Week 1 Exercise: Basic R

Z620: Quantitative Biodiversity, Indiana University

January 16, 2015

OVERVIEW

This exercise will introduce you to some of the basic features of the R computing environment. We will briefly cover operators, data types, and simple commands that will be useful for you during the course and beyond. In addition to using R's base package, we wil also use a couple of contributed packages, which together will allow us to visualized data and peform relatively simple statistics (e.g., linear regression and ANOVA).

HOW WE WILL BE USING R AND OTHER TOOLS

During the course, we will use RStudio, which is a relatively user-friendly integrated developement environment that allows R to interface with other tools. For example, you are currently working in an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.

We will also use a tool called knitr, which is a package that generates reports from R script and Markdown text. For example, when you click the **Knit PDF** button in the scripting window of Rstudio, a document will be generated that includes both the content as well as the output of any embedded R code. If there are errors in your markdown document, however, you will not be able to knit a PDF file. Assignments in this class will require that you successfully create a PDF using knitr and submit it to the course respository hosted on GitHub.

SETTING YOUR WORKING DIRECTORY

The first step of working with R is to set your working directory. This is where your R script and output will be saved. It's also a logical place to put data files that you plan to import into R.

The following command will return your current working directory:

```
getwd()
```

```
## [1] "/Users/lennonj/GitHub/QuantitativeBiodiversity/Assignments/Week1"
```

If you want to change your working directory, use the following command (note: you will need to modify to reflect your actual directory):

```
setwd("~/GitHub/QuantitativeBiodiversity/Assignments/Week1")
```

USING R AS A CALCULATOR

R is capable of performing various calcuations using simple operators and built-in **functions**

addition:

```
1 + 3
```

```
## [1] 4
```

subtraction:

```
3 - 1
```

```
## [1] 2
```

multiplication (with an exponent):

```
3 * 10^2
```

```
## [1] 300
```

division (using a built-in constant):

```
10 / pi
```

```
## [1] 3.183
```

trigonometry with a simple built-in **function** (i.e., *sin*) that takes an **argument** (i.e., '4'):

```
sin(4)
```

```
## [1] -0.7568
```

logarithms (another example of functions and arguments)

```
log10(100)
```

```
## [1] 2
```

```
log(100)
```

```
## [1] 4.605
```

ASSIGNING VARIABLES

In R, you will often find it useful and necessary to assign values to a variable or **object**. Generally speaking, it's best to use `<-` rather than `=` as an assignment operator.

```
a <- 10
b <- a + 20
```

What is the value of b?

Now let's reassign a new value to `a`:

```
a <- 200
```

Now, what is the value of 'b'? What's going on?

R holds onto the original value of 'a' that was used when assigning values to 'b'. You can correct this using the `rm` function, which removes objects from your R **environment**.

```
rm("b")
```

What happens if we reassign `b` now?

```
b <- a + 20
```

In some situations, it's good practice to clear all variables from your R environment. This can be done in a couple of ways. For example, you can just click `clear'in the Environment windwow of R Studio. The same procedure can be performed at the command line. The`ls' function allows us to view a list of objects in our R environment:

```
ls()
```

```
## [1] "a" "b"
```

We can now clear all of the stored variables from R's memory (using two functions: `rm` and `ls`):

```
rm(list=ls())
```

WORKING WITH VECTORS AND MATRICES

Vectors are the fundamental data type in R. Often, vectors are just a collection of data of a similar type, either numeric (e.g., 17.5), integer (e.g., 2), or character (e.g., "low"). The simplest type of vector is a single value, sometimes referred to as a **scalar** in other programming languages:

```
w <- 5
```

We can create longer one-dimensional vectors in R like this:

```
x <- c(2, 3, 6, w, w + 7, 12, 14)
```

What is the function `c()` that we just used to create a vector? The `help()` function is your friend. Let's try it out:

```
help(c)
```

What happens when you multiply a vector by a "scalar"?

```
y <- w * x
```

What happens when you multiply two vectors of the same length?

```
z <- x * y
```

You may need to reference a specific **element** in a vector. We will do this using the square brackets. In this case, the number inside of the square brackets tells R that we want call the second element of vector `z`:

```
z[2]
```

```
## [1] 45
```

You can also reference multiple elements in a vector:

```
z[2:5]
```

```
## [1]  45 180 125 720
```

In some instances, you may want to change the value of an element in a vector. Here's how you can substitute a new value for the second element of `z`:

```
z[2] <- 583
```

It's pretty easy to perform summary statistics on a vector using the built-in fuctions of R:

```
max(z)    # maximum
```

```
## [1] 980
```

```
min(z)    # minimum
```

```
## [1] 20
```

```
sum(z)    # sum
```

```
## [1] 3328
```

```
mean(z)   # mean
```

```
## [1] 475.4
```

```
median(z) # median
```

```
## [1] 583
```

```
var(z)    # variance
```

```
## [1] 133881
```

```
sd(z)     # standard deviation
```

```
## [1] 365.9
```

What happens when you take the standard error of the mean (`sem`) of z?

The standard error of the mean is defined as $SEM = \frac{sd(x)}{\sqrt{n}}$. This function does not exist in the base package of R. Therefore, you need to write your own functions sometimes. Let's give it a try:

```
sem <- function(x, ...){
  sd(x, ...)/sqrt(length(na.omit(x)))
  }
```

There are number of functions inside of **sem**. Take a moment to think about and describe what is going on here.

Often, datasets have missing values (designated as 'NA' in R):

```
i <- c(2, 3, 9, NA, 120, 33, 7, 44.5)
```

What happens when you apply your **sem** function to vector i?

One solution is to tell R to remove NA from the dataset:

```
sem(i, na.rm = TRUE)
```

```
## [1] 16.03
```

Matrices are just two-dimensional vectors containing data of the same type (e.g., numeric, integer, character).

There are three common ways to create a matrix (two dimensional vectors) in R.

Approach 1 is to combine (or concatenate) two or more vectors. Let's start by creating a one-dimensional vector using a new function **rnorm**.

<– seems odd to introduce this here. Also, why not NAs? – MEM <– Doesn't seem odd to me, but I'm defintiely open to discuss – JTL <– Were you wanting to build from the idea of NAs or something? – JTL

```
j <- c(rnorm(length(z), mean = z))
```

What does the **rnorm** function do? What are arguments doing? Remember your friend **help**!

Now we will use the function **cbind** to create a matrix from the two one-dimensional vectors:

```
k <- cbind(z, j)
```

Use the **help** function to learn about **cbind**. Use the **dim** function to describe the matrix you just created. What did you learn from this?

Approach 2 to making a matrix is to use the **matrix** function along with arguments that specify the number of rows (nrow) and columns (ncol):

```
l <- matrix(c(2, 4, 3, 1, 5, 7), nrow = 3, ncol = 2)
```

Approach 3 to making a matrix is to import or *load a dataset* from your working directory:

```
m <- as.matrix(read.table("matrix.txt", sep = "\t", header = FALSE))
```

In this case, we're reading in a tab-delimited file. The name of your file must be in quotes, and you need to specify tab-limited file type using the **sep** argument. The **header** argument tells R whether or not the names of the variables are contained in the first line; in the current example, they are not.

Often, when handling datasets, we want to be able to transpose a matrix. This is an easy operation in R:

```
n <- t(m)
```

Confirm the transposition using the `dim` function.

Frequently, you will need to **index** or retrieve a certain portion of a matrix. As with the vector example above, we will use the square brackets to retrieve data from a matrix. Inside the square brackets, there are now two subscripts corresponding to the rows and columns, respectively, of the matrix.

For example, the following code will create a new matrix (`n') based on the first three rows of matrix` (m`):

```
n <- m[1:3, ]
```

Or maybe you want the first two columns of a matrix:

```
n <- m[, 1:2]
```

Or perhaps you want non-sequential columns of a matrix. How do we do that?

It's easy when you understand how to reference data within a matrix:

```
n <- m[, c(1:2, 5)]
```

Describe what we just did in the last indexing.

BASIC DATA VISUALIZATION AND STATISTICAL ANALYSIS

In the following exercise, we will use a dataset from Lennon et al. (2003), which looked at zooplankton community assembly along an experimental nutrient gradient. Inorganic nitrogen and phosphorus were added to mesocosms for six weeks at three different levels (low, medium, and high), but we also direcly measured nutrient concentrations in the mesocosms. So we have categorical and continuous predictors that we're going to use to help explain variaiton in zooplankton biomass.

The first thing we're going to do is load the data:

```
meso <- read.table("zoop_nuts.txt", sep = "\t", header = TRUE)
```

Let's use the `str` function to look at the structure of the data.

```
str(meso)
```

```
## 'data.frame':    24 obs. of  8 variables:
##  $ TANK: int  34 14 23 16 21 5 25 27 30 28 ...
##  $ NUTS: Factor w/ 3 levels "high","low","medium": 2 2 2 2 2 2 2 2 3 3 ...
##  $ TP  : num  20.3 25.6 14.2 39.1 20.1 ...
##  $ TN  : num  720 750 610 761 570 ...
##  $ SRP : num  4.02 1.56 4.97 2.89 5.11 4.68 5 0.1 7.9 3.92 ...
##  $ TIN : num  131.6 141.1 107.7 71.3 80.4 ...
##  $ CHLA: num  1.52 4 0.61 0.53 1.44 1.19 0.37 0.72 6.93 0.94 ...
##  $ ZP  : num  1.781 0.409 1.201 3.36 0.733 ...
```

How does this dataset differ from the 'm' dataset above? We're now dealing with a new type of **data structure**. Specifically, the `meso` dataset is a **data frame** since it has a combination of numeric and character data.

Here is a description of the column headers: -TANK = unique mesocosm identifier

-NUTS = categorical nutrient treament

-TP = total phosphorus concentration (µg/L)

-TN = total nitrogen concentration (µg/L)

-SRP = soluble reactive phosphorus concentration (µg/L)

-TIN = total inorganic nutrient concentration (µg/L)

-CHLA = chlorophyll *a* concentration (proxy for algal biomass; µg/L)

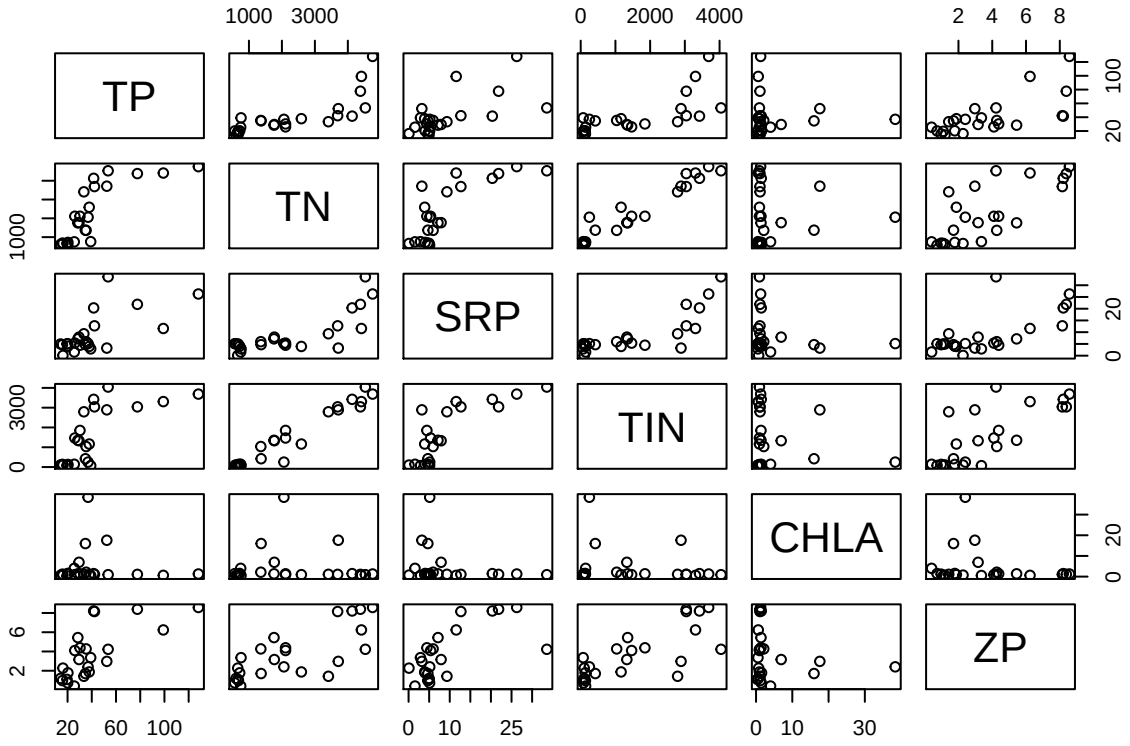
-ZP = zooplankton biomass (mg/L)

As part of iniital data exploration, we often look for correlations among our measured variables. Before we do this, let's index our numerical (continuous) data in the 'meso' dataframe. (correlations don't typically work well on categorical data.)

```
meso.num <- meso[,3:8]
```

We can conveniently visualize bi-plots of the data using the following command:

```
pairs(meso.num)
```



Now let's conduct a simple correlation analysis with the R `cor()` function.

```
cor1 <- cor(meso.num)
```

Describe what you found from the visualization and correlation anaylysis above?

The base pakcage in R won't always meet all of our needs. This is why there are > 6,000 **contributed packages** that have been developed for R. This may seem overwhelming, but it also means that there are tools (and web support) for just about any problem you can think of.

When using one of the contributed packages, the first thing we need to do is **install** them along with their dependencies (other packages). We're going to start out by using the 'psych' package, which has many features, but we're going to use it specifically for the `corr.test` function, which generates p-values for each pairwise correlation. (For whatever reason, the `cor` function in the base package does not provide p-values.)

```
require("psych")||install.packages("psych");require("psych") # require? if not, install and then requir
```

```
## Loading required package: psych
```

```
## [1] TRUE
```

Now, let's look at the correlations among variables and assess whether they are signficant

```
cor2 <- corr.test(meso.num, method = "pearson", adjust = "BH")
print(cor2, digits = 3)
```

```
## Call:corr.test(x = meso.num, method = "pearson", adjust = "BH")
## Correlation matrix
##          TP     TN    SRP    TIN   CHLA     ZP
## TP    1.000  0.787  0.654  0.717 -0.017  0.697
## TN    0.787  1.000  0.784  0.969 -0.004  0.756
## SRP   0.654  0.784  1.000  0.801 -0.189  0.676
## TIN   0.717  0.969  0.801  1.000 -0.157  0.761
## CHLA -0.017 -0.004 -0.189 -0.157  1.000 -0.183
## ZP    0.697  0.756  0.676  0.761 -0.183  1.000
## Sample Size
## [1] 24
## Probability values (Entries above the diagonal are adjusted for multiple tests.)
##         TP    TN   SRP   TIN  CHLA    ZP
## TP   0.000 0.000 0.001 0.000 0.983 0.000
## TN   0.000 0.000 0.000 0.000 0.983 0.000
## SRP  0.001 0.000 0.000 0.000 0.491 0.000
## TIN  0.000 0.000 0.000 0.000 0.536 0.000
## CHLA 0.938 0.983 0.376 0.464 0.000 0.491
## ZP   0.000 0.000 0.000 0.000 0.393 0.000
##
##  To see confidence intervals of the correlations, print with the short=FALSE option
```

Notes on corr.test:

a) for rank-based correlations (i.e., non-parametric), use method = "kendall" or "spearman". Give it a try!

b) the adjust = "BH" statement supplies the Benjamini & Hochberg-corrected p-values in the upper right diagonal of the square matrix; the uncorrected p-values are below the diagonal. This process corrects for **false discovery rate**, which arises whe making multiple comparisons.

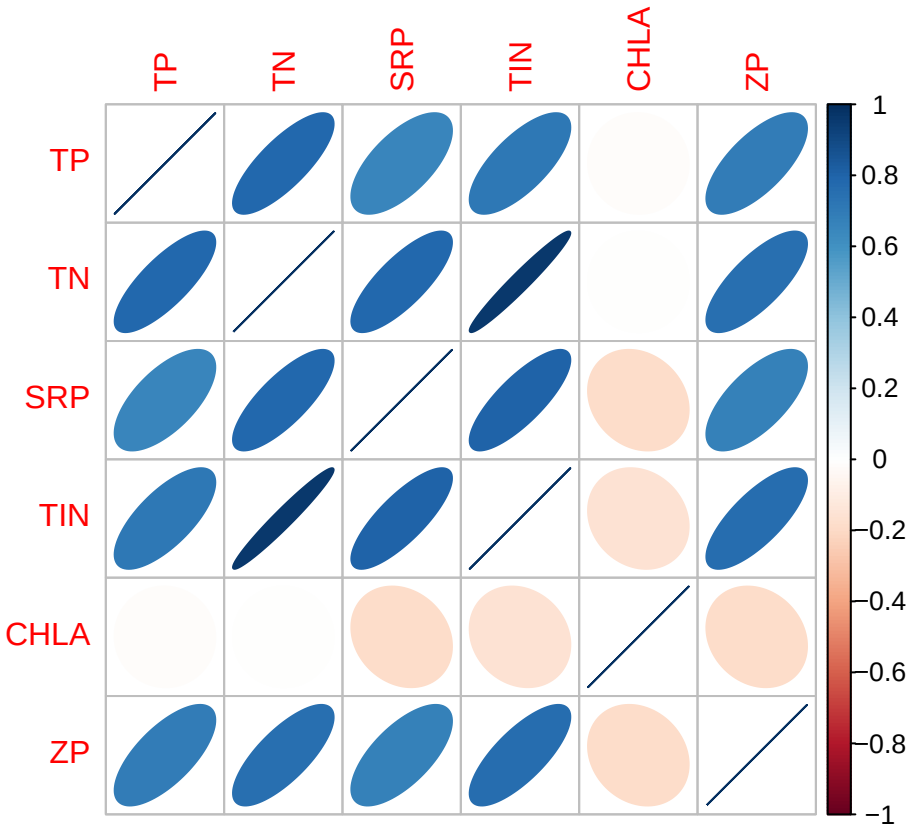
Let's load another package now that will let us visualize the sign and strength of the correlations:

```
require("corrplot")||install.packages("corrplot");require("corrplot")
```

```
## Loading required package: corrplot
```

```
## [1] TRUE
```

```
corrplot(cor1, method = "ellipse")
```



It seems that TN is a fairly good predictor of ZP and this is something that we directly manipulated. So, let's try some linear regression using the `lm` function:

```
fit <- lm(ZP ~ TN, data = meso)
```

Let's examine the output of our regression model:

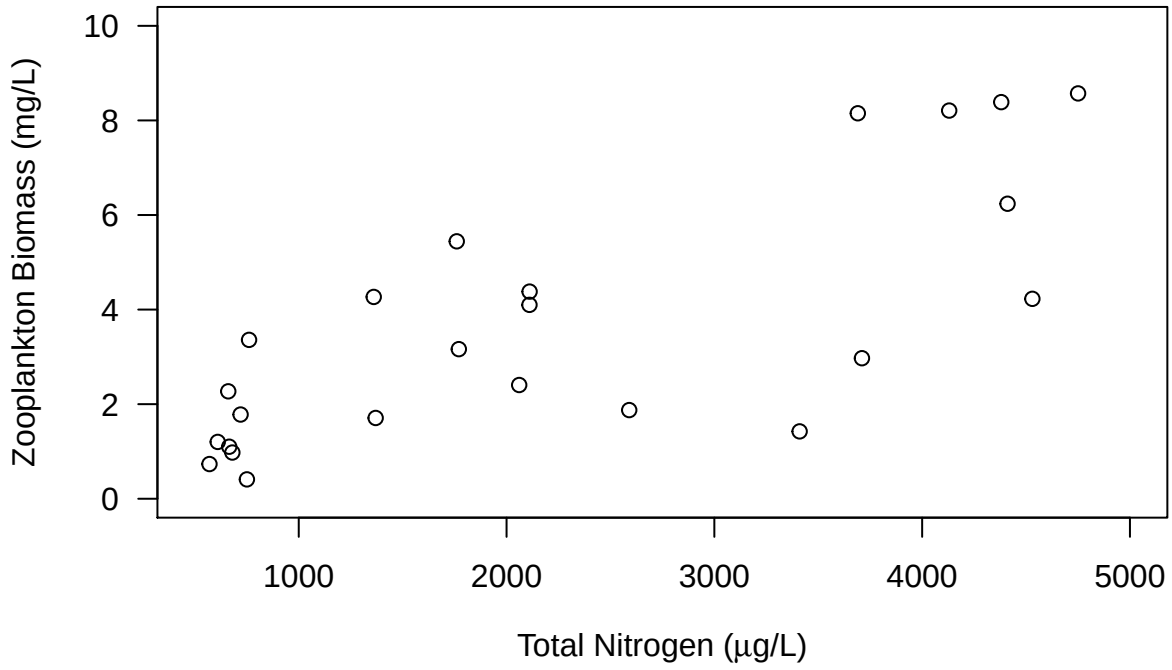
```
summary(fit)
```

```
##
## Call:
## lm(formula = ZP ~ TN, data = meso)
##
## Residuals:
```

```
##    Min     1Q Median     3Q    Max
## -3.769 -0.849 -0.071  1.624  2.589
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.697771   0.649631    1.07     0.29
## TN          0.001318   0.000243    5.42  1.9e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.75 on 22 degrees of freedom
## Multiple R-squared:  0.572,  Adjusted R-squared:  0.552
## F-statistic: 29.4 on 1 and 22 DF,  p-value: 1.91e-05
```

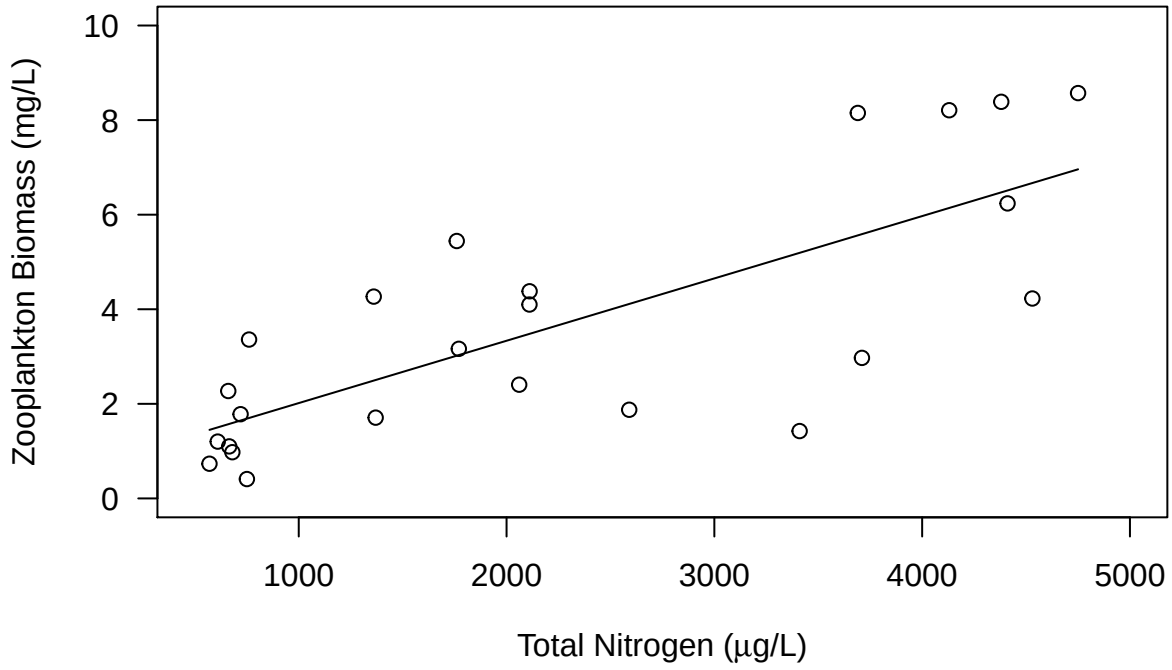
Now, let's look at a plot of the data used in our regression:

```
TN <- meso$TN
ZP <- meso$ZP
plot(TN,ZP,ylim=c(0,10),xlim=c(500,5000), xlab=expression(paste("Total Nitrogen (", mu,"g/L)")),
     ylab="Zooplankton Biomass (mg/L)",las=1)
```



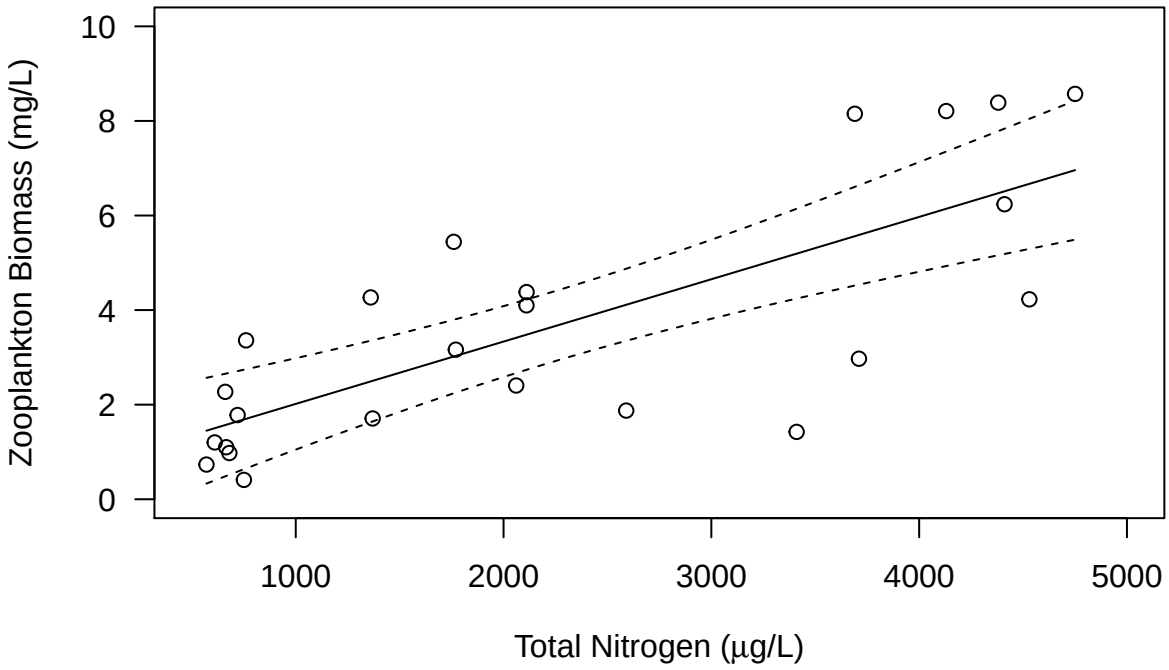
To add the regression line, the first thing we need to do is generate a range of x-values and then generate the corresponding predicted values from our regression model:

```
ZP <- meso$ZP
plot(TN,ZP,ylim=c(0,10),xlim=c(500,5000), xlab=expression(paste("Total Nitrogen (", mu,"g/L)")),
     ylab="Zooplankton Biomass (mg/L)",las=1)
newTN <- seq(min(TN),max(TN),10)
regline <- predict(fit,newdata=data.frame(TN=newTN))
lines(newTN,regline)
```



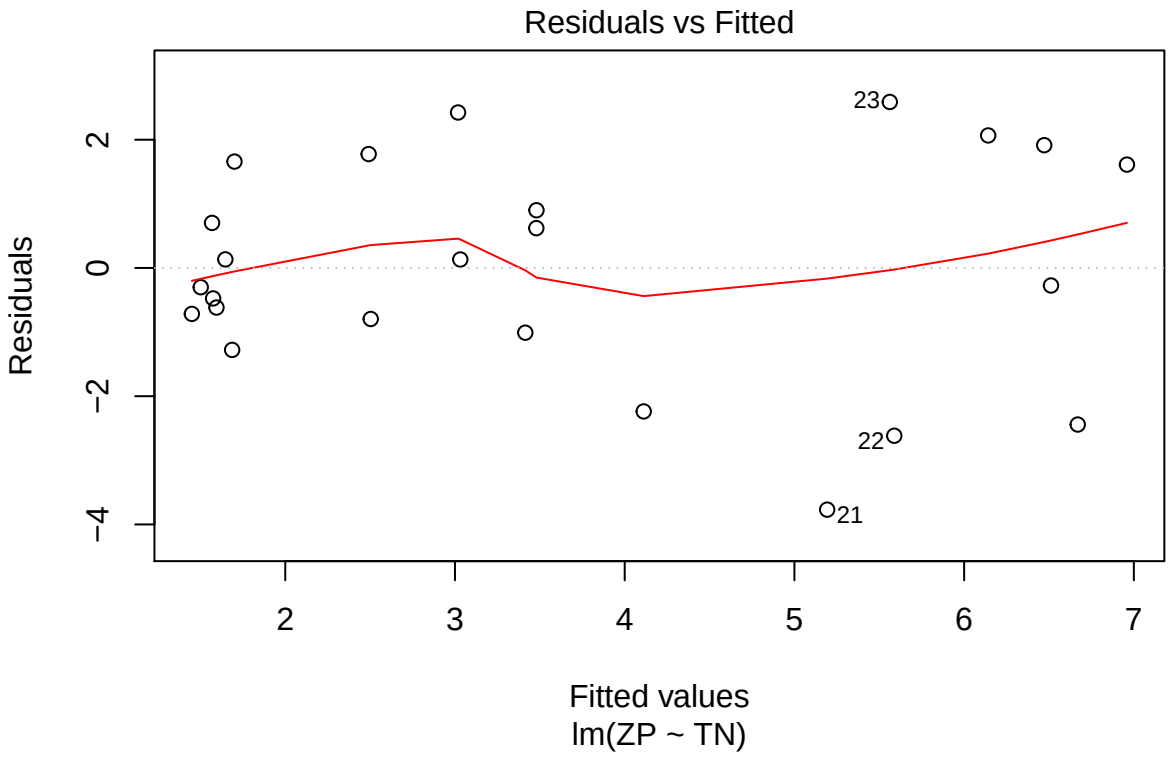
Now lets create and plot the 95% confidence intervals using the same procedure as above, that is, use newTN to generate corresponding confidence intervals from our regression model:

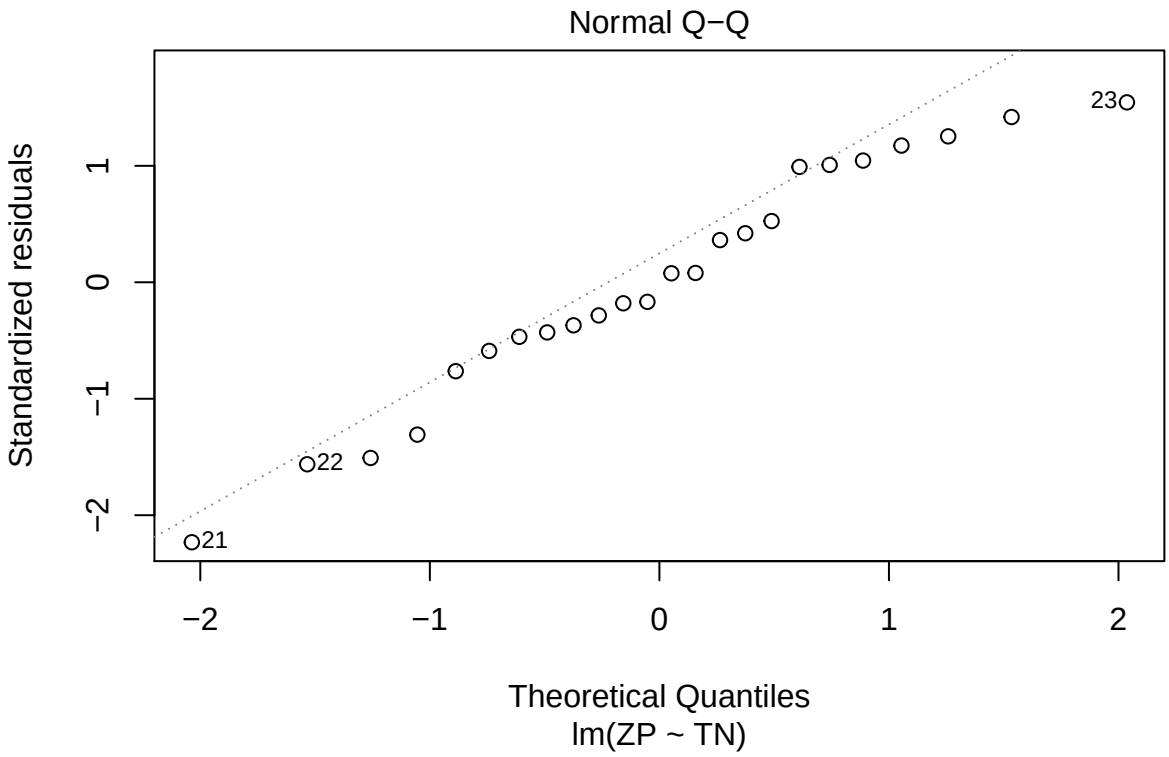
```
ZP <- meso$ZP
plot(TN,ZP,ylim=c(0,10),xlim=c(500,5000), xlab=expression(paste("Total Nitrogen (", mu,"g/L)")),
     ylab="Zooplankton Biomass (mg/L)",las=1)
newTN <- seq(min(TN),max(TN),10)
regline <- predict(fit,newdata=data.frame(TN=newTN))
lines(newTN,regline)
conf95 <- predict(fit,newdata=data.frame(TN=newTN),interval=c("confidence"),level=0.95,type="response")
matlines(newTN,conf95[,c("lwr","upr")],type="l",lty=2,lwd=1,col="black")
```

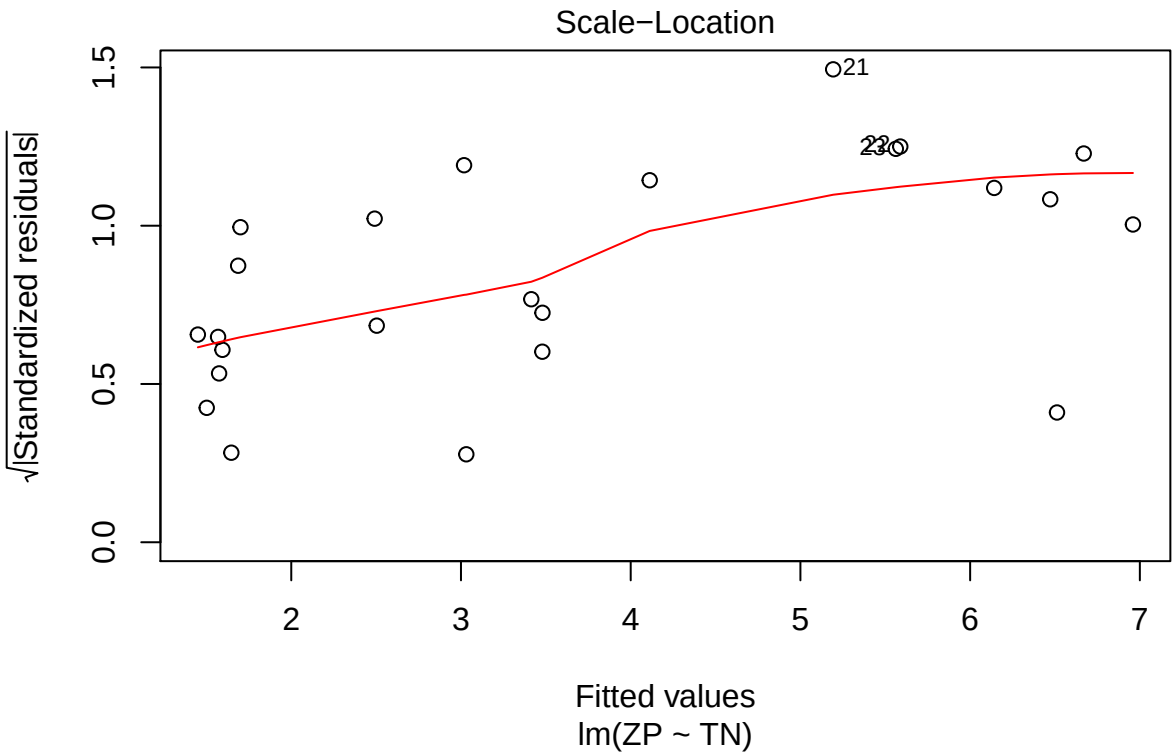


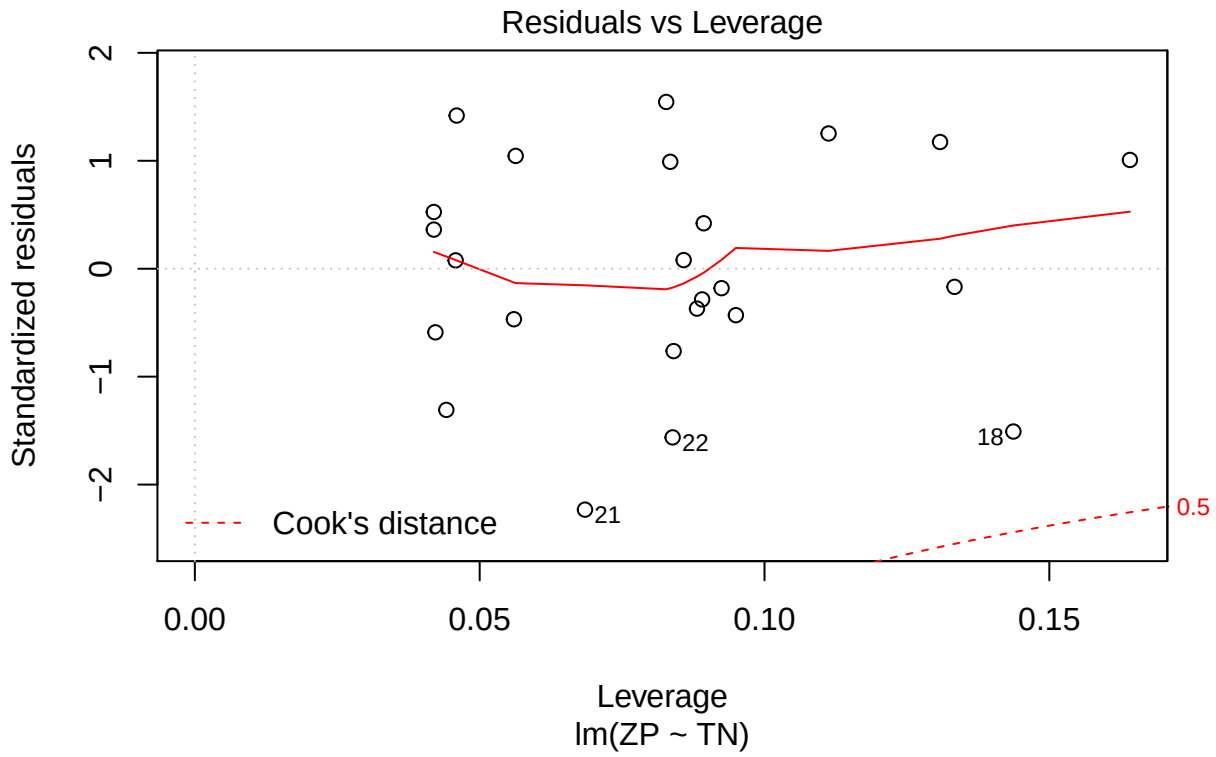
We should also look at the residuals (i.e., observed values - predicted values) to see if our data meet the assumptions of linear regression. Specifically, we want to make sure that the residuals are normally distributed and that they are homoskedastic. We can look for patterns in our residuals using the following diagnostics:

```
plot(fit)
```


Residuals vs Fitted
Residuals
Fitted values
lm(ZP ~ TN)
23
22
21
2
0
−2
−4
2
3
4
5
6
7


Normal Q−Q
Standardized residuals
Theoretical Quantiles
lm(ZP ~ TN)
21
22
23
−2
−1
0
1
2


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
21
2
3
4
5
6
7
Fitted values
lm(ZP ~ TN)



We also have the option of looking at the relationship between zooplankton and nutrients where the manipulation is treated categorically (low, medium, high). First, let's order the categorical nutrient treatments:

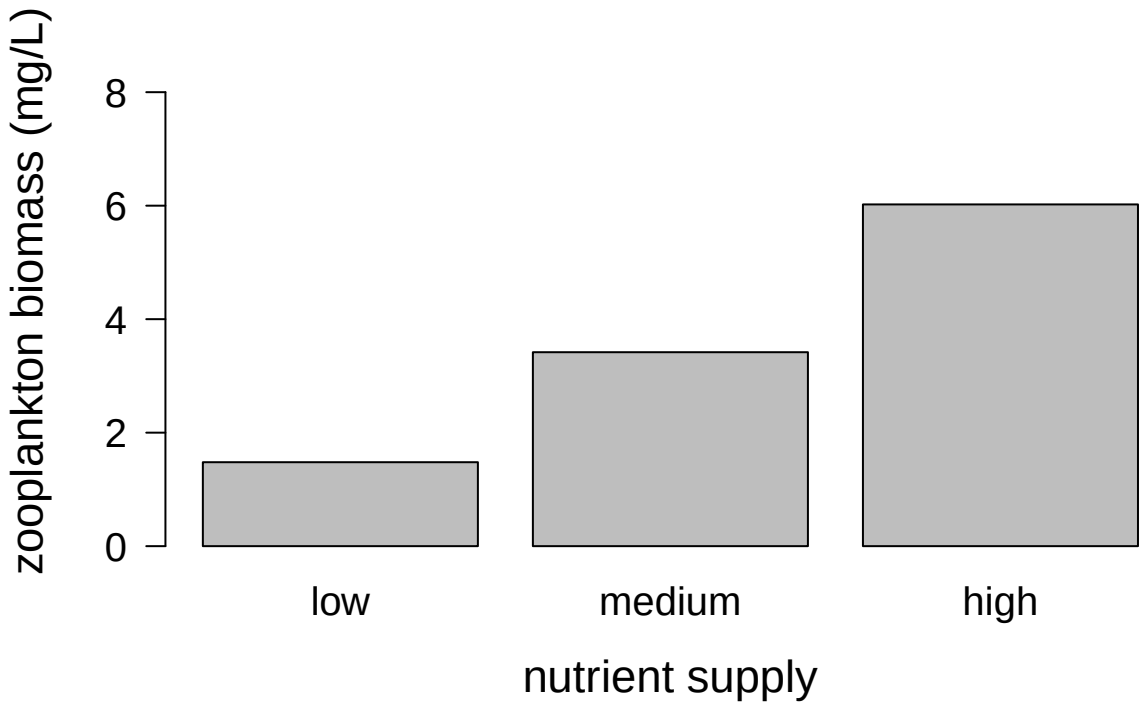
```
NUTS <- factor(meso$NUTS,levels = c('low','medium','high'))
```

Before plotting, we need to calcualte the means and standard errors for zooplankton biomass in our nutrient treatments:

```
sem <- function(x, ...){
  sd(x, ...)/sqrt(length(na.omit(x)))
}
zp.means <- tapply(ZP, NUTS, mean)
zp.sem <- tapply(ZP, NUTS, sem)
```

Now let's make the barbplot:

```
bp <- barplot(zp.means, ylim=c(0,round(max(ZP), digits=0)), pch=15, cex=1.25, las=1, xlab="nutrient sup
```




```
bp
```

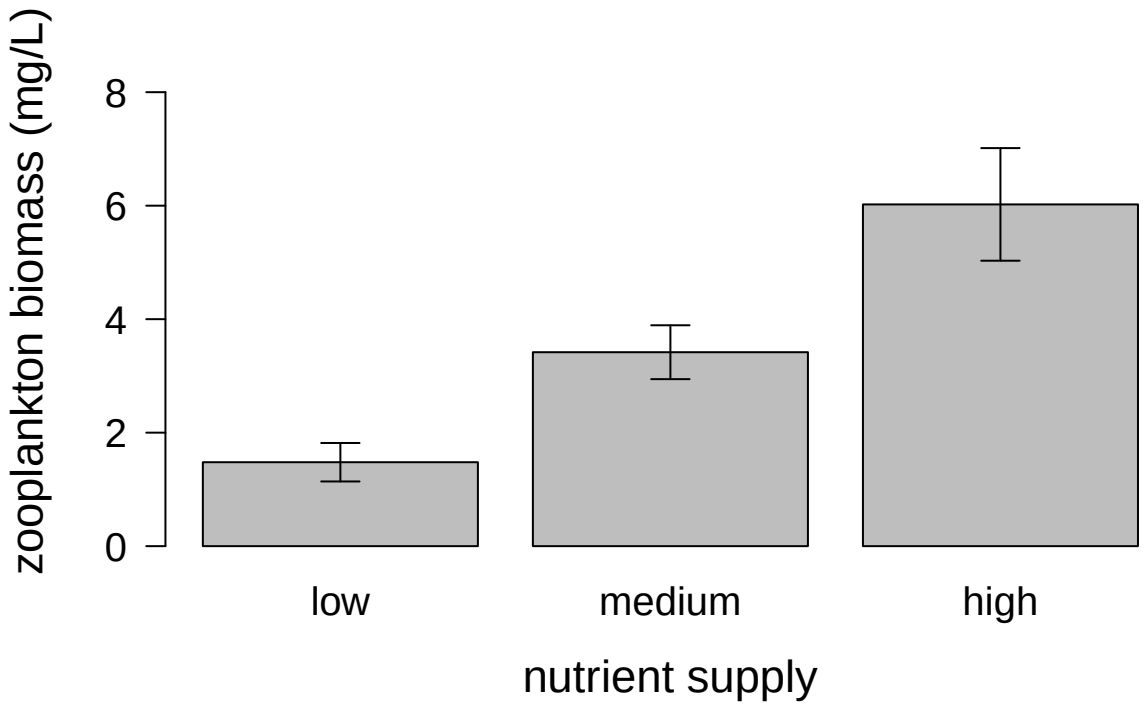
```
##      [,1]
## [1,]  0.7
## [2,]  1.9
## [3,]  3.1
```

We need to add the error bars (+/- sem) as follows:

```
bp
```

```
##      [,1]
## [1,]  0.7
## [2,]  1.9
## [3,]  3.1
```

```
arrows(x0 = bp, y0 = zp.means, y1 = zp.means - zp.sem, angle = 90,
       length=0.1, lwd = 1) #
arrows(x0 = bp, y0 = zp.means, y1 = zp.means + zp.sem, angle = 90,
       length=0.1, lwd = 1)
```



—> Follow up with anova model and post hoc comparison of means