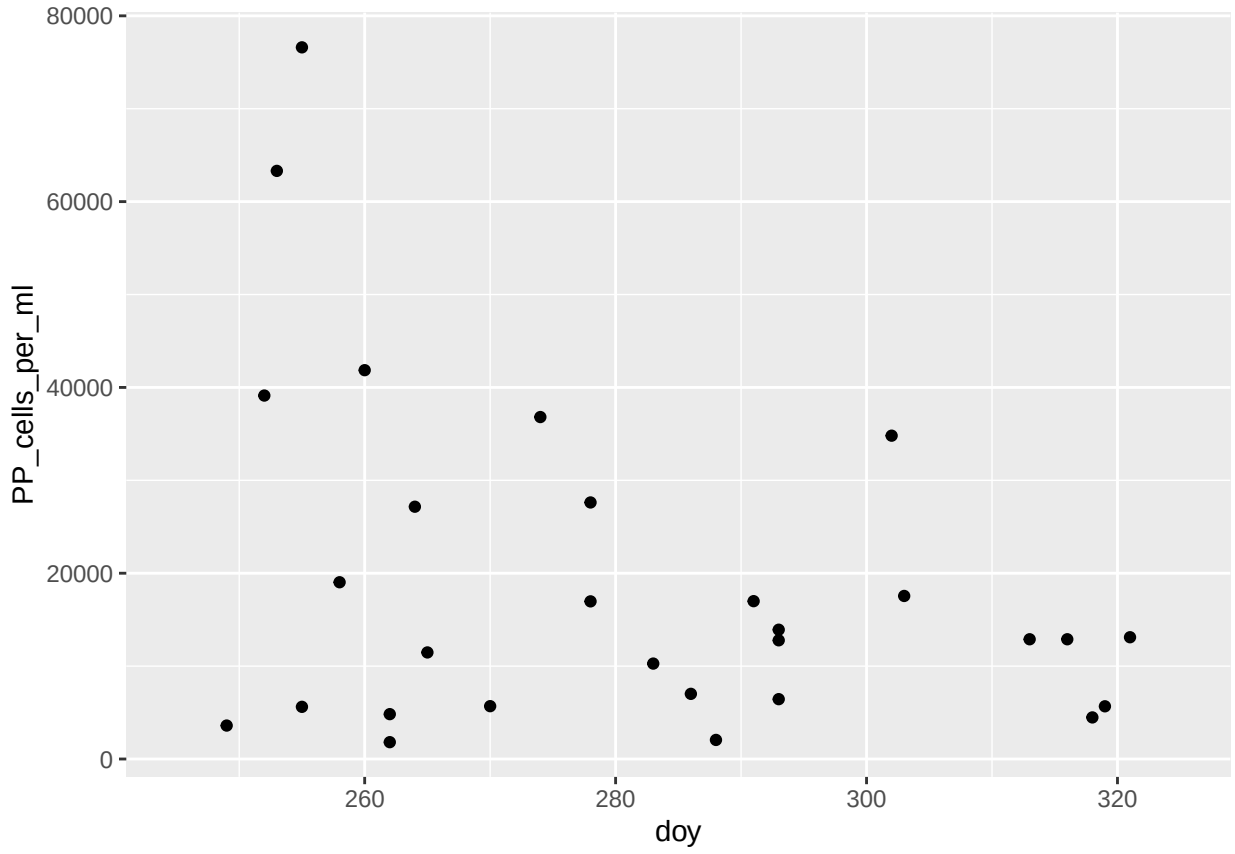
```
## Warning: Removed 41 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



3) What is the seasonal change at Station 1? Here we need to filter for station 1 data only, then plot by day of year as in the previous example.

```
surfaceData %>% filter(Station == 1) %>%
  ggplot(aes(x=doy, y= PP_cells_per_ml)) +
  geom_point() +
  xlim(245,325)
```

```
## Warning: Removed 35 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

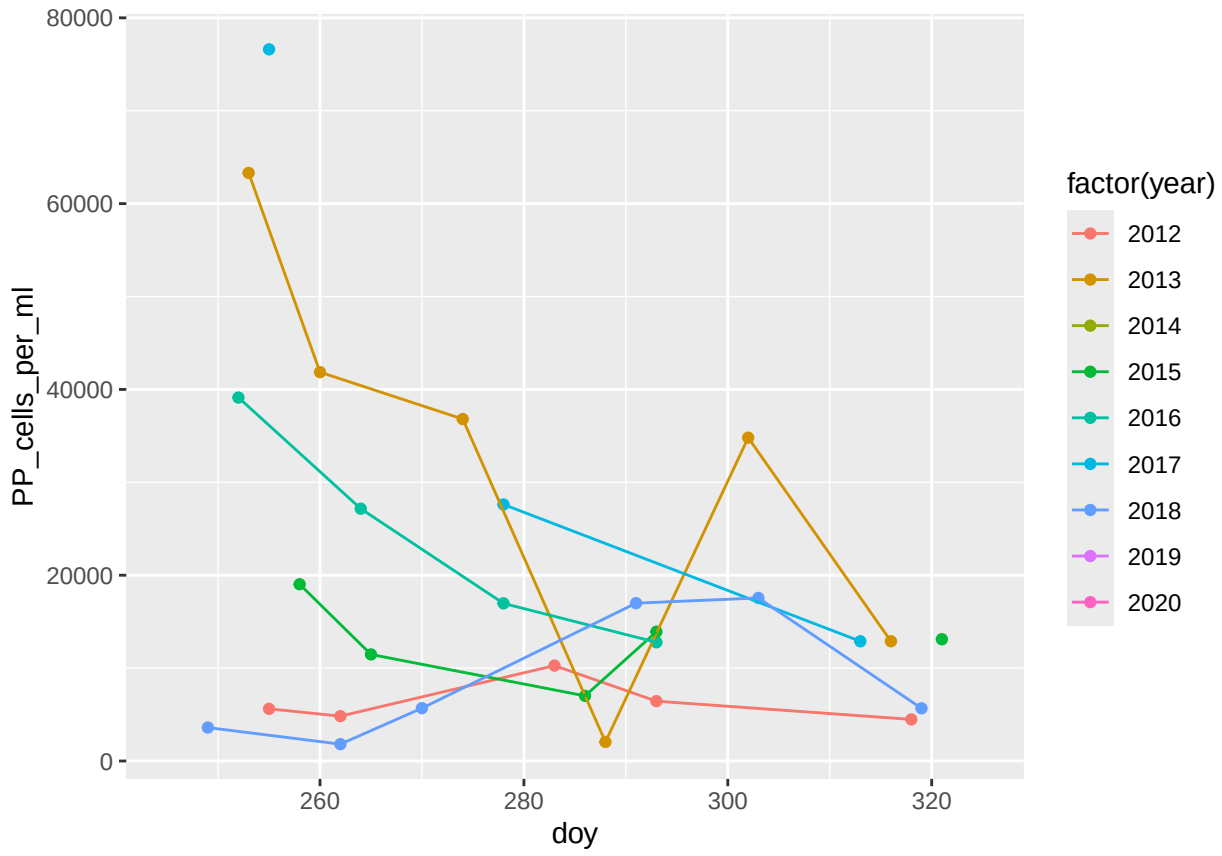


Let's color each point by the year.

```
surfaceData %>% filter(Station == 1) %>%
  ggplot(aes(x=doy, y= PP_cells_per_ml,
             color = factor(year))) +
  geom_point() +
  geom_line() +
  xlim(245,325)
```

```
## Warning: Removed 35 rows containing missing values or values outside the scale range
## (`geom_point()`).
```

```
## Warning: Removed 31 rows containing missing values or values outside the scale range
## (`geom_line()`).
```

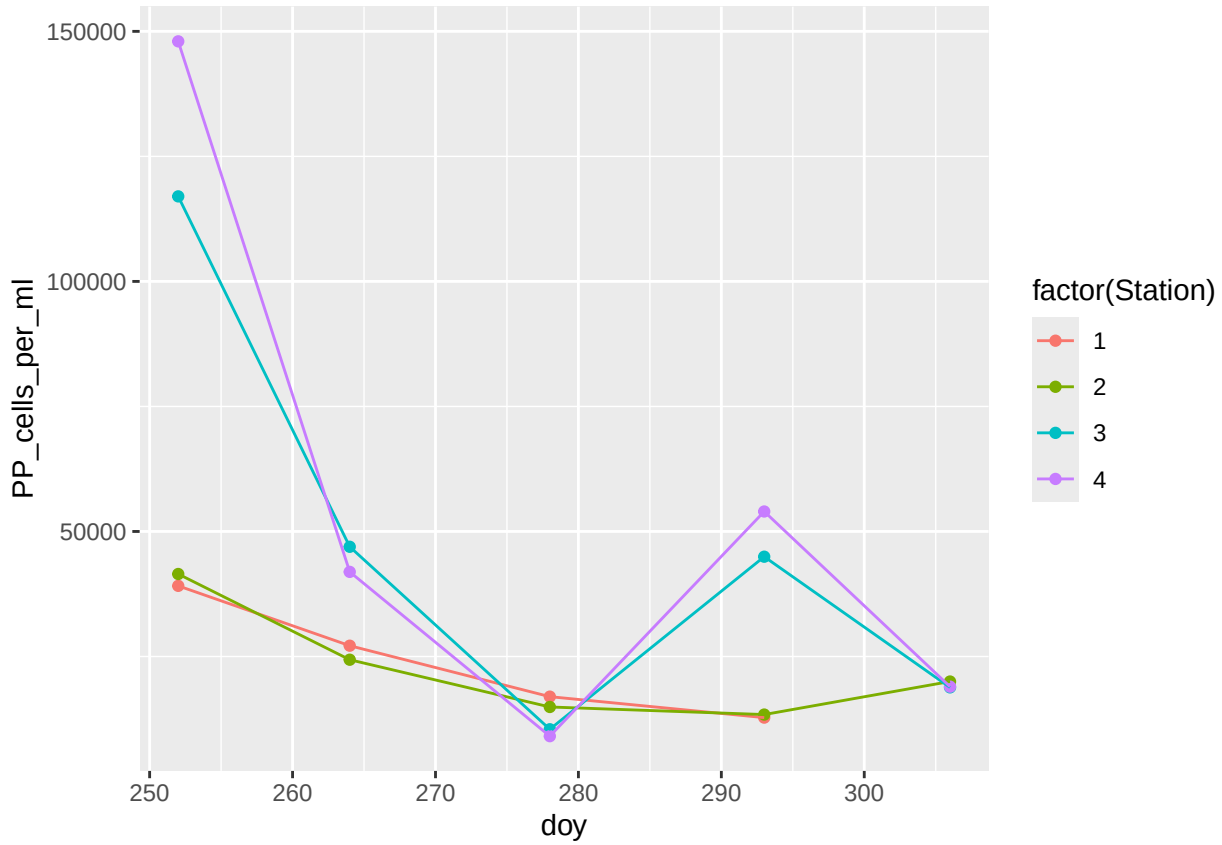



```
surfaceData %>% filter(year == 2016) %>%
  ggplot(aes(x=doy, y=PP_cells_per_ml,
             color = factor(Station))) +
  geom_point() +
  geom_line()
```

4) What is the seasonal change at all stations during 2016?

```
## Warning: Removed 1 row containing missing values or values outside the scale range
## (`geom_point()`).
```

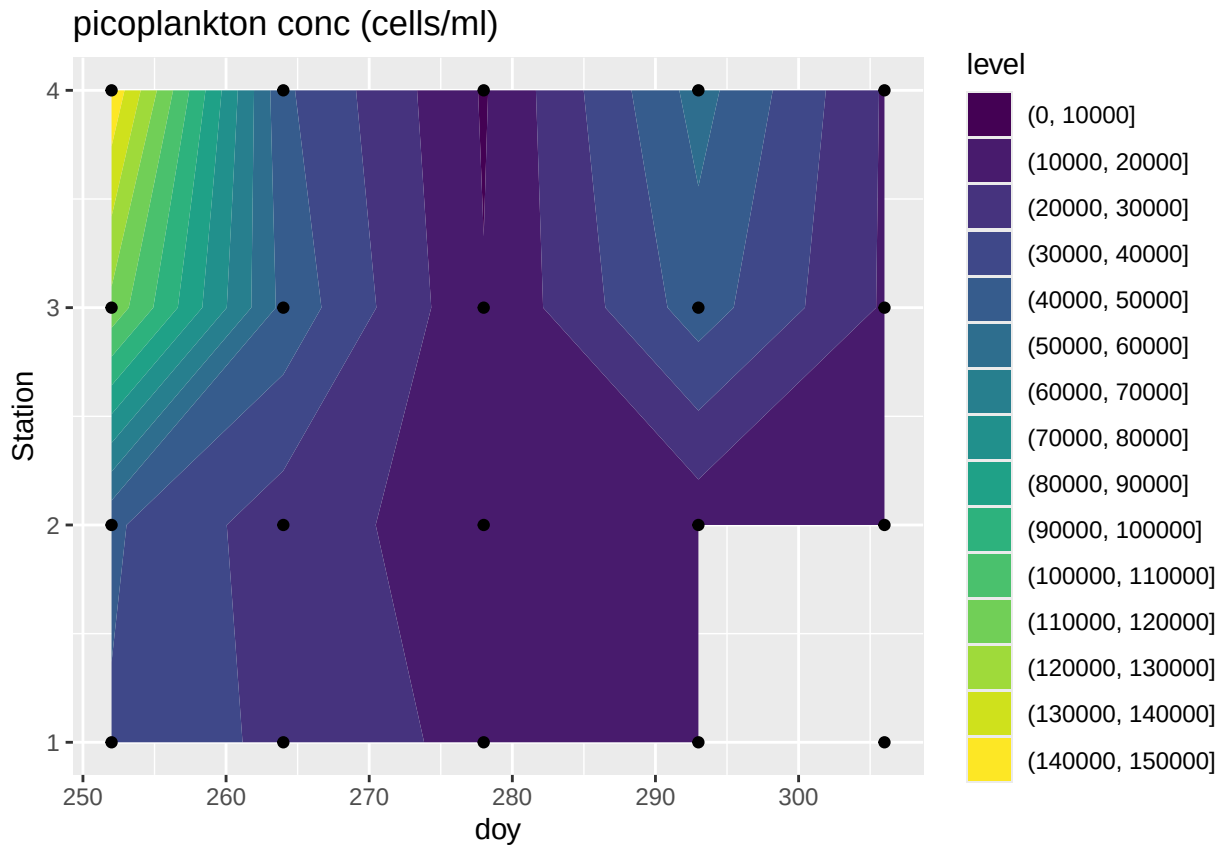
```
## Warning: Removed 1 row containing missing values or values outside the scale range
## (`geom_line()`).
```



Another way to plot this is with a contour plot

```
surfaceData %>% filter(year == 2016) %>%
  ggplot(aes(x=doy, y= Station,
             z = PP_cells_per_ml)) +
  geom_contour_filled() +
  geom_point() +
  labs(title = 'picoplankton conc (cells/ml)')
```

```
## Warning: Removed 1 row containing non-finite outside the scale range
## (‘stat_contour_filled()‘).
```



Assignment: Digging into temperature

Lab 3: Assignment Plots

Using the historical DaRTS data, answer the following questions through construction of plots in R.

1. What is the overall change in temperature year-on-year? (Hint: plot the temperature distribution for each year as a boxplot)
2. What is the mean seasonal temperature cycle within a year?
3. What is the seasonal change at Station 1? Show each year in a different color.
4. What is the seasonal change in temperature at all stations during 2016?