# Graphics Options for Ambrose et al. Plankton Comunity Composition Study

Curtis C. Bohlen, Casco Bay Estuary Partnership

3/24/2021

## Contents

# Introduction

Erin Ambrose has been looking at species composition of plankton in Penobscot Bay, Maine, working with Rachel Lasley Rasher, at University of Southern Maine.

Lasley Rasher is on the faculty at the University of Southern Maine, which also houses the Casco Bay Estuary Partnership. She has been kind enough to allow us to work in her lab when we worked on some coastal nutrient monitoring projects.

So, at Lasley Rasher's request, I helped Ambrose early in her graduate work with community analysis using nonmetric multidimensional scaling and cluster analysis, both supported by th excellent `vegan` package.

> Jari Oksanen, F. Guillaume Blanchet, Michael Friendly, Roeland Kindt, Pierre Legendre, Dan McGlinn, Peter R. Minchin, R. B. O'Hara, Gavin L. Simpson, Peter Solymos, M. Henry H. Stevens, Eduard Szoecs and Helene Wagner (2020). vegan: Community Ecology Package. R package version 2.5-7. https://CRAN.R-project.org/package=vegan

Recently, as Ambrose and Lasley Rasher were preparing a manuscript for publication, they ran into problems making BAse R and `ggplot2` graphics look similar, and they reached out to me again for some help.

To ensure my analysis and graphics are compatible with theirs I started with some of their existing analysis code (some of which I had helped develop more than a year ago), and revised and simplified it. IN what follows, I include some commentary on coding alternatives.

## The charge

In e-mail, Ambrose and Lasley Rasher asked me the following:

> These [two graphics] are actually the same data but these plots were made using two different codes in R to highlight the groupings (fig 4) and the drivers of those groupings (fig 5). Ideally, we would like to
> 1. Make the scales match
> 2. Keep the polygons in place in fig 5
> 3. Not sure how much we should worry about the color scheme matching?

## Initial Analysis of the Challenge

The two figures from their manuscript, Fig 4 and Fig 5, were NMDS plots, with one highlighting clusters of similar species composition (by drawing polygons around each cluster) and the other highlighting environmental variables (by showing `envfit` vectors).

The problem was, the two graphics relied on different plotting tools, and so did not play well together. Figure 4 was produced in "Base R" graphics, using plot functions included in the `vegan` package. In particular, it relied on the `ordihull()` function.

Figure 5 was constructed using the `ggplot2` graphic system, and included settings that force a 1 to 1 aspect ratio (as is appropriate for an NMDS plot).

### Proposed Solution

Most plotting functions invisibly return a data frame (or other R object) containing underlying plot coordinates. An initial approach, therefore, was to isolate the data from the `ordihull()` call from Ambrose's code, and figure out how to use it in `ggplot2` graphics.

The help page for the `ordihull()` function says the following:

> Function ordihull and ordiellipse return invisibly an object that has a summary method that returns the coordinates of centroids and areas of the hulls or ellipses.

That suggests a path forward.

## Load Libraries

```
library(tidyverse)
#> Warning: package 'tidyverse' was built under R version 4.0.5
#> -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
#> v ggplot2 3.3.5     v purrr   0.3.4
#> v tibble  3.1.6     v dplyr   1.0.7
#> v tidyr   1.1.4     v stringr 1.4.0
#> v readr   2.1.0     v forcats 0.5.1
#> Warning: package 'ggplot2' was built under R version 4.0.5
#> Warning: package 'tidyr' was built under R version 4.0.5
#> Warning: package 'dplyr' was built under R version 4.0.5
#> Warning: package 'forcats' was built under R version 4.0.5
#> -- Conflicts ------------------------------------------ tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(vegan)
#> Warning: package 'vegan' was built under R version 4.0.4
#> Loading required package: permute
#> Warning: package 'permute' was built under R version 4.0.4
#> Loading required package: lattice
#> This is vegan 2.5-7
library(readxl)
library(reshape2) #need to turn data from long to wide
#> Warning: package 'reshape2' was built under R version 4.0.4
#>
#> Attaching package: 'reshape2'
#> The following object is masked from 'package:tidyr':
#>
#>     smiths
```

## Set Graphics Theme

This sets `ggplot()`graphics for no background, no grid lines, etc. in a clean format suitable for (some) publications. You can get a lot fancier here with setting graphic defaults, but this is a good starting point.

```
theme_set(theme_classic())
```

## Folder References

I use folder references to allow limited indirection, thus making code from GitHub repositories more likely to run "out of the box".

```
data_folder <- "Original_Data"
```

## Input Data

### Environmental Data

This code generates a fair number of warnings about date conversions, but these are for the time variable, which is inconsistently coded in the source Excel file. We never use the time variable in this code, so we can ignore the warnings. We suppress the warnings here, but that's bad practice until you have worked out all problems loading data. Warnings often indicate a more serious problem that needs addressing.

```
filename.in <- "penob.station.data EA 3.12.20.xlsx"
file_path <- file.path(data_folder, filename.in)
station_data <- read_excel(file_path,
                           sheet="NMDS Happy", col_types = c("skip", "date",
                                                             "numeric", "text", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "numeric",
                                                             "numeric", "numeric", "text")) %>%
  rename_with(~ gsub(" ", "_", .x)) %>%
  rename_with(~ gsub("\\.", "_", .x))
```

Station names are arbitrary, and Ambrose expressed interest in renaming them from Stations 2, 4, 5 and 8 to Stations 1,2,3,and 4.

The `factor()` function by default sorts levels before assigning numeric codes, so a convenient way to replace the existing station codes with sequential numbers is to create a factor and extract the numeric indicator values with `as.numeric()`.

We'll have to make the same changes in other data too.

```
station_data <- station_data %>%
  mutate(station = factor(as.numeric(factor(station))))
head(station_data)
#> # A tibble: 6 x 61
#>   date                year month time_h latitude_decdeg longitude_decdeg riv_km
#>   <dttm>              <dbl> <chr>  <dbl>           <dbl>             <dbl>  <dbl>
#> 1 2013-05-28 00:00:00  2013 May    0.694            44.6              68.8  22.6
#> 2 2013-05-28 00:00:00  2013 May    0.604            44.6              68.8  13.9
#> 3 2013-05-28 00:00:00  2013 May    0.531            44.5              68.8   8.12
#> 4 2013-05-28 00:00:00  2013 May    0.415            44.5              68.8   2.78
#> 5 2013-07-25 00:00:00  2013 July   0.552            44.6              68.8  22.6
#> 6 2013-07-25 00:00:00  2013 July   0.477            44.6              68.8  13.9
#> # ... with 54 more variables: station <fct>, depth <dbl>,
#> #   discharge_week_cftpersec <dbl>, discharg_day <dbl>,
#> #   discharge_week_max <dbl>, tide_height <dbl>, ave_temp_c <dbl>,
#> #   ave_sal_psu <dbl>, ave_turb_ntu <dbl>, ave_do_mgperl <dbl>,
#> #   ave_DO_Saturation <dbl>, ave_chl_microgperl <dbl>, sur_temp <dbl>,
#> #   sur_sal <dbl>, sur_turb <dbl>, sur_do <dbl>, sur_chl <dbl>, bot_temp <dbl>,
#> #   bot_sal <dbl>, bot_turb <dbl>, bot_do <dbl>, bot_chl <dbl>, ...
```

## Composition Data

```
filename.in <- "Penobscot_Zooplankton and field data_EA_2.13.20.xlsx"
file_path <- file.path(data_folder, filename.in)
zoopl <- read_excel(file_path,
                    sheet = "NMDS Happy",
                    col_types = c("date",
                                  "text", "numeric", "numeric", "text",
                                  "text", "text", "text", "text", "text",
                                  "text", "numeric", "text", "text",
                                  "numeric", "numeric", "numeric",
                                  "text", "text", "text", "numeric",
                                  "numeric", "numeric", "numeric")) %>%
rename_with(~ gsub(" ", "_", .x)) %>%
  select(-c(`...20`:`...24`))
#> New names:
#> * `` -> ...20
#> * `` -> ...21
#> * `` -> ...22
#> * `` -> ...23
#> * `` -> ...24
```

We rename stations here as well. The code is similar.

```
zoopl <- zoopl %>%
  mutate(STATION = factor(as.numeric(factor(STATION))))
head(zoopl)
#> # A tibble: 6 x 19
#>   DATE                Month   Year STATION PHYLUM CLASS `SUB-CLASS` ORDER FAMILY
#>   <dttm>              <chr>  <dbl> <fct>   <chr>  <chr> <chr>       <chr> <chr>
#> 1 2015-09-16 00:00:00 Septe~  2015 4       Arthr~ Maxi~ Copepoda    <NA>  <NA>
#> 2 2014-05-02 00:00:00 May     2014 1       Arthr~ Maxi~ Copepoda    Cala~ <NA>
#> 3 2017-07-12 00:00:00 July    2017 3       Unkno~ <NA>  <NA>        <NA>  <NA>
#> 4 2016-07-20 00:00:00 July    2016 4       Unkno~ <NA>  <NA>        <NA>  <NA>
#> 5 2015-09-16 00:00:00 Septe~  2015 3       Unkno~ <NA>  <NA>        <NA>  <NA>
#> 6 2017-10-11 00:00:00 Octob~  2017 1       Unkno~ Unid~ <NA>        <NA>  <NA>
#> # ... with 10 more variables: GENUS <chr>, SPECIES <chr>, QUANTITY <dbl>,
#> #   LOWEST_TAXA <chr>, NAME <chr>, TOTAL_#_ORGANISMS <dbl>,
#> #   CORRECTED_PERCENT_ABUNDANCE <dbl>, NET_MESH_SIZE_(MICRONS) <dbl>,
#> #   NOTES <chr>, Picture_number <chr>
```

## Turn Data from Long to Wide

I no longer use the `reshape2` package (`dcast()` is from `reshape2`) for this type of data reorganization. Grouped tibbles in the tidyverse work well instead of the `aggregate()` command. For pivoting long to wide, the tidyverse's newer `pivot_wider()` function is fairly intuitive. But this code still works, so there is no reason to change it.

This step generates a total abundance for each taxa by site and date. I believe this was necessary because the "raw" data reflected sampling of plankton one microscope slide at a time. . . .

```
zoopl2 <- aggregate(CORRECTED_PERCENT_ABUNDANCE ~ NAME + DATE + STATION,
                    data=zoopl, FUN=sum)
#head(zoopl2)
```

The next step pivots the table to "wide" format, with a column for each taxa.

It is interesting that this step is necessary. Presumably it reflects the matrix format used for traditional community analysis, especially vegetation analysis. A matrix format was used by a lot of historically important community analysis code (like DECORANA).

```
zoopw <- dcast(zoopl2, DATE + STATION ~ NAME, drop = TRUE, fill = 0)
#> Using CORRECTED_PERCENT_ABUNDANCE as value column: use value.var to override.
#head(zoopw)
```

## Matrix of Species for `vegan`

The `vegan` package likes to work with a matrix of species occurrences. Although the matrix can have rownames that provide sample identifiers, that was not done here. Note that the "matrix" I produce here is really a data frame with nothing but numeric values. While those are quite different data structures internally, `vegan` handles the conversion in the background.

```
CDATA <- zoopw[,-c(1,2)]
```

We will put all our sample identifiers and environmental variables in other data frames that correspond row by row to the community data. Just remember when you drop a row from one data frame, you'll have to drop the same row from all related data frames!

First, we build a `header_data` file. The environmental data will go into a third data frame. We'll pick and chose which of those variables to work with later.

I notice here that the Year, Month and Day factors were being constructed by pulling substrings. That is interesting, as the DATE value in `zoopw` is actually a POSIXct date value. I had forgotten that you could treat them as strings successfully. (I also label the month factor, for later graphics).

## Header Data

```
header_data <- tibble(station = zoopw$STATION,
                          date = zoopw$DATE,
                          year = factor(as.numeric(substr(zoopw$DATE, 1,4))),
                          month = factor(as.numeric(substr(zoopw$DATE, 6, 7))),
                          day = factor(as.numeric(substr(zoopw$DATE, 9, 10)))) %>%
  mutate(month = factor(month, levels = 1:12, labels = month.abb))

head(header_data)
#> # A tibble: 6 x 5
#>   station date                year  month day
#>   <fct>   <dttm>              <fct> <fct> <fct>
#> 1 1       2013-05-28 00:00:00 2013  May   28
#> 2 2       2013-05-28 00:00:00 2013  May   28
#> 3 3       2013-05-28 00:00:00 2013  May   28
#> 4 4       2013-05-28 00:00:00 2013  May   28
#> 5 1       2013-07-25 00:00:00 2013  Jul   25
#> 6 2       2013-07-25 00:00:00 2013  Jul   25
```

I was concerned about behavior of the original code, if the sort order of factors were important. I don't think it is in this setting. In fact, except in some exploratory graphics (since deleted) we don't use those factors at all. I'm not sure we need factors at all, since we do not use these values in models. We could have left these values as strings, which might minimize risk of confusion. Still, it's worth thinking about the issues this brings up.

Pulling substrings returns a string. Creating factors with `factor()` will (by default) create levels based on sorted values. Here, those values are strings. When strings are sorted, "10" sorts before "9", as its first character ("1") sorts before "9". To be specific, that means the month factor will sort as January, October, November, December, February..., while the "day" factor will sort as 1, 10 - 19, 2, 20-29, 3, 30-31, 4-9.

To ensure proper sorting, I wrapped each of the `substr()` calls in `as.numeric()`. That way we build the sort orders on numeric values.As we don't model with these values, the distinction is probably not important here, but I like to be careful.

One could also extract similar values (and some others, like julian day, which is often useful for seasonal analysis) using date formats of the form:

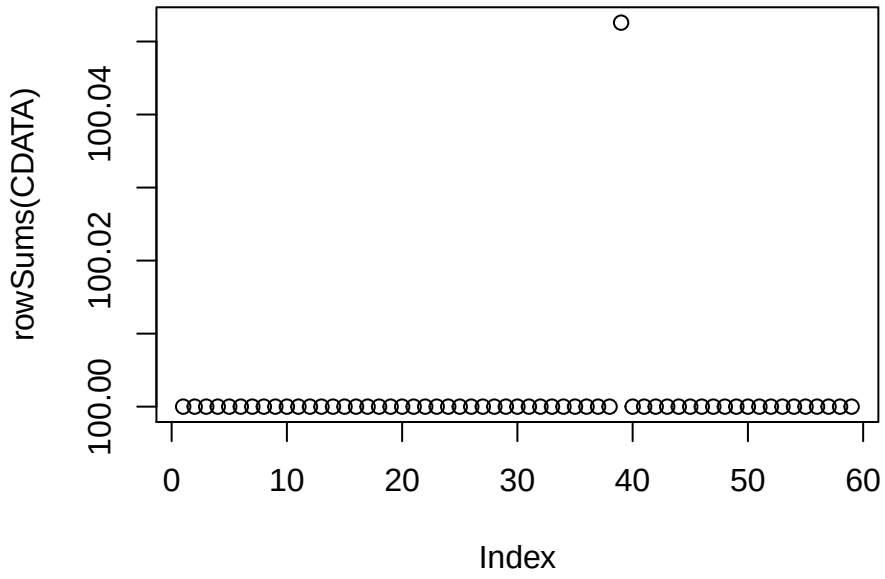
```
year = factor(as.numeric(format(zoopw$DATE, format = '%Y')))
```

(Note the format command also returns a string, so it also needs to be wrapped in a `as.numeric()` call).

## Data Sanity Checks

We should have no NAs, and row sums should all be 1, at least within reasonable rounding error.

```
anyNA(CDATA)
#> [1] FALSE
plot(rowSums(CDATA))
```



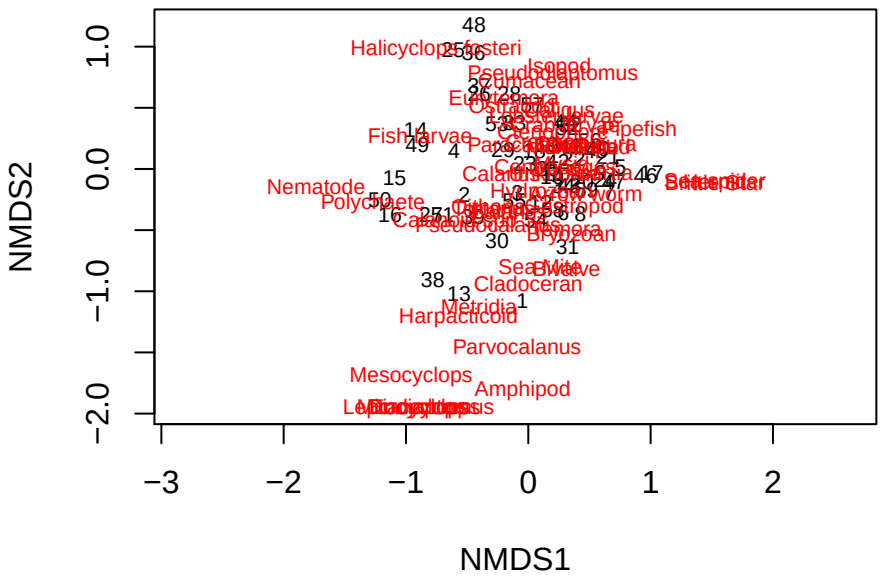

## NMDS Analyses

```
NMDSE <- metaMDS(CDATA, autotransform = FALSE, k = 2, trymax = 75)
#> Run 0 stress 0.1509449
#> Run 1 stress 0.1677185
#> Run 2 stress 0.1743709
#> Run 3 stress 0.1505044
#> ... New best solution
#> ... Procrustes: rmse 0.02781941  max resid 0.1876429
#> Run 4 stress 0.1577084
#> Run 5 stress 0.1729591
#> Run 6 stress 0.1987387
#> Run 7 stress 0.1684932
#> Run 8 stress 0.1508906
#> ... Procrustes: rmse 0.02214216  max resid 0.1417684
#> Run 9 stress 0.1504107
#> ... New best solution
#> ... Procrustes: rmse 0.006048624  max resid 0.03262898
#> Run 10 stress 0.1742828
#> Run 11 stress 0.1505673
#> ... Procrustes: rmse 0.03119367  max resid 0.2211219
```

```
#> Run 12 stress 0.1796661
#> Run 13 stress 0.1690316
#> Run 14 stress 0.1658234
#> Run 15 stress 0.1729591
#> Run 16 stress 0.168418
#> Run 17 stress 0.2174546
#> Run 18 stress 0.1709702
#> Run 19 stress 0.1505673
#> ... Procrustes: rmse 0.03119409  max resid 0.2211387
#> Run 20 stress 0.2036981
#> Run 21 stress 0.168418
#> Run 22 stress 0.1864582
#> Run 23 stress 0.15446
#> Run 24 stress 0.1690316
#> Run 25 stress 0.1835907
#> Run 26 stress 0.1509449
#> Run 27 stress 0.1505672
#> ... Procrustes: rmse 0.03119419  max resid 0.2211506
#> Run 28 stress 0.2319182
#> Run 29 stress 0.1796661
#> Run 30 stress 0.1504107
#> ... New best solution
#> ... Procrustes: rmse 1.536398e-05  max resid 7.972017e-05
#> ... Similar to previous best
#> *** Solution reached
NMDSE
#>
#> Call:
#> metaMDS(comm = CDATA, k = 2, trymax = 75, autotransform = FALSE)
#>
#> global Multidimensional Scaling using monoMDS
#>
#> Data:     CDATA
#> Distance: bray
#>
#> Dimensions: 2
#> Stress:     0.1504107
#> Stress type 1, weak ties
#> Two convergent solutions found after 30 tries
#> Scaling: centring, PC rotation, halfchange scaling
#> Species: expanded scores based on 'CDATA'
```

## Plot

```
plot(NMDSE, type = 't')
```




## Adding Environmental Data

I want to use the names of these variables as labels in graphics later. I capitalize variable names here, so they will appear capitalized in graphics without further action on my part.

```
envNMDS <- data.frame(NMDSE$points) %>%
  rownames_to_column(var = "sample") %>%
  mutate(Station = header_data$station) %>%
  mutate(Month = header_data$month) %>%
  mutate(Year = header_data$year) %>%
    mutate(Temp = station_data$ave_temp_c) %>%
    mutate(Sal = station_data$ave_sal_psu) %>%
    mutate(Turb = station_data$ave_turb_ntu) %>%
    mutate(DOsat = station_data$ave_DO_Saturation) %>%
    mutate(Chl = station_data$ave_chl_microgperl) %>%
    mutate(Fish = station_data$Fish) %>%
    mutate(RH = station_data$Herring)
head(envNMDS)
#>   sample        MDS1        MDS2 Station Month Year      Temp       Sal      Turb
#> 1      1 -0.04834133 -1.07698385       1   May 2013 11.651091  0.02000 17.71316
#> 2      2 -0.52259088 -0.21215040       2   May 2013  9.402855 14.57176  8.74000
#> 3      3 -0.08893372 -0.19140608       3   May 2013  6.974443 24.73746 10.50207
#> 4      4 -0.60640034  0.15271698       4   May 2013  9.510073 12.65232  6.20000
#> 5      5  0.75273422  0.01699826       1   Jul 2013 18.527296 15.99052 28.30037
#> 6      6  0.28658819 -0.35828127       2   Jul 2013 13.633036 26.98056 89.02592
#>   DOsat      Chl       Fish         RH
#> 1    NA 4.062250   52.63158   31.57895
```

```
#> 2    NA 2.175823  405.12048  367.46988
#> 3    NA 2.004560 2276.48579 2248.92334
#> 4    NA 2.159800   37.69401   34.92239
#> 5    NA 3.075191   47.89978   36.84598
#> 6    NA 4.297501  192.52078  179.36288
```

### Using `envfit` to Estimate Correlations

I revised your code to call on the same ordination you were using elsewhere. You were calling on another NMDS object. I believe it was one produced by `isoMDS()` from `MASS`, not `metaMDS()` from `vegan`. I'm not sure why that happened or whether it matters.

```
ef <- envfit(NMDSE, envNMDS[,c(4:11,13)], permu = 999, na.rm = TRUE)
ef
#>
#> ***VECTORS
#>
#>          NMDS1    NMDS2     r2 Pr(>r)
#> Temp   0.97222  0.23408 0.7512  0.001 ***
#> Sal    0.99654  0.08312 0.0706  0.203
#> Turb  -0.59731  0.80201 0.1550  0.035 *
#> DOsat -0.97482 -0.22297 0.3369  0.001 ***
#> Chl    0.73970 -0.67293 0.0746  0.192
#> RH    -0.23520  0.97195 0.2431  0.001 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#> Permutation: free
#> Number of permutations: 999
#>
#> ***FACTORS:
#>
#> Centroids:
#>            NMDS1   NMDS2
#> Station1  0.1747  0.2790
#> Station2 -0.1544  0.3196
#> Station3 -0.1882 -0.2354
#> Station4  0.0612 -0.2390
#> MonthMay -0.7440 -0.0008
#> MonthJul  0.1787  0.0050
#> MonthSep  0.4020  0.0946
#> MonthOct  0.3039  0.0615
#> Year2014 -0.0358 -0.0654
#> Year2015 -0.1294  0.1257
#> Year2016  0.1609  0.0834
#> Year2017 -0.1449 -0.0134
#>
#> Goodness of fit:
#>             r2 Pr(>r)
#> Station 0.1935  0.010 **
#> Month   0.4694  0.001 ***
#> Year    0.0424  0.676
#> ---
```

```
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#> Permutation: free
#> Number of permutations: 999
#>
#> 13 observations deleted due to missingness
```

I note 13 observations deleted due to missingness. Most of those must be the 2013 data. Presumably, they were deleted because they lack some of the predictor variables.

I also observe that Year is included in the model, but is NOT significant. However, dropping Year from the model has no effect on the vector NMDS loading estimates, or their R squared values, so it does not matter.

## Extracting Vector Information

The `envfit()` help page says the object returned by the function is a LIST with three components.

```
class(ef)
#> [1] "envfit"
names(ef)
#> [1] "vectors"   "factors"   "na.action"
```

The `ef` object is a `envfit` object, with C3 class, with three named slots. The vector information we need to plot the environment arrows is available in `vectors`. But that object is itself also an S3 object, with several named items.

```
v <- ef$vectors
class(v)
#> [1] "vectorfit"
names(v)
#> [1] "arrows"       "r"            "permutations" "pvals"        "control"
```

The help page for `envfit()` tells us that the information we need for the direction of the arrows is in the `arrows` component. We are told that 'arrows contains "Arrow endpoints from vectorfit. The arrows are scaled to unit length."

```
class(v$arrows)
#> [1] "matrix" "array"
v$arrows
#>            NMDS1       NMDS2
#> Temp   0.97722176  0.23407896
#> Sal    0.9965395  0.08312086
#> Turb  -0.5973127  0.80200847
#> DOsat -0.9748243 -0.22297421
#> Chl    0.7397028 -0.67293371
#> RH    -0.2351998  0.97194704
#> attr(,"decostand")
#> [1] "normalize"
```

The information we need to determine the magnitude of those vectors is buried in the `r` component of the `vectors` component. We scale each of the arrows by the square root of the related r squared value. (Following the strategy in the code shared with me).

```
arrows <- v$arrows
rsq    <- v$r
scaled_arrows <- as_tibble(arrows*sqrt(rsq)) %>%
  mutate(parameter = rownames(arrows))
```

I was actually a bit surprised that worked. I tend to work in dataframes, where this would be impossible. This works because both `arrows` and `rsq` are arrays.

Internally, and array is just a vector with dimensions.

When we multiply an array by a vector, R does not do matrix multiplication, but multiplies element by element, recycling the shorter vector as needed. Since the number of rows in our array matches the number of values in the vector, values line up, and we multiply rowwise.

This gets messy for vectors and arrays that are NOT of compatible dimensions. Some examples to make that clearer:

```
a <- 1:6
dim(a) <- c(2,3)
b <- c(1,2)
c <- c(1,2,3)
d <- c(1,2,3,4)
cat('Compatible Dimensions: Vector length matches rows in array\n')
#> Compatible Dimensions: Vector length matches rows in array
a*b
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    4    8   12
cat('Incompatible dimensions -- array values mutiplied by column, then row\n')
#> Incompatible dimensions -- array values mutiplied by column, then row
a*c
#>      [,1] [,2] [,3]
#> [1,]    1    9   10
#> [2,]    4    4   18
cat('If values are not divisible, extras from the shorter vector are dropped\n')
#> If values are not divisible, extras from the shorter vector are dropped
a*d
#> Warning in a * d: longer object length is not a multiple of shorter object
#> length
#>      [,1] [,2] [,3]
#> [1,]    1    9    5
#> [2,]    4   16   12
```

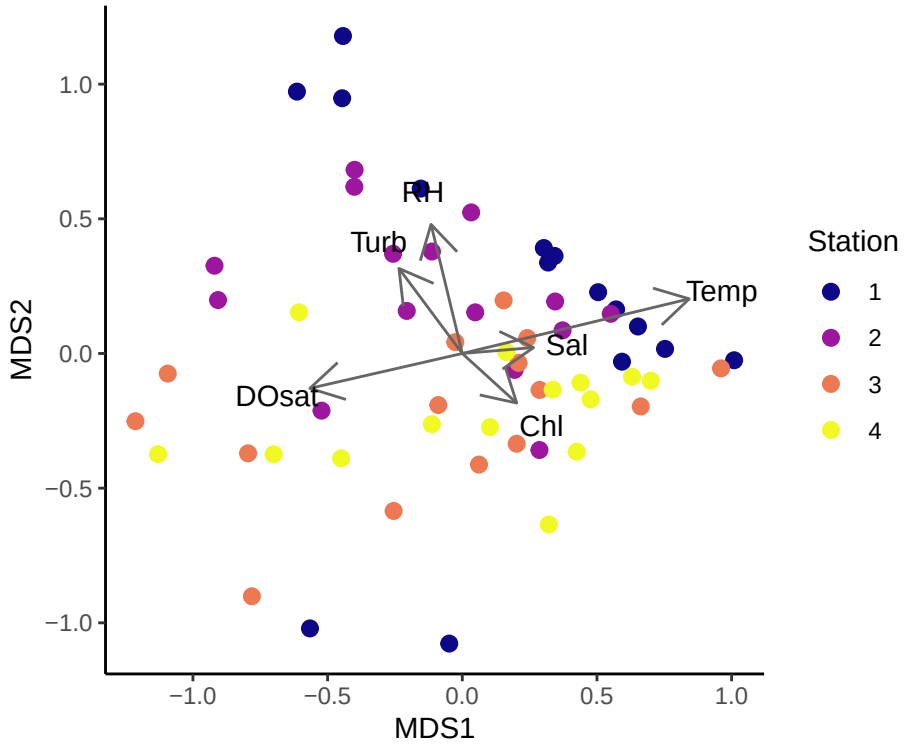
While we are creating vectors, we also want to create points for placing the annotations identifying each vector. We want to space the labels so they are a fixed distance beyond the end of each vector. We do that with a little vector addition. What we want is to add a small length to each scaled vector to offset the text. Vector addition is just element wise addition. We can pull values from the values in `arrows`, which are vectors scaled to unit length, as offsets from zero, which makes the math fairly easy. Note that we may want that extra space to differ for plots srawn at different scales or with different fonts.

```
scale_factor = 0.125   # Fraction of unit length beyond arrow to place annotation

scaled_arrows <- scaled_arrows %>%
  mutate(ann_xpos = NMDS1 + arrows[,1]*scale_factor,
         ann_ypos = NMDS2 + arrows[,2] *scale_factor)
```

### Plotting `envfit()` Information

```
plot_data <- data.frame(NMDSE$points) %>%
  rownames_to_column(var = "sample") %>%
  mutate(station = header_data$station) %>%
  mutate(month = header_data$month) %>%
  mutate(year = header_data$year)
```

```
plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_point(aes(color = station), size = 2.5) +
  geom_segment(data=scaled_arrows,
               mapping = aes(x=0,xend=NMDS1,y=0,yend=NMDS2),
               arrow = arrow(length = unit(0.5, "cm")) ,colour="grey40") +
  geom_text(data=scaled_arrows,
            mapping = aes(x=ann_xpos,y=ann_ypos,label=parameter),
            size=4, nudge_x =0, nudge_y = 0, hjust = .5)+
  scale_color_viridis_d(option = 'C', name = 'Station') +
  coord_fixed()
plt
```




## Cluster Analysis

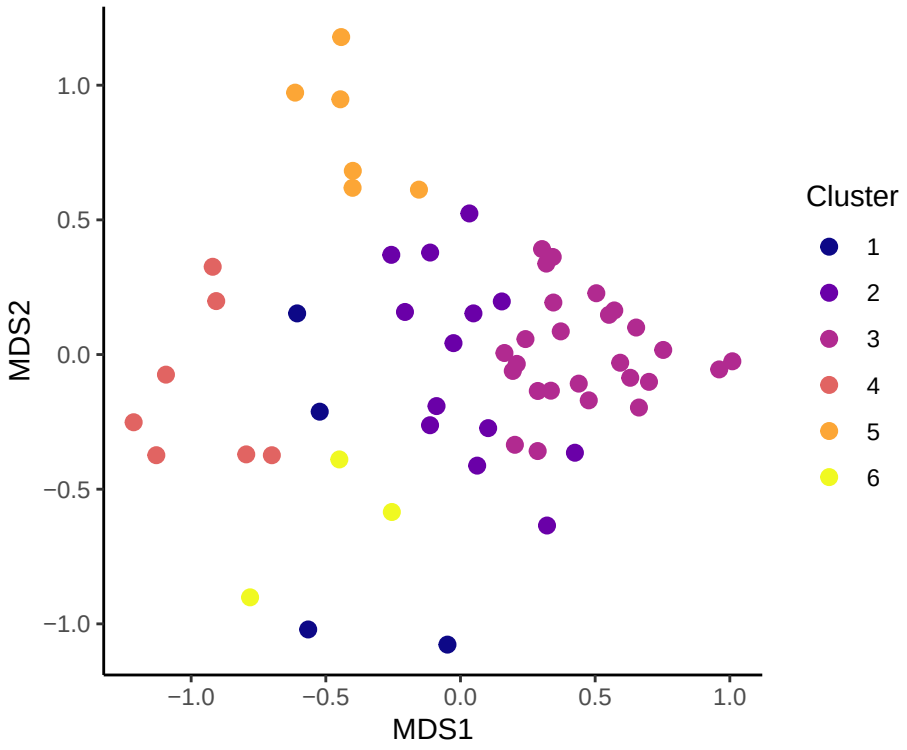
```
d <- vegdist(CDATA, "bray") # Bray-Curtis default
clust <- hclust(d)          # This is agglomerative clustering - build the groups
```

```
cut6 <- cutree(clust, 6)     # from a single observation not split them apart...
                             # this cut number is arbitrary - we can pick what we
                              # want. BUT having more than 7 groups is hard
                              # because of what you can visually see... play
                              # around and see what is most informative.
```

`cut6` is a vector with the cluster assignment of each sample, so we can merge it back into any of our other sample-oriented data structures.

### Plot Clusters

```
plot_data <- plot_data %>%
  mutate(cluster = factor(cut6))

plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_point(aes(color = cluster), size = 2.5) +
  scale_color_viridis_d(option = 'C', name = 'Cluster') +
  coord_fixed()
plt
```




## Drawing Minimum Bounding Polygons

the function `ordihul()` draws minimum bounding polygons around points in each cluster.

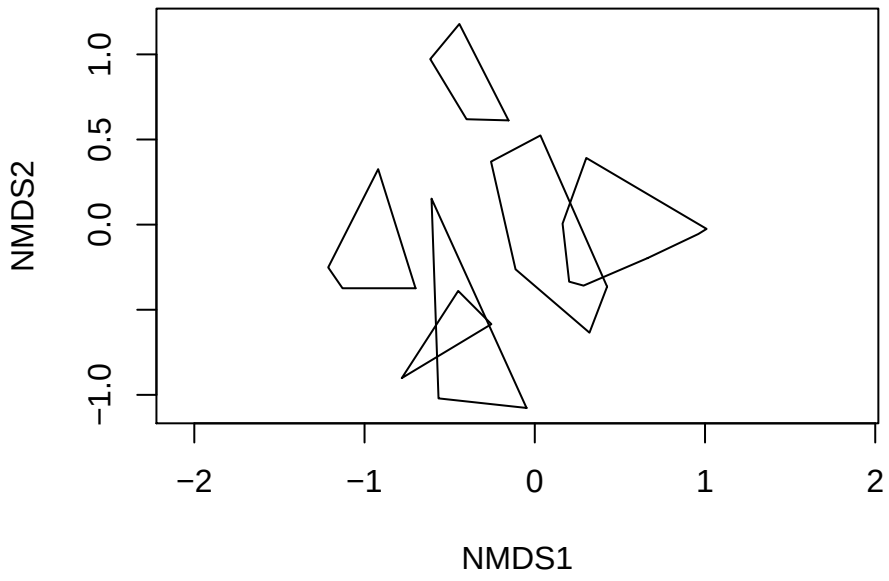
### Understanding `ordihull()`

It appears `ordihull()` must be called after a plot object has been created.

```
ordihull(NMDSE, groups = cut6, display = "sites")
#> Error in plot.xy(xy.coords(x, y), type = type, ...): plot.new has not been called yet
```

So, we create a plot, then call `ordiplot()`, and see what we get. The `ordihull()` helpfile is not very informative, saying only that functions "return the invisible plotting structure." But what that means is that the `ordihull()` function returns information invisibly while it modifies a plot. If we can capture the returned plotting structure, we can examine and re-use it.

```
plot(NMDSE, type = "n", display = "sites")
hull <- ordihull(NMDSE, groups = cut6, display = "sites")
```




`vegan` is built largely on S3 classes, which are implemented as named lists, so it's easy to find a starting point by looking at the names in the object returned by `ordihull()`.

```
class(hull)
#> [1] "ordihull"
names(hull)
#> [1] "1" "2" "3" "4" "5" "6"
```

I doubt it is a coincidence that the list has six objects and we defined six clusters. We look at the first item in this list.

```
class(hull[[1]])
#> [1] "matrix" "array"
hull[[1]]
#>          NMDS1     NMDS2
#> 1  -0.04834133 -1.076984
#> 13 -0.56540524 -1.020760
#> 4  -0.60640034  0.152717
#> 1  -0.04834133 -1.076984
```

It's just an array containing the points of the vertexes of the polygons. Each polygon is passed as an array of points. We can work with that, although it is going to be easier to "flatten" the data structure.

This is a bit tricky, as we need to convert each array to a data frame and append them, retaining their cluster identities. This can be done in several ways. Here I convert the arrays to tibbles, then bind them into one tibble with `bind_rows()`, which conveniently allows you to label each entry with the source data frame (here the cluster number).

```
hullsdfs <- map(hull, as_tibble)
hulls_df <- hullsdfs %>%
  bind_rows(.id = 'Cluster')
hulls_df
#> # A tibble: 32 x 3
#>    Cluster   NMDS1  NMDS2
#>    <chr>     <dbl>  <dbl>
#>  1 1       -0.0483 -1.08
#>  2 1       -0.565  -1.02
#>  3 1       -0.606   0.153
#>  4 1       -0.0483 -1.08
#>  5 2        0.425  -0.365
#>  6 2        0.321  -0.635
#>  7 2       -0.113  -0.262
#>  8 2       -0.257   0.370
#>  9 2        0.0331  0.524
#> 10 2        0.425  -0.365
#> # ... with 22 more rows
```

## Final Graphics

The instructions to authors suggests figure widths should line up with columns, and proposes figure widths should be: 39, 84, 129, or 174 mm wide, with height not to exceed 235 mm. Presumably that corresponds to 1,2,3,or 4 columns wide?

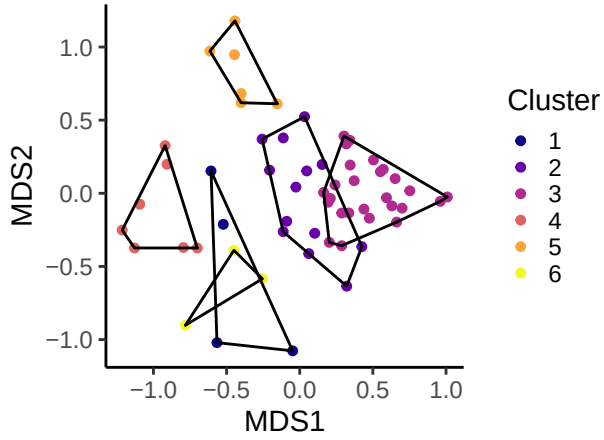
39 mm is about one and one half inches, which his quite small, so we will make the 84 mm and 129 mm wide options. Unfortunately RMarkdown / `knitr` likes figure dimensions in inches. 84 mm is close to 3.3 inches. 129 mm is close to 5 inches. I hope those values are close enough.

Note that ggplot sometimes scales plots in odd ways when you specify both figure height and width, especially if the relative scaling is constrained by fixed axis limits, as here , with `coord_fixed()`. I provide both for consistency. In my experience, otherwise you get surprises in eps / pdf output.

Also, in my experience, graphics produced by different graphics devices often look somewhat different. In this case, I think the EPS graphics have problems placing the letters from the Y axis label, although it is hard to tell, because without graphics software that handles EPS files well, I can just barely view the EPS files. Things look crisper in PDF, perhaps only because PDF viewers from Adobe are fairly sophisticated.

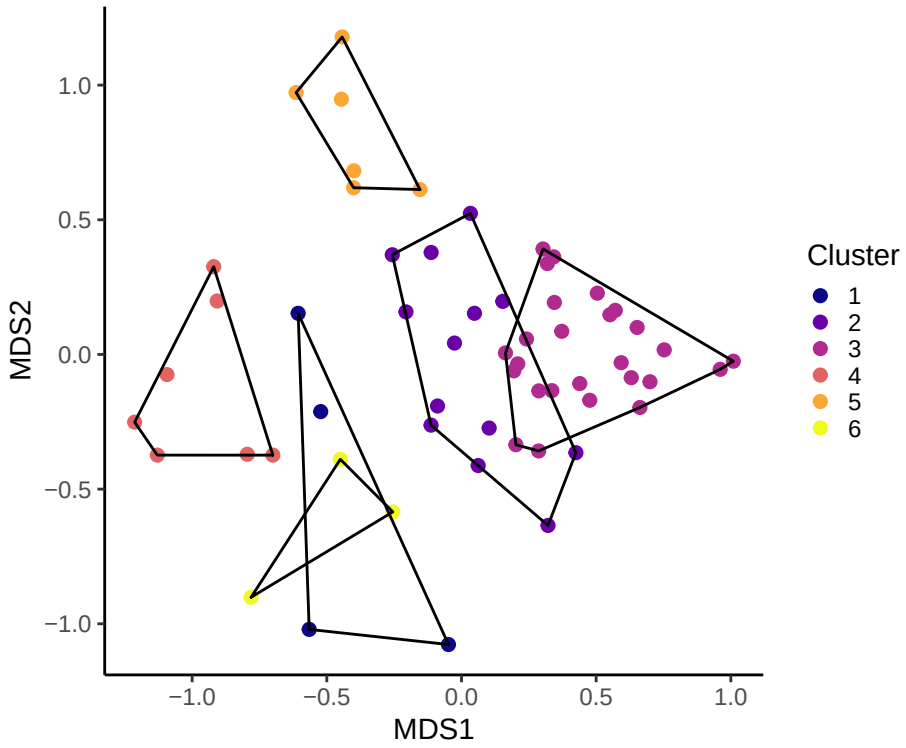
### Plot Clusters and Convex Hulls

```
plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_point(aes(color = cluster), size = 1.25) +
  geom_polygon(data=hulls_df,
               mapping = aes(x= NMDS1,y= NMDS2, group = Cluster),
               color = 'black', fill = NA) +
  scale_color_viridis_d(option = 'C', name = 'Cluster') +

  # Adjust size of legend
  theme(legend.key.size = unit(0.35, 'cm')) +

  # Set aspect ratio (defaults to 1)
  coord_fixed()
plt
```




```
ggsave('clusters_84.eps', device = 'eps', width = 3.3, height = 2.75)
ggsave('clusters_84.tif', device = 'tiff', width = 3.3, height = 2.75)
ggsave('clusters_84.eps', device = cairo_ps, width = 3.3, height = 2.75)
ggsave('clusters_84.pdf', device = cairo_pdf, width = 3.3, height = 2.75)
```

```
plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_point(aes(color = cluster), size = 2) +
  geom_polygon(data=hulls_df,
               mapping = aes(x= NMDS1,y= NMDS2, group = Cluster),
               color = 'black', fill = NA) +
  scale_color_viridis_d(option = 'C', name = 'Cluster') +

  # Adjust size of legend
  theme(legend.key.size = unit(0.35, 'cm')) +

  # Set aspect ratio (defaults to 1)
  coord_fixed()
plt
```



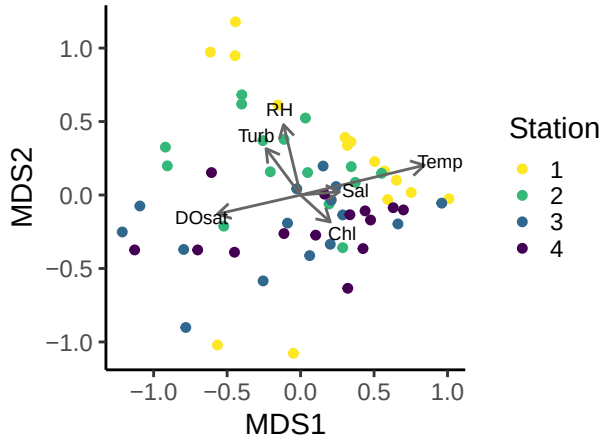

```
ggsave('clusters_129_alt.eps', device = "eps", width = 5, height = 4)
ggsave('clusters_129.tif', device = "tiff", width = 5, height = 4)
ggsave('clusters_129.eps', device = cairo_ps, width = 5, height = 4)
ggsave('clusters_129.pdf', device = cairo_pdf, width = 5, height = 4)
```

### Plot Environment Arrows

```
scale_factor = 0.11   # Fraction of unit length beyond arrow to place annotation

scaled_arrows <- scaled_arrows %>%
  mutate(ann_xpos = NMDS1 + arrows[,1]*scale_factor,
         ann_ypos = NMDS2 + arrows[,2] *scale_factor)
```

```
plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_point(aes(color = station), size = 1.25) +
  geom_segment(data=scaled_arrows,
               mapping = aes(x=0, xend = NMDS1, y = 0, yend = NMDS2),
               arrow = arrow(length = unit(0.2, "cm")) ,colour="grey40") +
  geom_text(data=scaled_arrows,
            mapping = aes(x = ann_xpos, y = ann_ypos,label=parameter),
            size=2.5, hjust = 0.5)+
  scale_color_viridis_d(option = 'D', direction = -1, name = 'Station') +

  # Adjust size of legend
  theme(legend.key.size = unit(0.35, 'cm')) +
```

```
  # Set aspect ratio (defaults to 1)
  coord_fixed()
plt
```

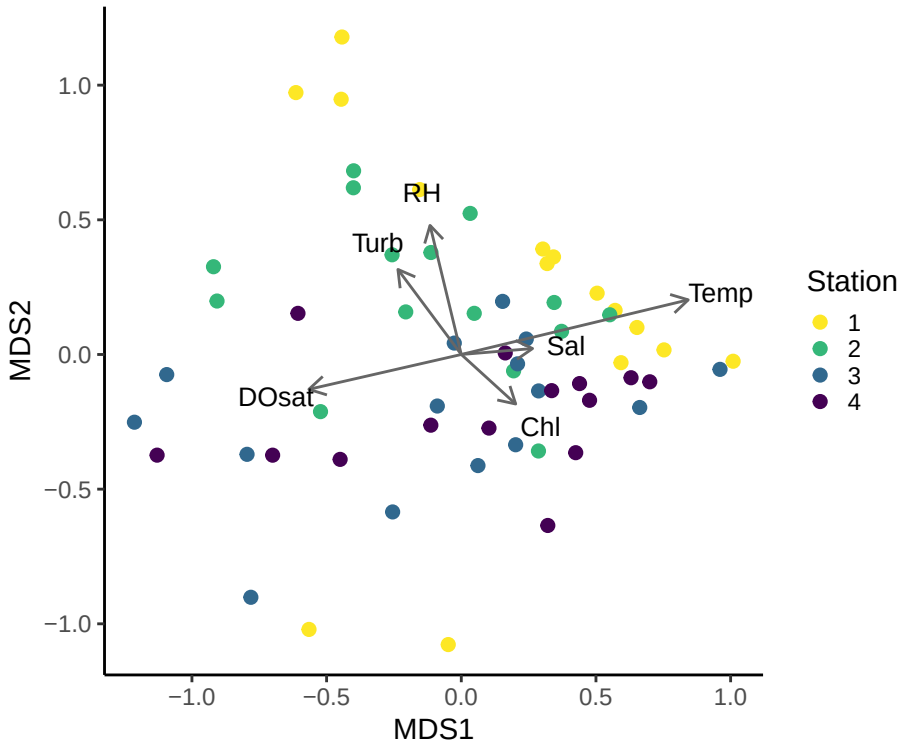



```
ggsave('arrows_84_alt.eps', device = "eps", width = 3.3, height = 2.5)
ggsave('arrows_84.tif', device = "tiff", width = 3.3, height = 2.5)
ggsave('arrows_84.eps', device = cairo_ps, width = 3.3, height = 2.5)
ggsave('arrows_84.pdf', device = cairo_pdf, width = 3.3, height = 2.5)
```

```
scale_factor = 0.125   # Fraction of unit length beyond arrow to place annotation

scaled_arrows <- scaled_arrows %>%
  mutate(ann_xpos = NMDS1 + arrows[,1]*scale_factor,
         ann_ypos = NMDS2 + arrows[,2] *scale_factor)
```

```
plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_point(aes(color = station), size = 2) +
  geom_segment(data=scaled_arrows,
               mapping = aes(x=0, xend = NMDS1, y = 0, yend = NMDS2),
               arrow = arrow(length = unit(0.25, "cm")) ,colour="grey40") +
  geom_text(data=scaled_arrows,
            mapping = aes(x = ann_xpos, y = ann_ypos,label=parameter),
            size=3.5, hjust = 0.5)+
  scale_color_viridis_d(option = 'D', direction = -1, name = 'Station') +

  # Adjust size of legend
  theme(legend.key.size = unit(0.35, 'cm')) +
```

```
  # Set aspect ratio (defaults to 1)
  coord_fixed()
plt
```




```
ggsave('arrows_129_alt.eps', device = "eps", width = 5, height = 4)
ggsave('arrows_129.tif', device = "tiff", width = 5, height = 4)
ggsave('arrows_129.eps', device = cairo_ps, width = 5, height = 4)
ggsave('arrows_129.pdf', device = cairo_pdf, width = 5, height = 4)
```

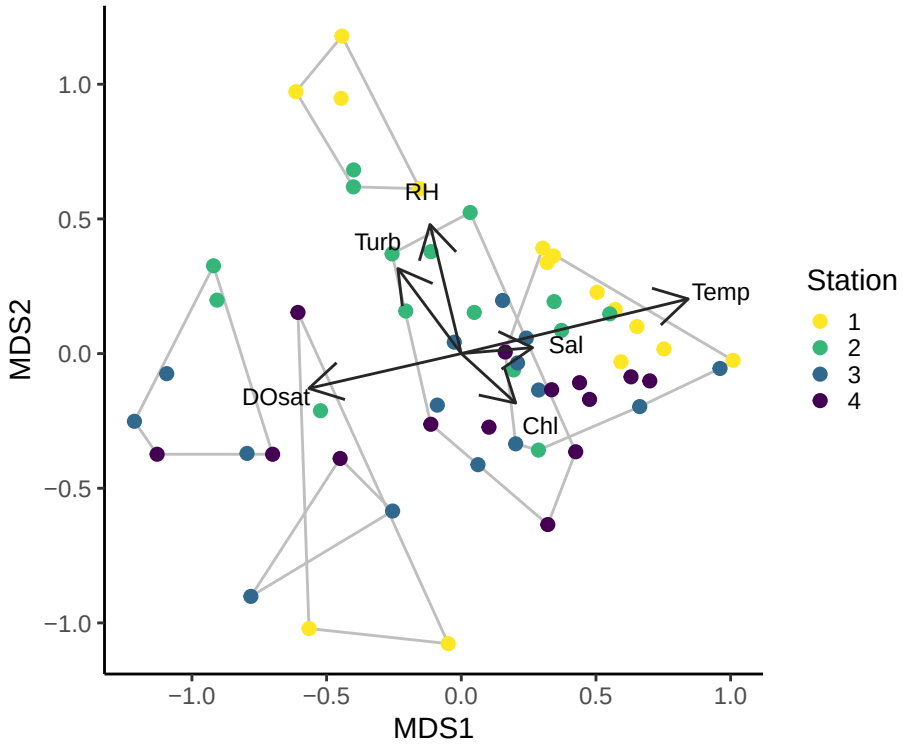
## Combined Graphics

Lasley-Rasher asked whether it would be possible to combine the graphics into one. This poses a design challenge, as we then have three different types of information on the graphic, and we need to figure out how to code all that information in an understandable way.

### Colored by Station

```
plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_polygon(data=hulls_df,
               mapping = aes(x= NMDS1,y= NMDS2, group = Cluster),
               colour = "gray75",
               fill = NA,
              # alpha = 0.2
               ) +
```

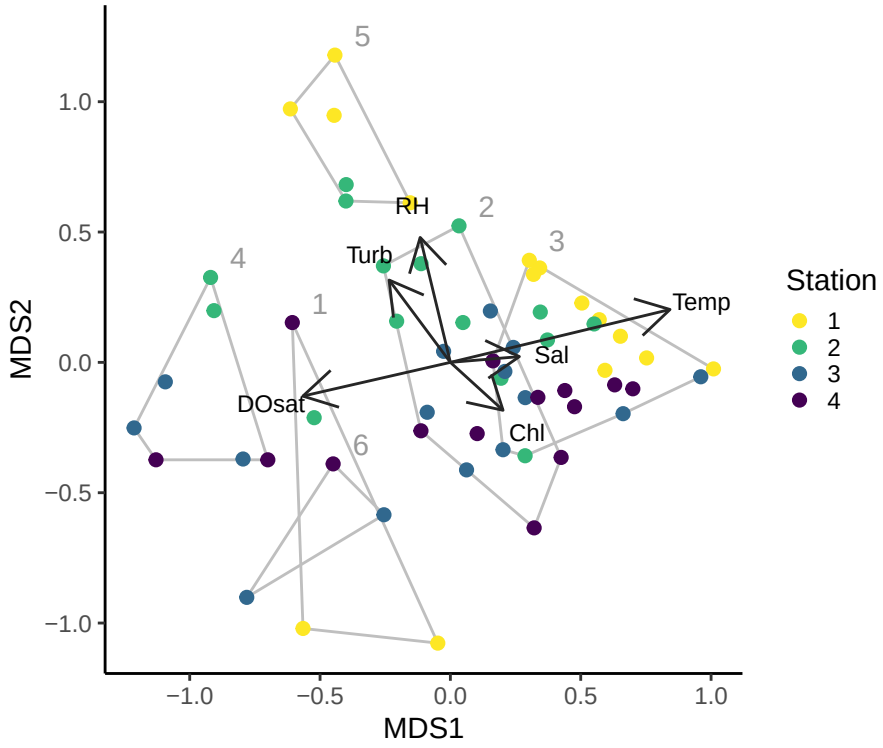
```
  geom_point(aes(color = station), size = 2) +

  geom_segment(data=scaled_arrows,
               mapping = aes(x=0,xend=NMDS1,y=0,yend=NMDS2),
               arrow = arrow(length = unit(0.5, "cm")) ,colour="gray15") +

  scale_color_viridis_d(option = 'D', direction = -1, name = 'Station') +
  #scale_fill_viridis_d(option = 'D', name = 'Cluster') +

  # Adjust size of legend
  theme(legend.key.size = unit(0.35, 'cm')) +

  # Set aspect ratio (defaults to 1)
  coord_fixed()  #+
  #theme(legend.position = c(0.9, 0.75),
  #      legend.background = element_blank())

plt +
   geom_text(data=scaled_arrows,
             mapping = aes(x=ann_xpos,y=ann_ypos,label=parameter), colour = "black",
             size=3.25, hjust = 0.5)
```




**Add Annotations for the Clusters** I can think of one moderately convenient way to label the polygons "automatically". That is to pull the top point in each cluster, and place the labels near that point. otherwise, we may want to place them manually.

```
labs <- hulls_df %>%
  group_by(Cluster) %>%
  filter(NMDS2 == max(NMDS2))
labs
#> # A tibble: 6 x 3
#> # Groups:   Cluster [6]
#>   Cluster   NMDS1  NMDS2
#>   <chr>     <dbl>  <dbl>
#> 1 1       -0.606   0.153
#> 2 2        0.0331  0.524
#> 3 3        0.303   0.392
#> 4 4       -0.920   0.326
#> 5 5       -0.443   1.18
#> 6 6       -0.450  -0.389
```

```
plt +
  geom_text(data = labs,
            mapping = aes(x = NMDS1,y = NMDS2, label = Cluster),
            colour = "gray60", size=4,
            nudge_x = 0.075, nudge_y = 0.075, hjust = 0) +
   geom_text(data=scaled_arrows,
            mapping = aes(x = ann_xpos, y = ann_ypos, label = parameter),
            colour = "black", size = 3.25, hjust = 0.5)
```

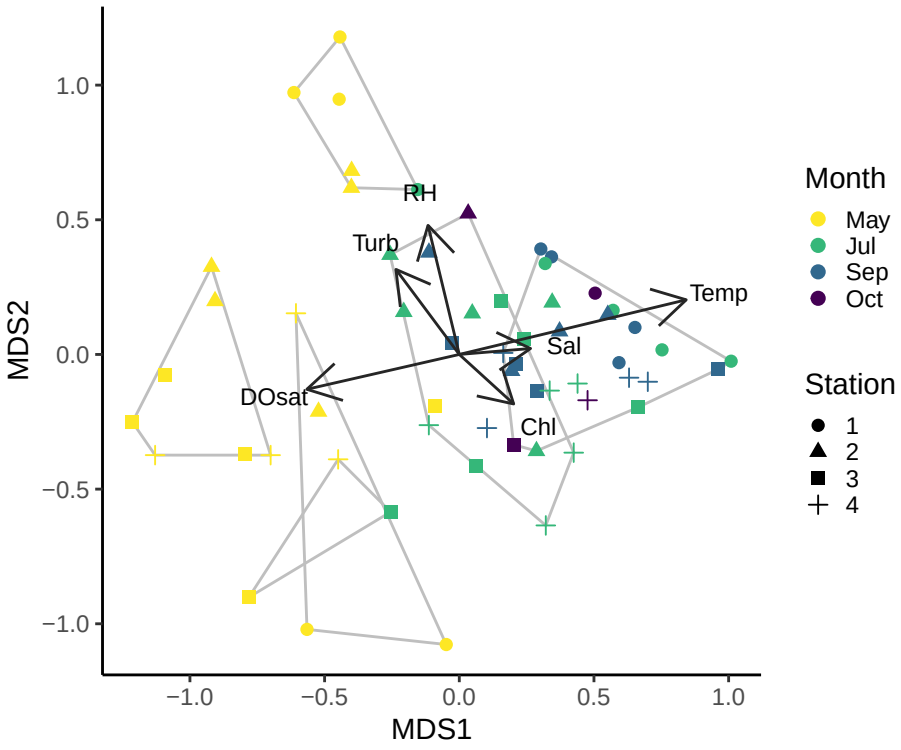



```
ggsave('both_129_alt.eps', device = "eps", width = 5, height = 4)
ggsave('both_129.tif', device = "tiff", width = 5, height = 4)
ggsave('both_129.eps', device = cairo_ps, width = 5, height = 4)
ggsave('both_129.pdf', device = cairo_pdf, width = 5, height = 4)
```

### Colored by Month

This was added as a working graphic to highlight the seasonal pattern in community composition.

```
plt <- ggplot(data = plot_data, aes(MDS1, MDS2)) +
  geom_polygon(data=hulls_df,
               mapping = aes(x= NMDS1,y= NMDS2, group = Cluster),
               colour = "gray75",
               fill = NA,
              # alpha = 0.2
               ) +

  geom_point(aes(color = month, shape = station), size = 2) +

  geom_segment(data=scaled_arrows,
               mapping = aes(x=0,xend=NMDS1,y=0,yend=NMDS2),
               arrow = arrow(length = unit(0.5, "cm")) ,colour="gray15") +

  scale_color_viridis_d(option = 'D', direction = -1, name = 'Month') +
  scale_shape(name = 'Station') +
  #scale_fill_viridis_d(option = 'D', name = 'Cluster') +

  # Adjust size of legend
  theme(legend.key.size = unit(0.35, 'cm')) +

  # Set aspect ratio (defaults to 1)
  coord_fixed()  #+
  #theme(legend.position = c(0.9, 0.75),
  #      legend.background = element_blank())

plt +
   geom_text(data=scaled_arrows,
             mapping = aes(x=ann_xpos,y=ann_ypos,label=parameter), colour = "black",
             size=3.25, hjust = 0.5)
```


1.0
0.5
0.0
−0.5
−1.0
MDS2
−1.0
−0.5
0.0
0.5
1.0
MDS1
RH
Turb
Temp
Sal
DOsat
Chl
Month
May
Jul
Sep
Oct
Station
1
2
3
4