Merging data and making plots of zooplankton distributions from the mDPI

Adam T. Greer

April 17, 2024

What is the mDPI?

The modular Deep-focus Plankton Imager (mDPI) is a towed instrument that measures oceanographic properties (temperature, salinity, dissolved oxygen, and chlorophyll-a) while continuously collecting images of plankton between ~500 microns and ~10 cm in size. The system is towed at an average speed of 1.0-1.5 m/s through the water. The vehicle goes up and down through the water via cable paying in and out from an opto-electric winch, which results in slow vertical movement of the vehicle and generates a "saw-tooth" pattern for its trajectory. The mDPI shadowgraph optical setup captures images that are 2330 px by 1750 px, which corresponds to a field of view of 10.2 cm by 7.7 cm. We typically say that the optical setup has "44 micron pixel resolution" because each pixel is 44 microns in width and height. The images have a depth of field of 25 cm, so each image captures ~2.0 L (1960 cm cubed). To calculate the volume sampled, simply multiply the volume of an inividual image by the total number of images collected in that area or vertical bin of interest. With the counts of individual plankton and the volume sampled, you can calculate concentrations over a variety of distances (from the whole transect to much smaller bins). You have to collect ~526 images to sample one cubic meter of water, which at a rate of approximately 5 images per second, will take approximately 1 minute and 45 seconds. Over long distances, the actual image aquisition rate of the mDPI is 4.924 images per second.

You will be working with the dataset collected between 2020 and 2023 in the South Atantic Bight mid-shelf area. The mid-shelf corresponds to a bottom depth of ~20 m to 50 m. This was part of an extensive field sampling campaign for the NSF project DolMICROBE (OCE 2023133). See the DolMICROBE project page for a summary of the types of oceanographic, chemical, and biological data that were collected.

This document assumes that you have three data types at your fingertips: 1) A data frame with all of the sensor data combined with a julian timestamp for each row. 2) A directory full of labelled zooplankton images (with the label as part of the file name), and 3) access to the original list of files collected with the mDPI. You will learn how to follow 4 steps below to generate vertical distributions of zooplankton that you choose within your directory of labelled images.

1) Create a data frame where each row correspond to one full frame image on the mDPI (and save for later use).

2) Create a date, time, and image number "key" in the full frame sensor data, and the same "key" in a series of identified segments, to merge each image segment to its nearest environmental data.

3) Calculate the volume sampled in each depth bin, and use that volume to generate vertical distributions of zooplankton.

4) Subset the data to make plots of abundances in stacked bars, lines, or plots that are faceted to be taxon-specific.

Once you understand what is happening in these 4 steps, you can modify them for different stations and make all kinds of discoveries about the water column conditions influencing the zooplantkon. This can also help you to perform more in-depth statistical analyses and data visualizations!

Loading packages and customized functions

Load the following packages for completing the 4 basic steps outlined below. You will need packages for plotting (ggplot2), shaping data into correct formats for plotting (reshape2), and splitting data by groups to apply a function (e.g., averaging or counting, plyr and dplyr). For plotting, the "gg" in ggplot2 stands for "grammar of graphics" based on a book with the same title written by Leland Wilkinson in 2005. In this book, as with ggplot2 code, the components of a plot are broken down into 3 parts. First, you have your data, which is essentially a table with columns corresponding to measured values or variables that identify certain sites or organisms. We call this data shape "long format" which is in contrast to "wide format." Long format is necessary because 1 columns will correspond to one aesthetic on the plot, and in ggplot2, it is impossible to refer multiple columns and relate them to one aesthetic (hopefully that will become clear once you start using ggplot2). The data from the table in long format are then "mapped" to certain aesthetics on the plot. This is a fancy way of saying that different parts of the plot represent values in the data table. For example, the x axis is mapped to a column containg temperature values, and the y axis is mapped to salinity. The final component is how the data are expreseed, and this is called a "geom." The simplest form of a geom is a point (geom_point()), which needs an x and a y value for it to show up on a basic x/y plot. The point geom can also have a color aesthetic that can be mapped to another data type. You could also add an aethetic for shapes if you had, for example, 5 sites that you want to compare on an x/y plot. Other geoms do not require as many mappings. A histogram, for example, only requires one variable because geom_histogram() is binning those data and tallying up how many there are in each bin. It would not make sense to give geom_histogram() an x and y value like you would with geom_point(). And it will give you an error if you supply geom_histogram() with x and y aesthetics!

So that is what ggplot2 needs: a data frame to reference, aesthetic mappings of variables to parts of the plot (x, y, colour, shape, etc.), and a geom for how the data should be expressed. That's it! But ggplot2 is incredibly versatile and can be used to make almost any kind of plot you can think of (even plot animations with the "gganimate" package, which is an add on to ggplot2). You can also use different data frames and overlay them as "layers" with different geoms, so these plots can get really complex! Towards the end of this document, we will look at one of the most powerful aspects of ggplot2 - faceting. This allows you to quickly make subplots for different unique values of a column in your data, which is really useful for comparing among sites or species.

```
library(ggplot2) #for plotting
library(reshape2) #manipulating data
library(plyr) #split data, apply a function to chunks, then combine
#library(dplyr)
#dplyr is basically the same as plyr, but it uses pipes %>% and the code
#looks really different. The same logic applies though: use unique
#values of a grouping variable, apply function(s), then combine into a
#data frame. dplyr also runs a bunch of C code in the background,
#so it is faster than the same operations in plyr!
```

You will also need to load a few customized functions. The function nearphys() will find values that are closest to one another between 2 arrays supplied to it. This function is useful for merging different data frames that do not have exactly matching values (but are very close, which often happens with timestamps for sensors not measuring at exactly the same rate). The function julianmakerPD, which was written by Patrick Duffy (PD), is a versatile function for making julian times (percentage of a day) from different strings of hr, minutes, seconds and image number. It incorporates the constant rate of image aquisition with the mDPI (4.924 images per second). The last function, qvertplot(), is short for "quick vertical plot" and will count up organisms and volume sampled in a specified depth bin size. The resulting data frame from qvertplot() is easy to feed into ggplot2 and see the vertical distribution of different organisms at your chosen vertical resolution, specified by the "bwidth" argument. If qvertplot() is failing, make sure that the "depth" variable name in the function matches your data. Depending on when it was collected, it might be called "pressure", "Pressure", or "Depth". And yes, capitalization matters in R!

```
nearphys <- function(x,y) {
  n <- length(x) #x will be the longer data frame
  phystime <- vector(length = n)
  for(i in 1:n) {
    phystime[i] <- which.min(abs(x[i]-y))
  }
    return(phystime)
}

julianmakerPD <- function (t,s) {
  if(missing(s)){
    #Handles time without any image number (slices) in the format hh:mm:ss.sss
    #This input without slices must be a string
    jul <- as.character(t)
    hr <- as.numeric(substr(jul, 1,2))
    minu <- as.numeric(substr(jul, 4,5))
    sec <- as.numeric(substr(jul, 7,12))
    julian <- (hr/24)+(minu/(24*60))+(sec/(24*60*60))
  } else {
    #Handles the timestamps from the ROI which are formatted without decimals
    #Uses the image number (slice) to add seconds onto julian calculation
    jul <- as.character(t)
    hr <- as.numeric(substr(jul, 1,2))
    minu <- as.numeric(substr(jul, 3,4))
    sec <- as.numeric(paste0(substr(jul, 5,6), ".", substr(jul,7,9)))
    #4.924 images per second, so this must be added to total seconds
    julian <- (hr/24)+(minu/(24*60))+((sec+(s/4.924))/(24*60*60))
  }
  return(julian)
}

qvertplot <- function(ph, pe, bwidth) {
  bins <- seq(0, round_any(max(ph$Pressure)+(2*bwidth), bwidth, ceiling), bwidth)
  numobs <- hist(ph$Pressure, breaks = bins, plot = FALSE)$counts
  volsamp <- numobs * .102 * .077 *.25 #field of view and depth of field in m
  planknum <- ddply(pe, .(ID),
                    function(x){a <- hist(x$Pressure, breaks = bins, plot = F)$counts
                    return(a)})
  planknum <- melt(planknum)[,-2]
  numtaxa <- length(unique(planknum$ID))
  plak <- cbind(planknum, rep(bins[-1], each = numtaxa))
  plak2 <- cbind(plak, rep(volsamp[seq(1,(max(bins)/bwidth),1)], each = numtaxa))
  colnames(plak2) <- c("ID", "count", "depth", "volume.sampled")
  plak2$conc <- as.numeric(plak2$count)/as.numeric(plak2$volume.sampled)
  return(plak2)
}
```

What you need for this tutorial and overall goals

To complete this tutorial you need 3 things: 1) A full processed physical data file (.csv). This is a massive table with sensor data at its highest resolution (~8-9 measurements per second) 2) A list of file names of the full frame images collected with the mDPI. This is just used as a reference and can be read off of the server. 3) A directory full of image segments (preferably labeled with an ID, but that is not a requirement). These

can be read off the server, but it is more fun to download them to your machine so you can quickly see all the cool zooplankton that you are using to make discoveries about their ecology!

The goal of this process is to make a data frame where each full frame image corresponds to one row of sensor data. This makes it really easy to calculate volumes sampled, and plotting vertical distributions is straightforward. Our overall goal is to eventually put these time-key data frames and labelled images into a database, which will allow us to easily view the data in interactive ways. This process is connecting our plankton images to the sensor data, so we can ask questions about what is driving the distributions of different animals. The first step in this process is just looking at the vertical distributions of zooplankton.

Sensor data from the mDPI

We typically refer to any non-image related data on the mDPI as "sensor data" or "physical data". These sensors include the CTD (salinity, temperature, and depth), dissolved oxygen (converted to mg/L), and chlorophyll-a fluorescence (measured as relative "counts"). We can calculate the distance traveled using the average speed of the vehicle (1.0-1.5 meters per second) and the amount of seconds the vehicle traveled since the start of the deployment. We will not be adding a distance in this tutorial, but the tutorial on the mDPI sensor data can show you how that is done.

First, set your working directory to the folder with the .csv files correpsonding to the sensor data using setwd(). The function getwd() will tell you where the current working directory is located.

```
setwd("d:/") #will be different for your machine
getwd() #use this to check that the working directory was changed
```

```
## [1] "d:/"
```

Once the working directory is set up to match where your sensor data files are located, you can then read in sensor data. We will read in the merged sensor data from a station at the 45m isobath on August 11, 2021. These data include a julian timestamp for each row and all of the sensors on the mDPI. It should include all sensor data recorded at that location, including the "bad" data. Don't worry, these bad values (i.e., when the mDPI may not have been in the water) will be removed when we merge our sensor data to the full frame images!

```
#Read in data frame with all sensors together
phys <- read.csv("Dol_16_SAV21_16_station4_45m_day.csv", header = TRUE)
#salinity needs to be converted to numeric
phys$Salinity <- as.numeric(as.character(phys$Salinity))
head(phys)
```

```
##           Time Temp.CTD Conductivity Pressure Salinity Velocity    julian
## 1 14:57:42.491  36.6151      0.00029   -0.037   0.0154 1522.937 0.6234085
## 2 14:57:42.519  36.6169      0.00030   -0.034   0.0154 1522.941 0.6234088
## 3 14:57:42.581  36.6186      0.00030   -0.035   0.0154 1522.944 0.6234095
## 4 14:57:42.644  36.6201      0.00031   -0.031   0.0154 1522.947 0.6234102
## 5 14:57:42.706  36.6215      0.00031   -0.034   0.0154 1522.949 0.6234109
## 6 14:57:42.769  36.6227      0.00031   -0.034   0.0154 1522.952 0.6234117
##   ChlaCounts     Date  AirSat Temp.Opt   O2mgL      Station TOD Cruise
## 1         53 08/11/21 101.069   37.421 6.74832 station4_45m day Dol_16
## 2         53 08/11/21 101.069   37.421 6.74832 station4_45m day Dol_16
## 3         53 08/11/21 101.069   37.421 6.74832 station4_45m day Dol_16
## 4         53 08/11/21 101.069   37.421 6.74832 station4_45m day Dol_16
## 5         53 08/11/21 101.069   37.421 6.74832 station4_45m day Dol_16
## 6         53 08/11/21 101.069   37.421 6.74832 station4_45m day Dol_16
```

Ok, so we have the sensor data loaded, but we do not need to do anything with them yet. For the next step, we need to load a list of files from the full frame images collected at the station. These file names contain information about the time they were collected, which will help us isolate the sensor data that correspond to when we have images from the mDPI. In these next steps, our goal is to make a julian time for each full frame image we collected with the mDPI by isolating the relevant information from the file name. Then we merge this with the nearest sensor data. We also create a "time-key" which is a unique identifier for every image taken with the mDPI. The format of the time-key is YYYYMMDD_HHMMSSSSS_Frame. So for example, the time-key for the 100th frame taken on June 12, 2021 at 13:32:22.333 would be 20210612_133222333_100. The HHMMSSSSS portion of the time key is the beginning of the series of images or frames, so if we lose contact with the vehicle, there is a new start time, and another series of images is collected. All of this should be accounted for in the calculation of julian times in julianmakerPD().

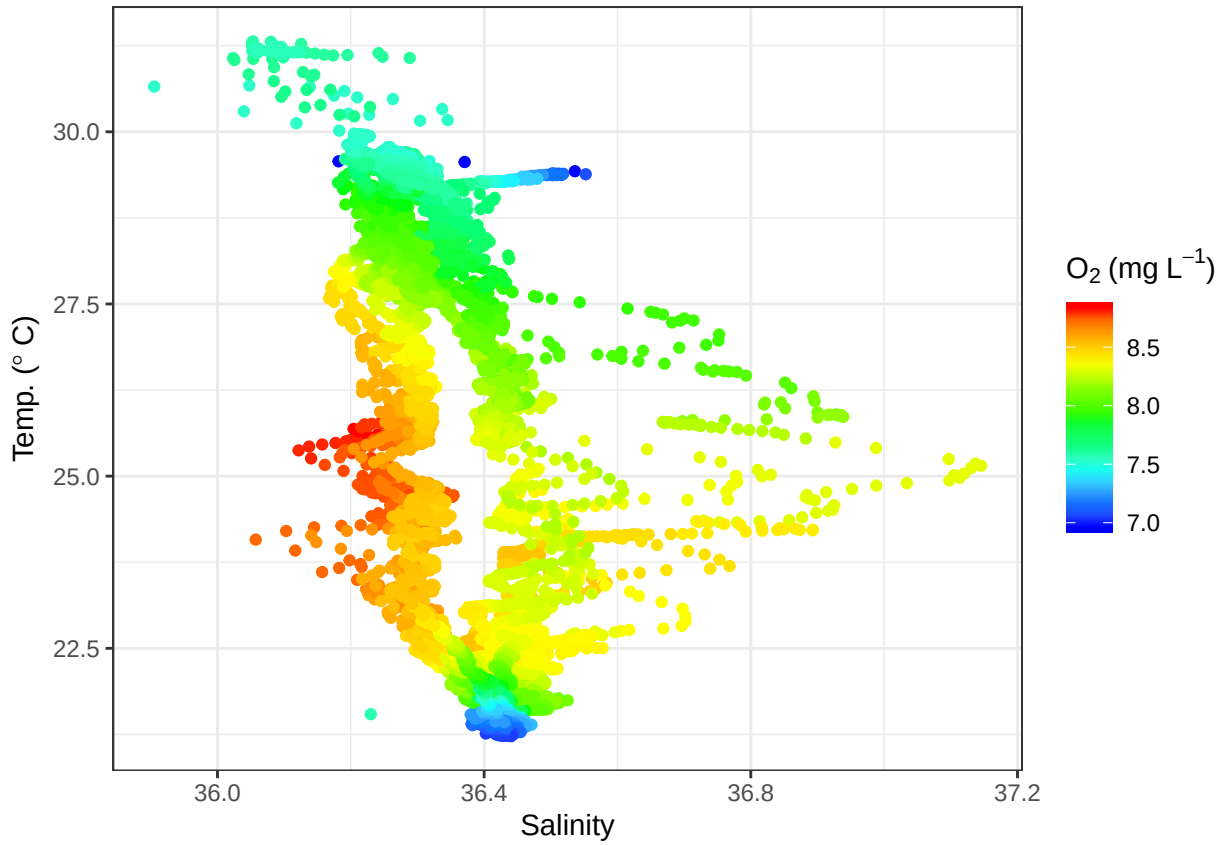
```
#Read in the list of full frame file names
frame_list <-  list.files("Z:/2020_DolMicrobe/Cruise_Data/SAV 21-16/Station4-45m/")

#Get rid any weird files or folders in there with the image files
#And get rid of .tiff in the frame_list
frame_list <- frame_list[!frame_list %in% c("data","Thumbs.db")]
frame_list <- gsub(".tiff", "", frame_list)

#split by the underscore and create a matrix
#with the values read in rows for 8 columns
imgf <- unlist(strsplit(frame_list, "[_]"))
imgf2 <- matrix(imgf, ncol = 8, byrow = TRUE)
imgf2 <- as.data.frame(imgf2, stringsAsFactors = FALSE)

#isolate the time component and the frame number
ftime <- data.frame(imgf2$V6, imgf2$V7, imgf2$V8)
colnames(ftime) <- c("date","timestamp", "frame")

#Make sure timestamp is a character to avoid dropping a leading 0
ftime$timestamp <- as.character(ftime$timestamp)
ftime$frame <- as.numeric(as.character(ftime$frame))

#make the julian time using timestamp and frame #
jul <- julianmakerPD(ftime$timestamp, ftime$frame)

#make the time key: YYYYMMDD_HHMMSSSSS_Frame
#and attach the corresponding julian time
tkey <- paste0(ftime$date,"_", ftime$timestamp, "_", ftime$frame)
nf <- data.frame(jul, tkey)

#merge each tkey to the nearest data point from phys
ind1 <- nearphys(nf$jul, phys$julian)
julph <- phys$julian[ind1]
nf2 <- cbind(julph, nf)
colnames(nf2)[1] <- "julian"

tkeyphys <- merge(nf2, phys, by = "julian", all.x = TRUE)

#get rid of the 2 julian columns and save
tkeyphys <- tkeyphys[,c(-1, -2)]
head(tkeyphys)
```

```
##                   tkey         Time Temp.CTD Conductivity Pressure Salinity
## 1 20210811_145924375_1 14:59:24.583  29.5709      5.96285    1.861  36.1817
## 2 20210811_145924375_2 14:59:24.770  29.5979      5.97824    1.898  36.2669
## 3 20210811_145924375_3 14:59:24.959  29.5610      5.98934    1.929  36.3708
## 4 20210811_145924375_4 14:59:25.209  29.4275      5.99865    1.977  36.5361
## 5 20210811_145924375_5 14:59:25.396  29.3835      5.99614    2.012  36.5524
## 6 20210811_145924375_6 14:59:25.583  29.3896      5.99191    2.050  36.5185
##   Velocity ChlaCounts     Date AirSat Temp.Opt    O2mgL      Station TOD Cruise
## 1 1545.980         58 08/11/21 98.111   33.633 6.965728 station4_45m day Dol_16
## 2 1546.126         58 08/11/21 98.111   33.633 6.965728 station4_45m day Dol_16
## 3 1546.158         58 08/11/21 98.111   33.633 6.965728 station4_45m day Dol_16
## 4 1546.051         58 08/11/21 98.111   33.633 6.965728 station4_45m day Dol_16
## 5 1545.976         58 08/11/21 98.854   33.072 7.084128 station4_45m day Dol_16
## 6 1545.954         59 08/11/21 98.854   33.072 7.084128 station4_45m day Dol_16
```

```
#Save this file so you don't have to deal with the raw sensor data anymore!
write.csv(tkeyphys, file = "Dol_10_SAV21_16_25m_day-frames.csv",
          row.names = FALSE)
```

Let's see how our image-relevant sensor data looks by making a Temperature-Salinity plot. The color of the dot will be the measured dissolved oxygen (mg/L).

```
ggplot(tkeyphys, aes(Salinity, Temp.CTD, colour = O2mgL))+geom_point()+theme_bw()+
  scale_colour_gradientn(colours=c("blue", "#007dff", "cyan", "#00ff7d", "green",
                                   "#7dff00", "yellow", "orange", "red"),
                         na.value="white")+
  labs(y = expression(paste("Temp. (", degree~C,")")), colour =
         expression(paste(O[2]~'(mg'~L^{-1},')')))
```



Now, let's look at the list of image segments with labels (which serve as the ID). We will want to isolate the part of the image file name that has the date, timestamp, and frame number. We will combine these into one string to make a "key" and use it to combine with our newly createed image-based sensor data frame (tkeyphys) which has its own key (tkey) attached to it. We also want to isolate the ID of our organisms. For the merging, we will want to keep all of our image files, resulting in a data frame where each row is a labelled organism with corresponding environmental data.

```
imglab <- list.files("d:/Station4_classified/")
#you can also locate this on the server
#files are in Z:/KyleAaronData/Adam-ROIsChecked/
#each folder is a different station

#check the exact pattern of .tiff in your file set
#need to remove it from the middle of files
#to properly calculate julian time
imglab <- gsub(".tiff.tif.tif", "", imglab)

#Get rid of Thumbs.db if it exists
imglab <- imglab[! imglab %in% c("data", "Thumbs.db")]
imgf2 <- unlist(strsplit(imglab, "[_]"))
imgf2 <- matrix(imgf2, ncol = 14, byrow = TRUE)
tkey <- paste0(imgf2[,8], "_", imgf2[,9], "_", imgf2[,10])

#combine the segment ID, timekey, and the full file name
biodat <- data.frame(imgf2[,1], tkey, imglab)
colnames(biodat) <- c("ID", "timekey", "filename")
```

```
colnames(tkeyphys)[1] <- "timekey"

#merge the segments to the time-key phys using the time-key
#YYYYMMDD_HHMMSSSSS_FRAME
#keep all of the biodat (image segments)
plankenv <- merge(biodat, tkeyphys, by = "timekey", all.x = TRUE)
levels(as.factor(plankenv$ID))
```

```
##  [1] "amphi"     "anemo"     "app"       "chaeto"    "copepod"   "cteno"
##  [7] "diatom"    "dolio"     "dolnurse"  "dolphor"   "dolphoro"  "fish"
## [13] "hetero"    "hydro"     "isopod"    "other"     "poly"      "ptero"
## [19] "rhiz"      "salp"      "salpchain" "scypho"    "shrimp"    "siphon"
## [25] "snow"      "stoma"     "tricho"    "zooea"
```

It looks like there is a typo where "dolphoro" should just be "dolphor." That is pretty easy to change in R (for plotting), but this will not alter the original file name. Generally, we want to leave files alone in all of our analyses. The R code provides a record of what we are doing to process (or correct) the data - there are always errors that need fixing!

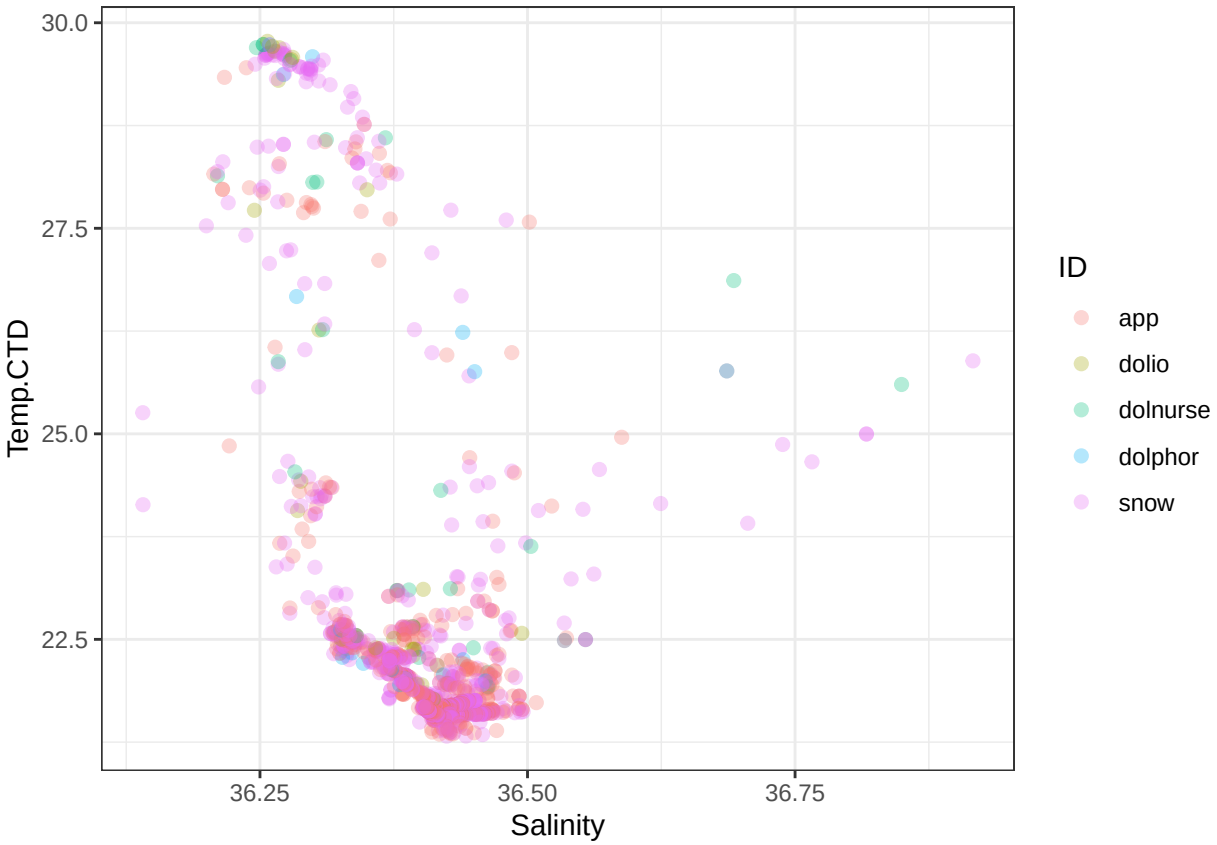
```
plankenv$ID <- ifelse(plankenv$ID == "dolphoro", paste("dolphor"), paste(plankenv$ID))
plankenv$ID <- as.factor(plankenv$ID)
levels(plankenv$ID) #check and see that dolphoro is gone
```

```
##  [1] "amphi"     "anemo"     "app"       "chaeto"    "copepod"   "cteno"
##  [7] "diatom"    "dolio"     "dolnurse"  "dolphor"   "fish"      "hetero"
## [13] "hydro"     "isopod"    "other"     "poly"      "ptero"     "rhiz"
## [19] "salp"      "salpchain" "scypho"    "shrimp"    "siphon"    "snow"
## [25] "stoma"     "tricho"    "zooea"
```

Next comes the fun part - making vertical distribution plots of different zooplankton. Whatever you choose to plot, you are the first person in the history of the Earth to see the distribution of these animals with high spatial resolution (~1 m) in the South Atlantic Bight. This area has been completely unexplored with plankton imaging systems, so detailed spatial data on all of these organisms does not exist until now!

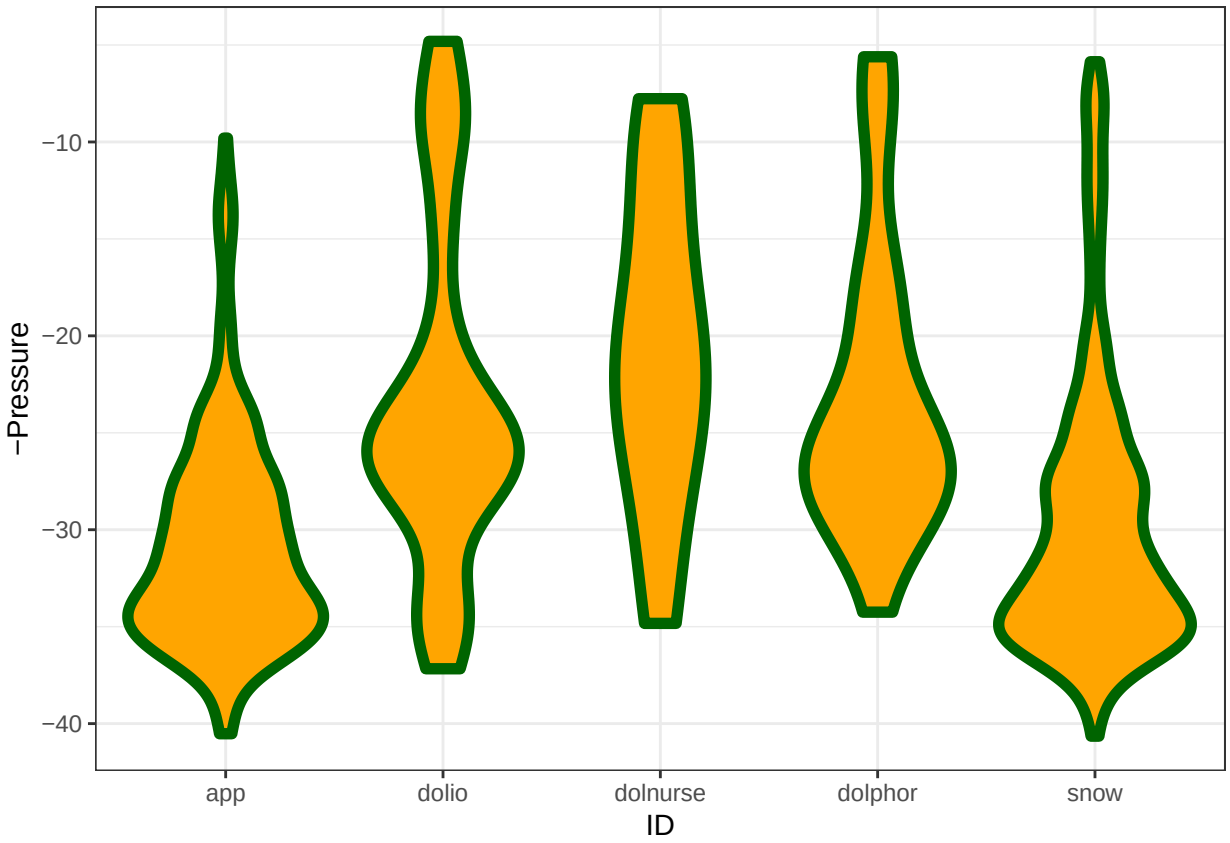
We can do lots of fun and interesting things just with the plankenv data frame. We can make a T/S plot of the different organisms and view them as different colors (with some degree of transparency in the point specified by alpha, which is 1 or completely opaque by default). To make the visualization look nice, I will subset the data frame to only look at 5 groups. Otherwise, every level of ID will be plotted, and we won't be able to see anything useful

```
pesmall <- subset(plankenv, ID %in% c("dolphor", "dolnurse", "dolio", "snow", "app"))
ggplot(pesmall, aes(Salinity, Temp.CTD, color = ID))+
  geom_point(alpha = 0.3, size = 2)+theme_bw()
```



We can use geom_violin() to get a quick glance at the vertical distribution of these 5 groups. This geom is just fitting a function for how dense the points are along the y axis (which is supplied with aes() in the second argument). Let's make the violins orange and give them a fat green outline (just for fun). Because these characteristics are specific outside of aes(), but inside a geom, they will simply alter the way that geom looks in a way that is completely unrelated to any data values.

```
ggplot(pesmall, aes(ID, -Pressure))+
  geom_violin(fill = "orange", colour = "darkgreen", size = 2)+
  theme_bw()
```



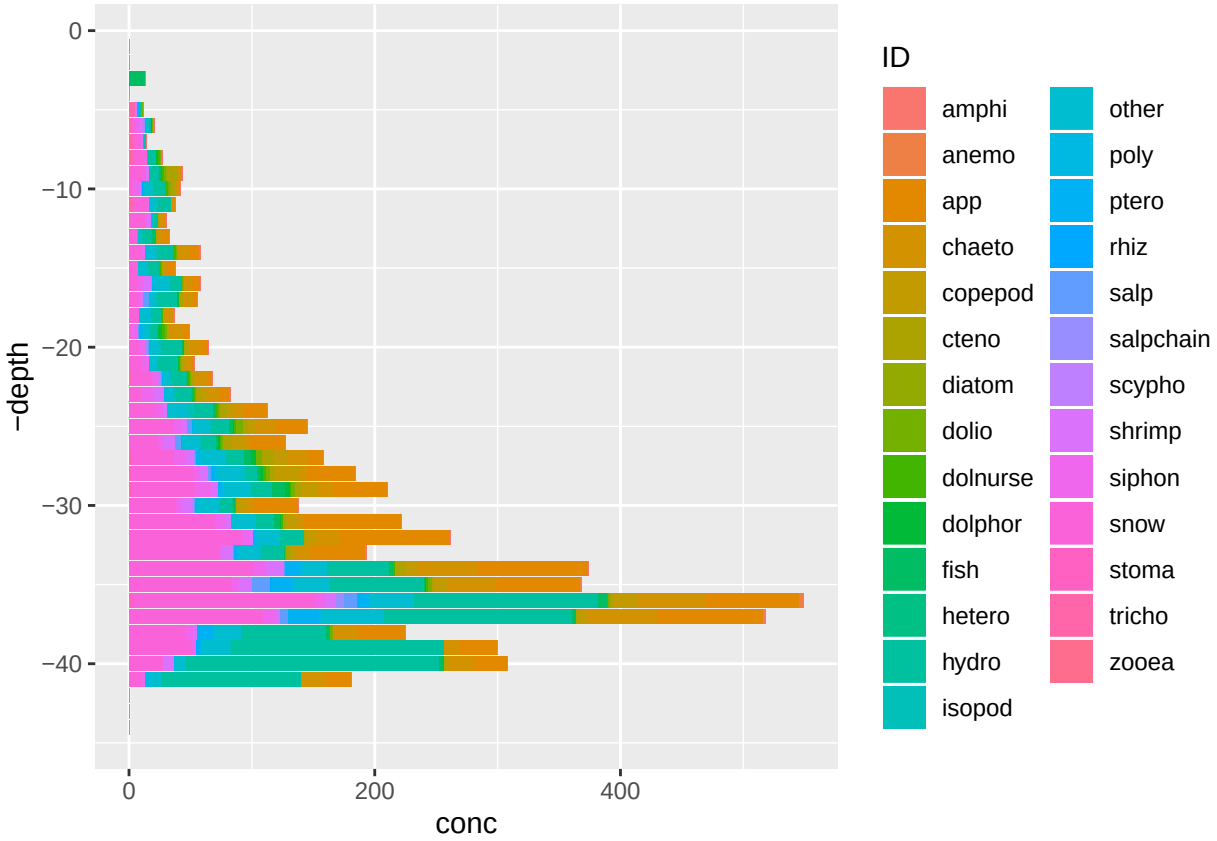
Wow! Appendicularians and marine snow are almost identical in their vertical distribution. But if you look really closely, there is a slight vertical offset where the appendicularians are a little shallower by about 1 m. Now that is the power of high resolution observations! This plot is interesting and all, but we have to remember that these violins are not correcting for the fact that we do not spend the same amount of time at each depth zone. Therefore, we need to use the phsm data frame to calculate volumes sampled and get more precise vertical distributions. The function qvertplot() will be really important here.

```
alltaxa <- qvertplot(tkeyphys, plankenv, 1) #1 m vertical depth bins
```

```
## Using ID as id variables
```

```
#Plot the vertical distribution of all identified segments
ggplot(alltaxa, aes(-depth, conc, fill = ID))+geom_bar(stat = "identity")+coord_flip()
```

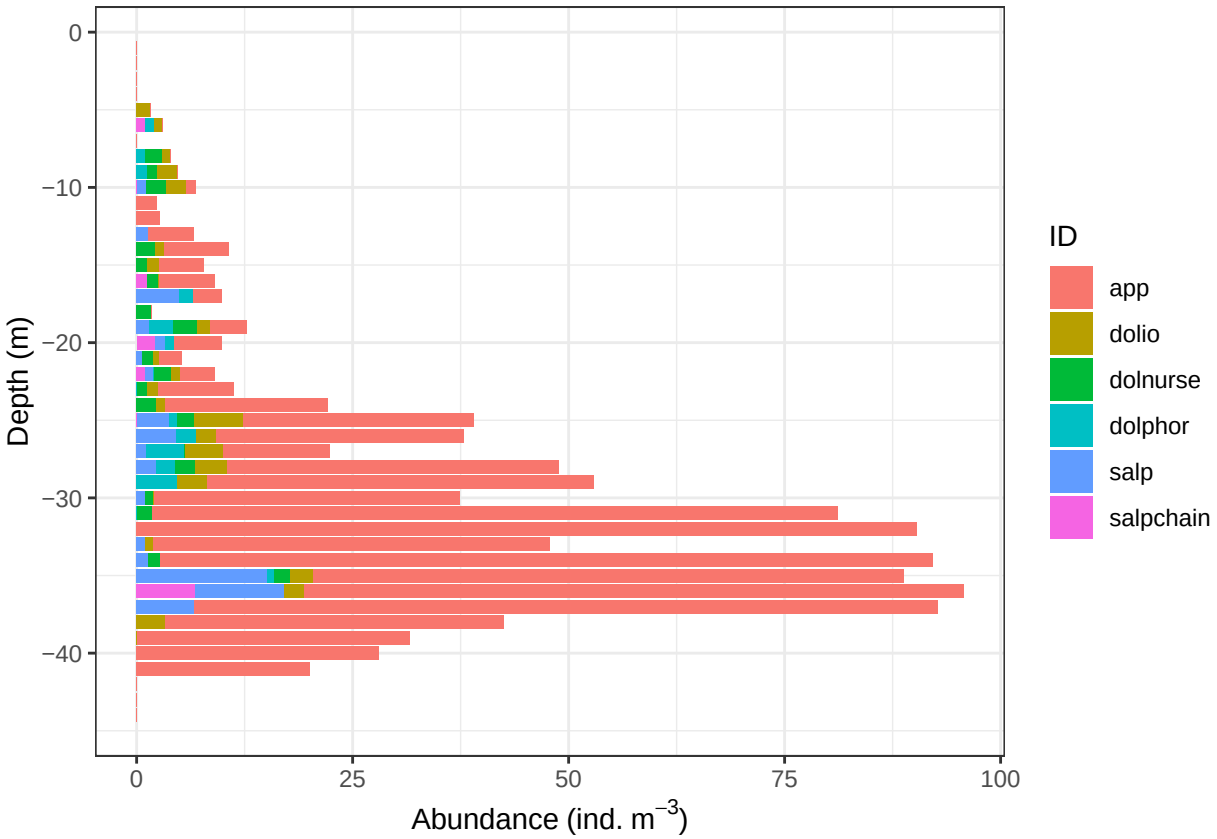
```
## Warning: Removed 54 rows containing missing values (position_stack).
```



That's way too many color levels! Let's simplify this a bit and only look at the thaliaceans (salps, doliolids, and appendicularians). We will do this by creating a new data frame that is a subset of the original data frame and isolates these 6 groups of interest. We can make the same plot but with better axis labels, so it looks more professional.

```
thals <- subset(alltaxa, ID %in% c("dolio", "dolnurse", "dolphor", "app", "salp", "salpchain"))
ggplot(thals, aes(-depth, conc, fill = ID))+geom_bar(stat = "identity")+coord_flip()+
  labs(x = "Depth (m)", y = expression(paste('Abundance'~'(ind.'~m^{-3},')')))+theme_bw()
```

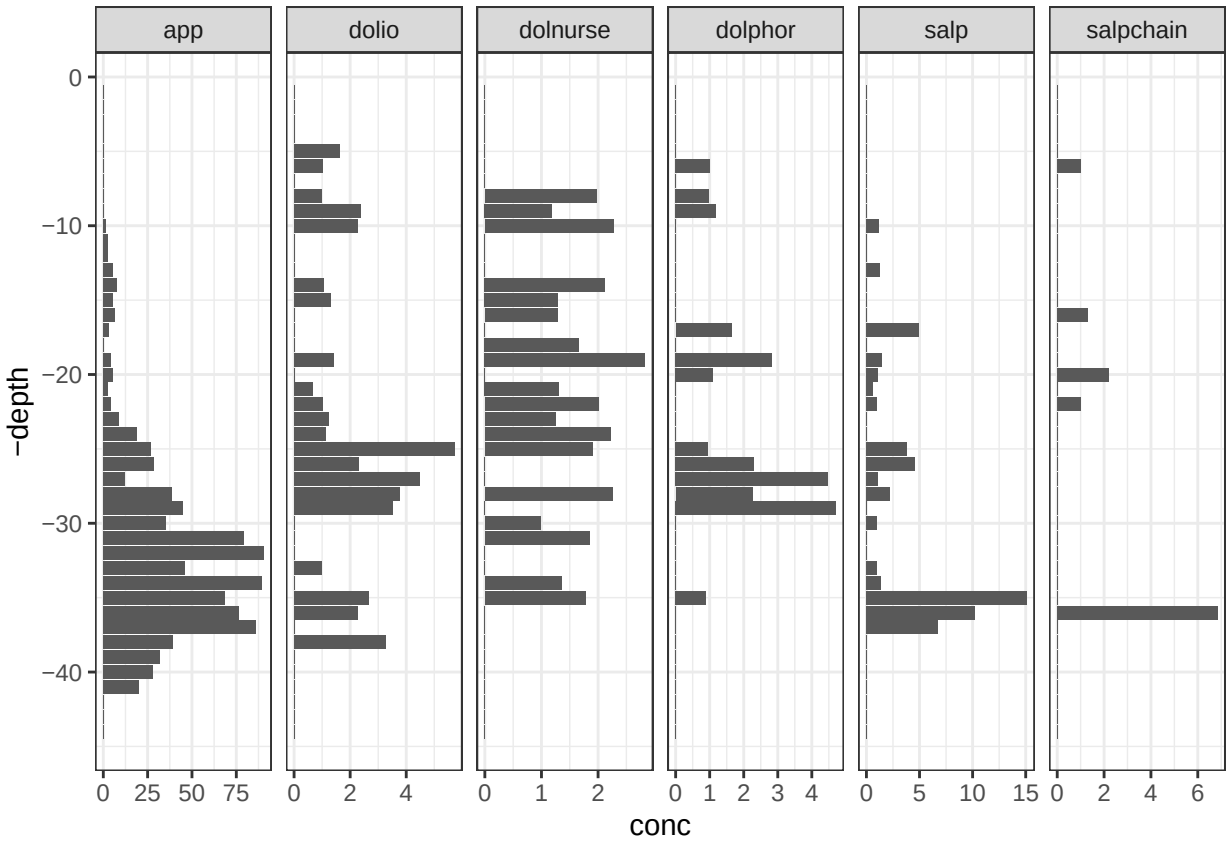
```
## Warning: Removed 12 rows containing missing values (position_stack).
```



It's still difficult to compare the distributions because the abundances are stacked on top of one another. How about we make a little subplot for each taxa and let the abundance scale (x axis) change? We will also keep them in one row across, so the y axis (depth) is lined up. This will let us easily compare the distributions amongst the different zooplankton.

```
ggplot(thals, aes(-depth, conc))+geom_bar(stat = "identity")+
  facet_wrap(~ID, scales = "free_x", nrow = 1)+coord_flip()+theme_bw()
```

```
## Warning: Removed 12 rows containing missing values (position_stack).
```



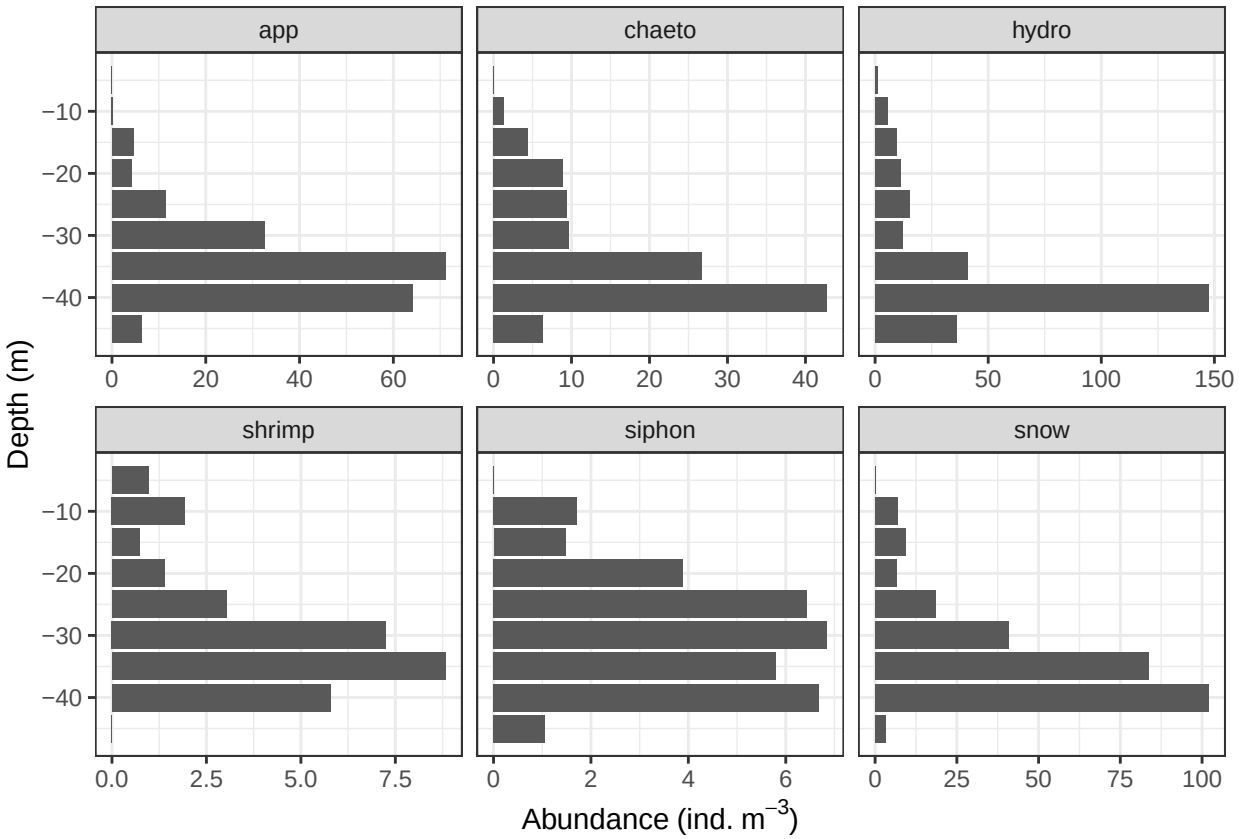
We can also just repeat the use of qvertplot() to make larger bins (5 m in this case) and plot some different groups of zooplankton. What other kinds of plots can you think of? This is just for one station, but with combining data frames, we can also compare distributions of taxa between time points (weeks, months, or years).

```
alltaxa <- qvertplot(tkeyphys, plankenv, 5)
```

```
## Using ID as id variables
```

```
newtax <- subset(alltaxa, ID %in% c("hydro", "snow", "app", "chaeto", "siphon", "shrimp"))

ggplot(newtax, aes(-depth, conc))+geom_bar(stat = "identity")+
  facet_wrap(~ID, scales = "free_x")+
  coord_flip()+theme_bw()+
  labs(x = "Depth (m)", y = expression(paste('Abundance'~'(ind.'~m^{-3},')')))
```

```
## Warning: Removed 12 rows containing missing values (position_stack).
```




```
#save your plot!
ggsave("CommmonTaxa08112021.png", dpi = 300, height = 4, width = 5)
```

```
## Warning: Removed 12 rows containing missing values (position_stack).
```